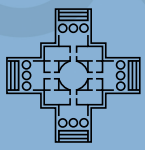
The Karlsruhe Series on Software Design and Quality

26



Implicit Incremental Model Analyses and Transformations

Georg Hinkel

KIT Scientific Publishing

Georg Hinkel

Implicit Incremental Model Analyses and Transformations

The Karlsruhe Series on Software Design and Quality
Volume 26

Chair Software Design and Quality
Faculty of Computer Science
Karlsruhe Institute of Technology

and

Software Engineering Division
Research Center for Information Technology (FZI), Karlsruhe

Editor: Prof. Dr. Ralf Reussner

Implicit Incremental Model Analyses and Transformations

by
Georg Hinkel

KIT Scientific Publishing

Karlsruher Institut für Technologie
Institut für Programmstrukturen und Datenorganisation

Implicit Incremental Model Analyses and Transformations

Zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften von der KIT-Fakultät für Informatik des Karlsruher Instituts für Technologie (KIT) genehmigte Dissertation

von Dipl.-Ing. Georg Hinkel

Tag der mündlichen Prüfung: 29. November 2017
Erster Gutachter: Prof. Dr. Ralf H. Reussner
Zweiter Gutachter: Prof. Dr. Uwe Assmann (TU Dresden)

Impressum

Scientific Publishing

Karlsruher Institut für Technologie (KIT)
KIT Scientific Publishing
Straße am Forum 2
D-76131 Karlsruhe

KIT Scientific Publishing is a registered trademark
of Karlsruhe Institute of Technology.
Reprint using the book cover is not allowed.

www.ksp.kit.edu

Print on Demand 2021 – Gedruckt auf FSC-zertifiziertem Papier

ISSN 1867-0067
ISBN 978-3-7315-0763-5
DOI: 10.5445/KSP/1000080522

Abstract

In many engineering disciplines, abstract models are used to describe systems on a high level of abstraction. On this abstract level, it is often easier to gain insights about that system that is being described.

When models of a system change – for example because the system itself has changed – any analyses based on these models have to be invalidated and thus have to be reevaluated again in order for the results to stay meaningful. In many cases, the time to get updated analysis results is critical. However, as most often only small parts of the model change, large parts of this reevaluation could be saved by using previous results but such an incremental execution is barely done in practice as it is non-trivial and error-prone.

The approach of implicit incrementalization offers a solution by deriving an incremental evaluation strategy implicitly from a batch specification of the analysis. This works by deducing a dynamic dependency graph that allows to only reevaluate those parts of an analysis that are affected by a given model change. Thus advantages of an incremental execution can be gained without changes to the code that would potentially degrade its understandability.

However, current approaches to implicit incremental computation only support narrow classes of analysis, are restricted to an incremental derivation at instruction level or require an explicit state management. In addition, changes are only propagated sequentially, meanwhile modern multi-core architectures would allow parallel change propagation. Even with such improvements, it is unclear whether incremental execution in fact brings advantages as changes may easily cause butterfly effects, making a reuse of previous analysis results pointless (i.e. inefficient).

This thesis deals with the problems of implicit incremental model analyses by proposing multiple approaches that mostly can be combined. Further, the thesis suggests a new formalism how this incrementalization system can be used to empower incremental, uni- or bidirectional model transformations.

To define and ensure the correctness of the resulting incremental evaluation strategy, the thesis presents a formalization of the incrementalization process using functors from category theory.

A first approach as a direct consequence of the formalization allows an incrementalization at the level of method calls such that often used methods can be annotated with an optimized incremental execution algorithm. By extending the functor to distributed computing, memory problems can be resolved.

A second approach simplifies dynamic dependency graphs by generalizing model changes and thus summarizing parts of the analysis using several strategies. The selection of strategies gives the developer a chance to adjust the incrementalization process to a given scenario. Alternatively, an automated design space exploration can be performed to find a (Pareto-)optimal configuration with regard to memory consumption and response time of the analysis to a given model change.

The combination of these approaches improves the incrementalization process such that it never gets worse than batch execution but in many cases gains significant speedups. Generic operators to be reused in many analyses can be optimized by choosing appropriate algorithms whereas complex domain logic can be optimized for incremental execution by the incrementalization system. The implicit nature of the overall approach allows these improvements to happen automatically and transparent to the developer of the analysis.

Although the presented approach is Turing-complete and therefore universally applicable, especially in the context of model-driven engineering a special class of artifacts deserves a special investigation as they are usually hard to describe with general-purpose languages: Model transformations. Therefore, we propose a new formalism and a language to describe uni- or bidirectional model transformations in a way that they can profit from the incrementalization system for analyses. For this formalism, we can proof a correct and hippocratic synchronization process.

All approaches have been implemented and integrated into the .NET Modeling Framework that supports developers in model-driven engineering. The advantages of all approaches regarding performance are validated using seven case studies. In particular, we use five case studies from the Transformation Tool Contest (TTC) from the years 2015 to 2017 for which also

solutions using other tools are available. The expressiveness of the model transformation approach is validated by a higher-order transformation from the commonly used ATL transformation language into the language presented in this thesis. Using this transformation, also the performance of model transformations is compared to ATL for a number of model transformations.

The results of the case studies show that especially for the application of the proposed incrementalization systems to model transformation, significant speedups compared to classic model transformations, but also compared to existing incremental model transformation languages can be achieved, often by multiple orders of magnitude. In particular, the case studies suggest that time to propagate a change in the source model is often independent from the size of the input model. Especially for large input models, this causes large speedups.

The incrementalization of a model analysis is always bound to the metamodel. However, metamodels used in practice only use a subset of the commonly used modeling standard MOF. This sometimes causes a high accidental complexity of the metamodel and necessitates a range of analyses. An extension of the modeling language can help to make several model analyses obsolete an simplify others, thereby also improving the performance characteristics of these analyses.

Zusammenfassung

In vielen Ingenieursdisziplinen werden Modelle verwendet, um Systeme verschiedenster Art auf einem hohen Abstraktionsgrad zu beschreiben. Auf diesem Abstraktionsgrad ist es häufig einfacher, Aussagen über den Zustand des Systems zu treffen.

Wenn sich Modelle eines Systems ändern – beispielsweise, weil sich das System selbst geändert hat – müssen Analysen auf Grundlage dieses Modells jedoch neu berechnet werden, um weiterhin gültig zu sein. In vielen Fällen ist diese Neuberechnung der Analyseergebnisse zeitkritisch. Da sich oft nur kleine Teile des Modells ändern, könnten zwar große Teile des letzten Analysedurchlaufs durch eine inkrementelle Ausführung der Analyse wiederverwendet werden, in der Praxis ist eine solche Inkrementalisierung aber nicht trivial und oft fehleranfällig.

Eine Lösungsmöglichkeit für dieses Problem bietet der Ansatz der impliziten Inkrementalisierung, bei der ein inkrementeller Algorithmus für eine gegebene Analyse aus der Batch-Spezifikation abgeleitet wird. Aus der Spezifikation wird ein dynamischer Abhängigkeitsgraph konstruiert, der es erlaubt, nur die Teile einer Analyse neu auszuwerten, die von einer Änderung tatsächlich betroffen sind. Damit lassen sich Vorteile einer Inkrementalisierung nutzen, ohne dass der Code angepasst werden muss und die Lesbarkeit des Analysecodes leidet.

Leider unterstützen derzeitige Verfahren für implizite Inkrementalisierung nur eine bestimmte Klasse von Analysen, sind auf eine Inkrementalisierung auf Ebene von einzelnen Instruktionen beschränkt oder benötigen eine explizite Zustandsverwaltung. Auch mit diesen Verbesserungen ist unklar, in welchen Fällen eine Inkrementalisierung Vorteile bringt, da in einigen Szenarien Änderungen Schmetterlingseffekte verursachen können und eine Wiederverwertung des letzten Analysedurchlaufs keinerlei Beschleunigungspotential hat.

Diese Dissertation behandelt diese Probleme bei impliziter Inkrementalisierung von Modellanalysen mittels mehrerer Verfahren, die größtenteils kombinierbar sind. Desweiteren wird ein neuer Formalismus vorgestellt, mit dessen Hilfe Inkrementalisierungssysteme auch für uni- oder bidirektionale Modelltransformationen einsetzbar sind. Um die Korrektheit der entstehenden inkrementellen Modellanalysen zu definieren und zu zeigen, wird Inkrementalisierung in Kategorientheorie als Funktor beschrieben.

Ein erstes Verfahren ermöglicht als direkte Konsequenz der formalen Darstellung die Inkrementalisierung auf Ebene von Methodenaufrufen, sodass für häufig verwendete Operatoren eine optimierte Inkrementalisierung zur Verfügung gestellt werden kann. Durch Erweiterung des Funktors auf Verteilung lassen sich auf ähnliche Weise auch etwaige Speicherprobleme lösen.

Ein zweites Verfahren vereinfacht die entstehenden dynamischen Abhängigkeitsgraphen, indem Teile der Analyse durch eine generalisierte Betrachtung von Modelländerungen mit mehreren Strategien zusammengefasst werden können. Die Auswahl der Strategien ermöglicht dem Entwickler eine Anpassung der Inkrementalisierung auf einen konkreten Anwendungsfall. Alternativ kann für ein gegebenes Szenario auch durch automatische Entwurfsraumexploration eine (Pareto-) optimale Konfiguration hinsichtlich Speicherverbrauch und Antwortzeit der Aktualisierung eines Analyseergebnisses nach einer Modelländerung gefunden werden.

Die Kombination dieser Verfahren ermöglicht es, die Performanz von Inkrementalisierungen so zu verbessern, dass diese bis auf einmalige Initialisierung nie schlechter ist als die batchmäßige Wiederholung der Analyse, in vielen Fällen aber teils deutlich schneller sein kann. Generische Operatoren, die in vielen Modellanalysen wiederverwendet werden, können für die Inkrementalisierung durch geeignete Algorithmen spezifisch optimiert werden, während komplexe Domänenlogik durch das System optimiert werden kann. Durch den impliziten Ansatz geschehen diese Verbesserungen vollautomatisch und transparent für den Entwickler der Modellanalyse.

Obwohl der so geschaffene Ansatz Turing-mächtig und somit universell einsetzbar ist, gibt es doch gerade in der modellgetriebenen Entwicklung eine Klasse von Artefakten, die eine besondere Betrachtung erfordern, da sie sich im Allgemeinen nur schwer mit gewöhnlichen objekt-orientierten Sprachen beschreiben lassen: Modelltransformationen. Daher wird in dieser Dissertation ein neuer Formalismus und eine darauf aufbauende Sprache vorgestellt, die Modelltransformationen so beschreiben, dass diese leicht mit

Hilfe eines Inkrementalisierungssystems inkrementell ausgeführt werden können. Die Synchronisierung einer Modelländerung ist hierbei bewiesen korrekt und hippokratisch.

Alle Verfahren wurden implementiert und in das .NET Modeling Framework integriert, welches Entwickler auf der .NET Plattform bei der modellgetriebenen Entwicklung unterstützen soll. Die entstandenen Vorteile aller Verfahren hinsichtlich Performanz werden anhand von sieben Fallstudien in verschiedenen Domänen validiert. Insbesondere werden hierzu fünf Fallstudien des Transformation Tool Contests (TTC) der Jahre 2015 bis 2017 herangezogen, für die auch mit anderen Ansätzen verfasste Lösungen zur Verfügung stehen. Die Ausdrucksmächtigkeit der Modelltransformationssprache wird durch eine Transformation der in der modellgetriebenen Entwicklung weit verbreiteten Transformationssprache ATL in die neu geschaffene Transformationssprache validiert. Mithilfe dieser Transformation wird weiterhin die Ausführungsgeschwindigkeit von Modelltransformationen mit der von ATL in einigen Modelltransformationen verglichen.

Die Ergebnisse aus den Fallstudien zeigen gerade bei der Anwendung des Inkrementalisierungssystems auf Modelltransformationen deutliche Performance-Steigerungen im Vergleich zu herkömmlichen Modelltransformationen, aber auch gegenüber anderen inkrementellen Modelltransformationssprachen zeigt der vorgestellte Ansatz deutliche Beschleunigungen, teils um mehrere Größenordnungen. Insbesondere weisen die Fallstudien darauf hin, dass die benötigte Zeit für die Propagation von Änderungen des Eingabemodells in vielen Fällen unabhängig von der Größe des Eingabemodells ist. Gerade bei großen Eingabemodellen kommen so sehr hohe Beschleunigungen zustande.

Die Inkrementalisierung einer Analyse ist dabei immer an das Metamodell gebunden. In der Praxis verwenden aber die meisten eingesetzten Metamodelle nur den eingeschränkten Modellierungsstandard EMOF, der teilweise zu einer unnötigen Komplexität des Metamodells führt und viele Analysen überhaupt erst notwendig macht. Eine Erweiterung des Modellierungsstandards kann hier einige Klassen von Modellanalysen komplett überflüssig machen und andere Analysen deutlich vereinfachen, sowie auch die Performance der entsprechenden Analyse beschleunigen.

Contents

Abstract i

Zusammenfassung v

List of Figures xvii

List of Tables xxiii

Listings xxv

I. Prologue 1

1. Introduction 3
1.1. Motivation 3
1.2. Assumptions and Definitions (A) 9
1.3. Research Questions (RQ) 12
1.4. Contribution Summary (C) 17
1.5. Validation Overview 20
1.6. Running Example 21
1.7. Structure 24

2. Foundations 27
2.1. Model-driven engineering 27
2.1.1. Metamodels 28
2.1.2. Editors 28
2.1.3. Model Transformations 29
2.1.4. The .NET Modeling Framework 31
2.1.5. NMeta 31
2.2. Incrementality 32

2.3. Advanced C# Language Features . . . 35
2.3.1. Expression Trees . . . 35
2.3.2. Language Integrated Query and the C# Query Syntax . . . 37
2.4. Virtual Actors . . . 39
2.5. The Palladio Component Model (PCM) . . . 40
2.6. Partially Ordered Sets and Lattices . . . 44
2.7. Lenses . . . 45
2.8. A very short Primer to Category Theory . . . 46
2.8.1. Categories . . . 46
2.8.2. Functors and Monads . . . 50

3. Mutable Type Categories . . . 55
3.1. Types and State as a Category . . . 55
3.2. Stateless Types and Stateless Methods . . . 61
3.3. Inheritance . . . 62
3.4. Collections . . . 63
3.5. Composition . . . 72
3.6. Opposites . . . 73

II. Implicit Incremental Model Analyses . . . 77

4. Efficient Incremental Computation Systems . . . 79
4.1. Incrementalization as a Functor . . . 80
4.2. Integrating Dynamic Algorithms into Incremental Analyses . . . 87
4.2.1. Choice of Algorithms . . . 87
4.2.2. Reification of the Problem for Incrementalization . . . 89
4.3. An Extensible Implicit Incremental Computation System . . . 91
4.3.1. Overview . . . 92
4.3.2. Incrementalization at Instruction Level . . . 94
4.3.3. Incrementalization of Higher-order Functions . . . 97
4.3.4. Extensibility . . . 98
4.4. Incremental Queries as an Example Extension . . . 99

4.5. Distributed Incrementality using Virtual Actors . . . 101
4.5.1. Distributed Incremental Analyses through Virtual Actors . . . 101
4.5.2. Application Model . . . 102
4.5.3. Mapping Incremental Analyses to Virtual Actors . . . 104
4.6. Summary . . . 107

5. Using Containments to Optimize Incremental Model Analyses . . . 109
5.1. Covering Triggers for Incremental Values . . . 110
5.2. Generalization of Model Changes . . . 116
5.3. Obtaining Approximate Trigger Coverages . . . 119
5.4. Incrementalization Strategies . . . 121
5.4.1. Instruction-Level . . . 121
5.4.2. Argument Promotion . . . 121
5.4.3. Reaction to Repository Changes . . . 123
5.4.4. Tree Extension . . . 124
5.5. Finding Pareto-Optimal Configurations . . . 124
5.6. Incerator . . . 127
5.7. Summary . . . 129

III. Implicit Incremental Model Transformations . . . 131

6. Incremental Model Transformations . . . 133
6.1. Finite State Machines to Petri Nets . . . 133
6.2. Model Synchronization with Synchronization Blocks . . . 136
6.2.1. Combining Bidirectionality and Change Propagation . . . 136
6.2.2. Synchronization Blocks . . . 141
6.2.3. Composition of Synchronization Blocks . . . 149
6.3. Implementation in an Internal DSL . . . 152
6.3.1. Lenses and Morphisms . . . 152
6.3.2. Isomorphisms . . . 155
6.3.3. Synchronization Modes . . . 157
6.3.4. Execution . . . 159
6.3.5. Inheritance and Superimposition . . . 162

6.4. Synchronization of Finite State Machines and Petri Nets . . . 164
6.5. Summary . . . 171

IV. Meta-metamodel Extensions to Simplify Model Analyses . . . 173

7. Using type system guarantees for model analyses . . . 175
7.1. Running Examples . . . 177
7.2. Refinements and Structural Decomposition . . . 179
7.3. Implementation in NMeta . . . 184
7.4. Code Generation . . . 187
7.5. Alternatives to Model Production Automation . . . 192
7.5.1. Sensors with an AC Power Supply . . . 192
7.5.2. Sensors without Power Supply . . . 195
7.5.3. Connections of a three phase squirrel cage motor . . . 196
7.6. Summary . . . 198

8. Simplify model analyses through Deep Modeling . . . 201
8.1. Challenges and Potentials Applying Deep Modeling for Architecture Description Languages . . . 204
8.1.1. Architecture description languages using Potency concepts . . . 204
8.1.2. Composite Components . . . 209
8.2. Deep Modeling through Structural Decomposition . . . 211
8.3. A Deep Modeling version of PCM . . . 214
8.3.1. An architecture description language using Deep Modeling . . . 214
8.3.2. Example Usage: A Media Store in DeepPCM . . . 222
8.4. Incremental Model Analyses for Deep Models . . . 225
8.5. Summary . . . 229

V. Validation and Evaluation . . . 231

9. Validation and Evaluation . . . 233
9.1. Validation Goals . . . 234
9.2. Validation Strategy . . . 235

9.3. Case Study: Incremental Queries on Railway Models 239
9.3.1. Benchmark Setup 240
9.3.2. Validation Goals 241
9.3.3. The NMF Solution 242
9.3.4. Benchmark Results 247
9.3.5. Incerator Results 253
9.3.6. Distributed Computing 261
9.3.7. Summary 265
9.4. Case Study: Incremental Data Flow Transformations 266
9.4.1. Benchmark Setup 267
9.4.2. Validation Goals 269
9.4.3. NMF Solution 270
9.4.4. Results 273
9.4.5. Summary 275
9.5. Case Study: Incremental Model Transformation of Petri Nets to State Charts 276
9.5.1. Benchmark Setup 276
9.5.2. Validation Goals 278
9.5.3. Results 279
9.5.4. Summary 282
9.6. Case Study: Incremental ATL transformations 282
9.6.1. Validation Goals 283
9.6.2. Transforming ATL to Synchronization Blocks 284
9.6.3. Limitations 296
9.6.4. Results 298
9.6.5. Summary 307
9.7. Case Study: Refactoring of Java Code 307
9.7.1. Benchmark Setup 308
9.7.2. Validation Goals 310
9.7.3. NMF Solution 310
9.7.4. Results 315
9.7.5. Summary 316
9.8. Case Study: Incremental Views in the Smart Grid Domain . 317
9.8.1. Benchmark Setup 317
9.8.2. Validation Goals 325
9.8.3. NMF Solution 326
9.8.4. Results 329
9.8.5. Summary 334

9.9. Case Study: Bidirectional Transformation from Families to Persons . . . 335
9.9.1. Benchmark Setup . . . 336
9.9.2. Validation Goals . . . 338
9.9.3. NMF Solution . . . 338
9.9.4. Results . . . 343
9.9.5. Summary . . . 350
9.10. Case Study: Modeling a Bike shop . . . 351
9.10.1. Task . . . 351
9.10.2. Case Analysis . . . 353
9.10.3. Validation Goals . . . 354
9.10.4. NMF Solution . . . 355
9.10.5. Results . . . 361
9.10.6. Summary . . . 364
9.11. Case Study: Incremental Analyses for Deep Architecture Description Models . . . 365
9.11.1. Validation Goals . . . 365
9.11.2. Benchmark Setup . . . 365
9.11.3. Results . . . 366
9.11.4. Summary . . . 368
9.12. Threats to Validity . . . 368
9.12.1. Internal Validity . . . 369
9.12.2. External Validity . . . 372
9.13. Validation Summary . . . 373

10. Experiences and Lessons Learned . . . 377
10.1. Whether incremental execution is beneficial depends on how an analysis is formulated . . . 377
10.2. A good speedup in parallel execution is not an indicator for a good speedup in incremental execution . . . 379
10.3. A good speedup in incremental execution is not an indicator for a good speedup in parallel execution . . . 381
10.4. The influence of the order . . . 382
10.5. The influence of the model and the change sequence . . . 383
10.6. Usage of Deep Modeling . . . 383
10.7. Conclusions . . . 384

VI. Epilogue . . . 385

11. Related Work . . . 387
11.1. Incremental Computation Systems . . . 387
11.1.1. General-Purpose Incremental Computation Systems . . . 387
11.1.2. Specialized Incremental Approaches . . . 390
11.1.3. Reactive Programming . . . 392
11.1.4. Scenario-specific optimization . . . 393
11.1.5. Distributed Incremental Tools . . . 393
11.2. Model Transformations . . . 393
11.2.1. Incremental Model Transformations . . . 393
11.2.2. Bidirectional Model transformation languages . . . 395
11.2.3. Lenses . . . 396
11.2.4. Model Transformation Languages as Internal Languages . . . 397
11.2.5. Higher-order Transformations . . . 398
11.3. Metamodeling Improvements . . . 399
11.3.1. Refinements . . . 399
11.3.2. Level-adjuvant languages . . . 400
11.3.3. Level-blind languages . . . 401
11.3.4. Deep Modeling compatible with EMOF . . . 401
11.3.5. Aspect-oriented modeling . . . 402

12. Limitations and Future Work . . . 403
12.1. Incremental Model Analyses . . . 403
12.2. Contraction of Dependency Graphs . . . 404
12.3. Incremental Model Transformations . . . 404
12.4. Meta-metamodel extensions and Deep Modeling . . . 405

13. Conclusion . . . 407

Bibliography . . . 409

Appendix . . . 437

Acronyms . . . 449

Glossary . . . 451

List of Figures

1.1. Goals and research questions of the dissertation 13
1.2. Contributions of this dissertation 18
1.3. Validation of Research Questions 20
1.4. A visualization of the railway network model as used in the TTC 2015 Train Benchmark case [197] 21
1.5. An excerpt of the railway metamodel used in the Train Benchmark [197] 22

2.1. The type system of NMeta (simplified) 33
2.2. The expression tree for the lambda expression to obtain whether the entry signal of a route shows *GO*. 36
2.3. An example e-commerce system in PCM 41
2.4. System assembly in PCM (simplified excerpt) 42
2.5. System deployment in PCM (simplified excerpt) 43

3.1. The metamodel excerpt of the railway network metamodel used in Example 18 60

4.1. The DDG and nodes that must be reevaluated when changing the signal to *FAILURE* 93
4.2. The DDG and nodes that are disconnected if the entry semaphore is changed to `null`. 97
4.3. Workflow of a distributed incremental analysis offloading incremental analysis to a Microsoft Orleans cluster 103
4.4. An example stream processor chain 105
4.5. Dispatching collection items between node grains 106

5.1. The contracted DDG and effects of changing the signal to *FAILURE* 119
5.2. The contracted DDG if the entry semaphore is not contained in a route and effects of changing the signal to *FAILURE* 122

5.3. Finding pareto-optimal incrementalization configurations using genetic search algorithms 125

6.1. The metamodels of finite state machines and Petri nets 134
6.2. Illustration of the considered example transformation from finite state machines to Petri nets exemplified for a simulation lifecycle. 135
6.3. A simple metamodel of men and women 139
6.4. An example instance of men and woman at state ω_0 139
6.5. An example of men and women at state ω_1 140
6.6. Schematic overview of synchronization blocks 142
6.7. Synchronization of the states of a finite state machine with the places of a Petri net 143
6.8. Schematic overview of unidirectional synchronization blocks 148
6.9. Synchronization of the transitions of a finite state machine with the transitions of a Petri net 167
6.10. Synchronization of the end states of a finite state machine with swallowing transitions of a Petri net 169
6.11. Synchronization of the names of a finite state machine and a Petri net 169

7.1. Sensor example 178
7.2. Star and delta connections in motors 179
7.3. NMeta meta-metamodel with extensions to support structural decompositions and refinements. 185
7.4. Generated model representation class and interface for references 188
7.5. Modeling that an `ACSensor` requires an `AC` power supply in NMeta. The inserted refinement is printed in blue. 195
7.6. Modeling that a `PassiveSensor` must not have a power supply in NMeta. The inserted reference constraint is printed in blue. 196
7.7. Star and delta connections in motors modeled in NMeta. The inserted structural decomposition is printed in blue. 197

8.1. Levels of DeepADL 205
8.2. Modeling architectures in level-adjuvant languages 207

8.3. Nested composite components that imply an arbitrarily deep level structure . . . 210
8.4. Poodle is both a class and an instance . . . 211
8.5. Component types in DeepPCM . . . 216
8.6. System deployment in DeepPCM . . . 220
8.7. Composite components in DeepPCM . . . 221
8.8. Excerpt of the implied DeepPCM metamodel to specify the architecture of a MediaStore system based on a repository of component types . . . 223
8.9. Deployment of the MediaStore in DeepPCM . . . 225

9.1. Phases of the Train Benchmark [197] . . . 241
9.2. Performance results for revalidation . . . 248
9.2. Performance results for revalidation(cont.). . . . 249
9.3. Rete network created by EMF-IncQuery for the *SwitchSet* query of the Train Benchmark . . . 250
9.4. Performance Results for batch validation . . . 251
9.4. Performance Results for batch validation (cont.) . . . 252
9.5. Working sets against relative model size . . . 254
9.5. Working sets against relative model size (cont.) . . . 255
9.6. Design space of different configurations for different validation queries of the Train Benchmark run . . . 257
9.7. Design space of different configurations for the *SemaphoreNeighbor* query of the Train Benchmark run . . . 258
9.8. Histogram of the revalidation times for the *SemaphoreNeighbor* query of the Train Benchmark run . . . 259
9.9. Ranks of average revalidation times . . . 260
9.10. Ranks of average revalidation times for the *SemaphoreNeighbor* query . . . 261
9.11. Performance results for the Train Benchmark in incremental setting . . . 264
9.12. Primitives of FlowM2M (simplified) . . . 268
9.14. The dynamic dependency graph template for the function in listing 9.9 applied to a member "Jane Jones" and the propagation of changes to her last name. . . . 271
9.15. Performance results for the NMF solution of the TTC 2016 Live Contest . . . 275
9.16. Performance results for the initial transformation . . . 280

9.17. Performance results running 20 randomly generated changes . . . 280
9.18. Synchronization block from Listing 9.13 . . . 289
9.19. Response times of a generated change sequence in the *Families2Persons* example . . . 301
9.20. Response times of a generated change sequence in the *Make2Ant* example . . . 303
9.21. Response times of a generated change sequence in the *Make2Ant* transformation . . . 304
9.22. Response times of a generated change sequence in the *Class2Relational* transformation . . . 306
9.23. Sketch of the Transformation Chain for the Java Refactoring case at the TTC 2015 [131] . . . 308
9.24. The Program Graph (PG) metamodel cf. [131] . . . 309
9.25. Excerpt from the COSEM metamodel relevant for task 1 . . . 319
9.26. Excerpt from the CIM metamodel relevant for task 1 . . . 320
9.27. The Viewtype for the Outage Detection Task . . . 322
9.28. An excerpt from the CIM metamodel relevant for Task 2 . . . 323
9.29. An excerpt from the Substation standard relevant for Task 2 . . . 324
9.30. The Voltage Three-Phase Measurement Matrix . . . 325
9.31. The join in the outage detection task formulated in a synchronization block . . . 327
9.32. Time to propagate change sequences of 20 elementary changes in the *Outage Detection* task . . . 332
9.33. Time to propagate change sequences of 20 elementary changes in the *Outage Prevention* task . . . 333
9.34. Working set sizes solving the Smart Grid benchmark . . . 334
9.35. Source and target metamodel in the FamiliesToPersons case study [7] . . . 336
9.36. Synchronization block to synchronize family members with person elements . . . 339
9.37. Synchronization block to synchronize names . . . 339
9.38. Performance results for the scalability tests in the *FamiliesToPersons* case study . . . 344
9.39. Performance results for the scalability tests in the *FamiliesToPersons* case study (cont.) . . . 345
9.40. Performance results for the scalability incremental forward tests in the *FamiliesToPersons* case study . . . 346

9.41. Performance results for the scalability incremental backward tests in the *FamiliestoPersons* case study 346
9.42. Configurations and components in a bike shop 355
9.43. Bicycles in an NMeta metamodel as specializations of components 356
9.44. Stacked refinements for MountainBikes 358
9.45. Inheritance hierarchy of bicycle types 359
9.46. Level 2: Bicycle in the MultEcore solution of the MULTI 2017 modeling challenge 362
9.47. Level 3: Racing Bicycle in the MultEcore solution of the MULTI 2017 modeling challenge 363
9.48. Initial analysis time to check the correct allocation of assemblies 367
9.49. Median response times for updates of the allocation model to updated analysis results 367

10.1. Benchmark results for parallel execution of the original *SemaphoreNeighbor* query 380

List of Tables

9.1. Summary of the case studies presented in this thesis 239
9.2. Results from the open peer review of the TTC 2015 Train Benchmark 247
9.3. Complexity of the data flow transformation solutions, measured in lines of code 273
9.4. Example ATL-OCL operations and their mapping to the corresponding C# expressions 293
9.5. Open Peer Review Results for the TTC 2015 Java Refactoring Case 315
9.6. Responses from the open peer reviews and from the live evaluation at the TTC 2017 for the *FamiliesToPersons* case study 347

Listings

1.1. Query for inaccurate switch positions 23

2.1. Lambda Expressions in C# 36
2.2. Query to find inaccurate switch positions in a collection of routes (as treated by the compiler) 37
2.3. Query to find inaccurate switch positions in a collection of routes (method chaining syntax) 38

5.1. The example predicate to check whether a routes entry semaphore shows GO 118

6.1. Implementation of a FirstOrDefault lens for arrays 155
6.2. Implementation of a FirstOrDefault lens for lists 156
6.3. A model synchronization in NMF Synchronizations 164
6.5. The DeclareSynchronization method of AutomataToNet 165
6.4. Definition that states and places should correspond based on their names 165
6.6. Matching transitions 168
6.7. One way synchronizations 169

8.1. The generated interface for an `AssemblyContext` 218
8.2. A model analysis whether assemblies are deployed to the same or connected resource containers in DeepPCM 226
8.3. An analysis whether components are correctly allocated 228
8.4. Alternative analysis whether assemblies are deployed to the same or connected resource containers in DeepPCM 229

9.1. .NET Modeling Framework (NMF) implementation of the *PosLength* query 243
9.2. NMF implementation of the *SwitchSensor* query 243
9.3. NMF implementation of the *SwitchSet* query 244

9.4. NMF implementation of the *RouteSensor* query 244
9.5. NMF implementation of the *SemaphoreNeighbor* query 245
9.6. A simplified implementation of the `Fix` function 246
9.7. The solution to the *PosLength* query adapted for distributed computing 262
9.8. The solution to the *SemaphoreNeighbor* query adapted for distributed computing 265
9.9. An evaluate node in the incremental NMF solution to compute the full name of a person 270
9.10. ATL matched rule without bindings 285
9.11. NMF Synchronizations rule without synchronization blocks 285
9.12. An exemplary ATL binding 288
9.13. The synchronization block generated for the binding in Listing 9.12 288
9.14. Definition of a simple ATL functional helper 290
9.15. C# version of the simple ATL helper from Listing 9.14 with attribute and proxy 291
9.16. Collection initialization in C# 293
9.17. ATL matched rule with a variable 294
9.18. Synchronization of classes in JaMoPP and the PG metamodel 311
9.19. The synchronization rule for method definitions 313
9.20. The implementation of *Pull Up Method* 314
9.21. Task 1 realized in ModelJoin 321
9.22. The implementation of the main rule for outage the outage detection task 327
9.23. Implementation of kept attributes and references in the outage detection task 328
9.24. The implementation of the main rule in the outage prevention task 330
9.25. Transforming power system resources 330
9.26. Implementation of main synchronization blocks 340
9.27. Implementation of the `GetFullName` lens 340
9.28. Implementation of the `FamilyMemberCollection` 341
9.29. The `MemberToMale`-rule 342
9.30. Repairing an inconsistent names in EVL+Strace 348
9.31. Creating instances using the generated model API 360
9.32. Querying the average sales price for race bikes 360

9.33. Querying the average sales price for race bikes incrementally 361

10.1. The original NMF solution of the *SemaphoreNeighbor* query in the TTC Train Benchmark 2015. 378
10.2. The original NMF solution of the *SemaphoreNeighbor* query in the TTC Train Benchmark 2015 run in parallel. 379

Part I.

Prologue

1. Introduction

The goal of this chapter is to motivate and explain the purpose and structure of this dissertation. Section 1.1 introduces the motivation on a high level. Section 1.2 discusses the working hypotheses under which the dissertation is created. Section 1.3 lists the research questions tackled. Section 1.4 lists the core contributions made in this thesis. Section 1.5 gives an overview how these contributions are validated. Section 1.6 demonstrates the contributions of this thesis in an example taken from the railway domain. Section 1.7 explains the structure of the remaining document.

1.1. Motivation

A common goal in many engineering disciplines is to create abstract models of a system in order to reason on properties of the modeled system by analyzing the model. Nowadays, many of these analyses are supported by software. As the systems evolve, the models are changed and the analyses may have to be recomputed. These analyses include simple validations, but also more complex ones such as simulations.

For many analyses, even little changes have a dramatic effect so that keeping prior intermediate results does not yield any benefits. For others, surprisingly large benefits can be drawn from incremental execution. An early example of the latter is digital circuit simulation. There, incremental simulation yields orders of magnitude in performance [45, 178] by introducing buffers to save some intermediate results to speed up response times from a model change to updated analysis results. In many application areas these response times to get updated analysis results for a given model change are critical. Examples include the area of self-adaptive systems where it is important to reconfigure the system as fast as possible before it crashes or breaks service level agreements [44, 55]. Hence, this response time is the most common

measurement for evaluating approaches in the area of self-adaptive systems [198].

Saving such intermediate results for future requests on changed input models and their invalidation is called incrementalization. The goal of this process is that ideally only those *parts of an analysis* have to be recomputed that are affected by a change of *parts of the underlying models*. This may lead to an improved performance if the efforts to invalidate those analysis parts are smaller than rerunning the analysis from scratch. Similar to parallelization and distributed computing, incrementalization can be thought of as a technique to improve the scalability of an analysis[1]. However, while the possible speedup that can be gained from parallelism and distributed computing is bound by the used resources[2], incrementalization in theory offers speedups dependent on the size of the analyzed data: Ideally, the response times to adjust the result of an analysis is dependent only on the size of a model change rather than on the size of the entire model[3]. On the other hand, saving intermediate results costs memory and time to traverse this memory or keep these intermediate results up to date. Therefore, the speedup that can be achieved for practical problems is unclear.

The baseline for this comparison is a batch implementation where only the analysis result is stored and invalidated for every model change. Analysis and underlying models are not divided in this batch incrementalization. Therefore, the mapping which parts of the analysis need to update for which parts of the model is trivial. In addition to the fact that the entire model has to be reevaluated for every model change, this approach also has the disadvantage that complex analysis results are complex to compare since the analysis will usually create entirely new result objects. However, if otherwise small changes of the underlying models may cause to reevaluate large parts of the analysis in a butterfly-effect, the batch incrementalization may be the

[1] Another technique to improve the performance of an analysis is forecasting, where some calculations are made in advance to accelerate future analysis runs. However, there are only few problems where forecasting solutions exist. Thus, forecasting is not in the focus of this thesis.

[2] Amdahl's law [6] provides a more precise upper bound for the speedup that can be achieved with parallelism but it in some cases it is not even obvious how this speedup can be achieved.

[3] Giese and Wagner even define incremental model transformation engines through their performance characteristics: A model transformation is fully incremental if the response time for a model change only depends on the change size and effectively incremental, if the response time depends strictly less than linear on the overall model size [74].

best choice. If such changes exist but occur rather seldom, risking them may produce better results in the average case. Therefore, it is also not clear how to incrementalize a given analysis.

Even if the incrementalization strategy is clear, manual incrementalization is a non-trivial and error-prone task. One has to identify a suitable partition how analysis and underlying models can be divided. Upon changes of underlying model parts, the right intermediate results of the analysis (parts) have to be invalidated. This opens the gates for bugs since it is very easy to forget cases in which intermediate results need to be invalidated[4]. In any case, a manual incrementalization is quite costly to develop. Furthermore, the management of intermediate results may conceal the analysis code and degrade understandability. As a possible consequence, domain experts may no longer be able to proofread the code. This may lead to undetected bugs in the analysis and hence wrong analysis results implying wrong conclusions on the real system. Besides correctness, the understandability is crucially important. Currently, understanding existing code makes up almost half of software maintenance costs [22]. Maintenance in turn is the main driver for overall project costs [193].

A promising approach to tackle these problems is implicit incrementality. An implicit incrementalization system decides based on a batch specification of the analysis which intermediate results should be saved and manages their invalidation, typically by tracking its dependencies. Such systems exist either for general-purpose languages capable of expressing any analysis [43] or for specific classes of analyses such as incremental queries [217, 25], incremental pattern matching [26] or even incremental model transformations [74, 73]. These specialized incremental approaches limit their applicability to a given class of analyses and use abstractions common to these analyses to make incremental execution more efficient. Meanwhile, existing general-purpose approaches are limited in their applicability as there is currently no approach that can handle Essential Meta Object Facility (EMOF) models.

[4] Several authors [118, 60, 18] have claimed that a third of the code base of Adobe desktop applications concern event handlers to user actions. These event handlers would make 50% of the defects. Edwards even denotes this as a *Callback Hell*. However, while I agree on the difficulty of event handler management, these authors all seem to draw from a keynote talk by Sean Parent at the OOPSLA 2006, which is no longer online accessible.

As of today, most existing incrementalization systems divide analyses using the instruction set of the programming language in which the analysis is specified and divide the underlying models in their individual model elements[5]. When the analysis is run initially, the system tracks the execution in a graph data structure called Dynamic Dependency Graph (DDG) and uses this graph to propagate changes. The expressiveness of such an approach, but also the size of a DDG and hence the memory footprint of such an approach is determined very much by the level of abstraction offered by the programming language. Because the incrementalization operators are limited to the instruction set, this technique is referred to in the remainder of the thesis as instruction-level[6].

Incrementalization systems exist both for Turing-complete programming languages such as OCaml, Rust and Python [43, 84] but also for specialized languages such as IncQuery [26] for graph pattern matching. While the instructions of the former are on a very low level such as adding two numbers or a conditional expression, the primitives of a language for graph pattern matching are on a much higher abstraction level such as joining two partial pattern matches. This means that much less primitives are required for an analysis and therefore a much smaller DDG suffices. Here the problem is rather that they are restricted to certain classes of analysis as the underlying formalisms are not Turing-complete. This is problematic especially when analyses have to be modified such that they fall out of this scope.

In a batch specification of model analyses, these problems are mitigated by internal composition, i.e. developers may use libraries and frameworks that raise the level of abstraction to specify analyses. As an example, math libraries such as the Intel MKL[7] lifts the level of abstraction when working with math operators. At the same time, it is massively optimized for performance. In a similar way, it would be desirable to give library or framework developers a tool at hand such that they tune their libraries and frameworks for incremental execution such that they can get a comparable efficiency like specialized incrementalization systems without losing the general-purpose applicability of a Turing-complete functional programming language.

[5] Some approaches such as Adapton or Self-adjusting computation are originally defined on immutable data and are not aware of models at all but these approaches have a granularity similar to individual model elements.

[6] An exception to this are Traceable Data Types [3] which are discussed in Section 11.

[7] `https://software.intel.com/en-us/mkl`, retrieved 06 Sep 2017

Not only the expressiveness of the language used for model analyses is important. Meyerovich and others have shown that programmers do not appreciate to change their primary programming language [149], thus creating a new language causes a language adoption problem. Furthermore, Mohaghegi and others found that in model-driven engineering, an important factor for the adoption of these techniques in industry is the availability of tools [152] but multiple studies reported that this tool support is lacking [188, 153]. On the contrary, modern mainstream general-purpose programming languages are already on a high level of abstraction. Akehurst and others even state that C# makes the Object Constraint Language (OCL) redundant [5]. According to many indices on the popularity of programming languages such as the TIOBE index[8] or the IEEE Spectrum[9], C# is one of the most widespread programming languages in the world. Therefore, it is appealing to use this language as a basis for incrementalization, in particular the purely functional part of C# that is used by Akehurst and others in their discussion.

However, even though functional programming is theoretically capable to express all kinds of model analyses or other artifacts built on top of them, general-purpose languages are not always an appropriate tool. An important example of such model analyses are model analyses implemented as model transformations [184][10]. Specialized incrementalization systems for model transformations in the form of incremental model transformation languages exist, based on Triple Graph Grammars (TGGs) [75, 72] and textual transformation languages [122], but it is unclear whether general-purpose incrementalization systems can be extended in this direction. Even though model transformations have some characteristics that make them hard to specify in general-purpose languages, other parts such as helper functions or predicates are often specified in functional expression languages and thus their incrementalization yield the same problems. Therefore, reusing general-purpose incrementalization systems for these kinds of model analyses, in particular model transformations, would be highly appreciated.

Model transformations are also very important for any other model analyses as they enable external composition: An analysis may consist of a

[8] `https://www.tiobe.com/tiobe-index/`, retrieved 06 Sep 2017

[9] `https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages`, retrieved 06 Sep 2017

[10] This statement is almost a matter of course in the model-driven community but is hardly supported by empirical studies.

transformation of the input model to a different metamodel and a chained analysis on that metamodel. For example, to analyze whether a system is free of deadlocks, one often transforms the model into Petri nets and analyzes this property there. However, an incrementalization of such a composed analysis always requires an incrementalization of the model transformation to Petri nets. Otherwise, the analysis would always have to operate on an entirely new input model which makes it hard to identify what has actually changed.

In order to avoid the language adoption problem and the tool availability problem for model analyses, the thesis in particular uses the approach to use an internal Domain Specific Language (DSL) [67] for model transformation. This approach has been followed by multiple tools in the past [50, 70, 92, 110, 129] for various different reasons, but none of these languages supports incremental change propagation.

An implicit incrementalization system can only restructure the analysis in the boundaries of the underlying metamodel. However, some inefficiencies and even some analysis needs may come from lacking capabilities of the used meta-metamodels: The probably most common meta-metamodel is Ecore, an implementation of the EMOF standard that only is a subset of the Meta Object Facility (MOF) standard. In particular, Ecore does not support redefinitions. The usage of redefinitions could simplify metamodels in such a way that certain classes of analyses such as several constraint validation checks are no longer required as can be guaranteed by the target platform type system.

Furthermore, several metamodels, in particular architecture-description languages, contain a high accidental complexity caused by the inapplicability of strict metamodeling techniques to clearly represent instance-of relations between different model elements [135]. This leads to more complex analyses that could be simplified significantly if the instantiation relation was reflected in the metamodel.

Approaches that overcome this limitation are usually referred to as Deep Modeling or Multi-level modeling approaches. However, these approaches require entirely new tools [13]. To avoid a necessary adaptation of implicit incrementalization techniques, it is desirable to have an approach for Deep Modeling that is less intrusive as an extension to existing metamodeling frameworks.

In summary, this dissertation follows two main goals:

1. Reduce the response time from changes in a model to updated analysis or transformation results and
2. minimize the efforts for the developers of these artifacts to enable change propagation.

The contributions of this dissertation are implemented for C# or an internal DSL in C# in order to reduce the language adoption problem or the tool availability problem.

1.2. Assumptions and Definitions (A)

Before we discuss how the goals of this dissertation are broken down to research questions, this section introduces assumptions and definitions that this dissertation is based upon.

A1: A model analysis is a side-effect free computation that takes one or multiple models as input and returns the analysis results in the form of a model, a primitive value or a (possibly ordered) collection of any of the former. This assumption defines the space of programs affected by the term model analysis to clarify the scope of the dissertation. A primitive value in this context can be a number, a date, an enumeration item or a string. While the restriction on a particular result format is only a syntactical restriction, the demand that the analysis is side-effect free is a hard limitation. The reason we require the absence of side-effects is that for the behavior of the overall program it must not matter which parts of an analysis are reevaluated as long as the correct result can be obtained.

Because model analyses usually generate new insights on a model, they are usually not injective and therefore not invertible. In particular, it is impossible to reconstruct a model based on a given analysis result.

A2: Existing programming languages are well-suited to develop model analyses in batch manner. We believe that existing mainstream programming languages are well-suited to specify model analyses in a batch manner. This specification may either be based on an imperative or functional programming paradigm.

A3: Manually implementing a model analysis incrementally is non-trivial, error-prone and conceals the intention of the model analysis. Manual incrementalization of a model analysis requires the developer to think about which intermediate results are worth saving and demands the invalidation of these intermediate results. Furthermore, the developer has to find the right granularity in which the analysis is incrementalized. The incrementalization can either be done together with the model modification or based on events. In any case the choice how an analysis can be incrementalized is non-trivial and the implementation is error-prone.

A4: A model transformation is a program that takes one or multiple models as input and returns a new (transformed) model as well as a trace that defines the correspondence between parts of the input model and parts of the output model or collections thereof A model transformation maps elements of a source model to elements of the target model. This implicitly defines a correspondence relation between these elements referred to as trace. Notably, we existing classifications of model transformation approaches see traceability as optional [53], but we think that this feature is central for model transformation approaches. In particular, we think that just referring to the input and the output of a program as models is not enough to qualify for the term model transformation.

A model transformation may be invertible and may even be bidirectional, a feature often required for model synchronization purposes [53].

A5: General-purpose programming languages are not suitable to specify model transformations. This assumption was originally introduced by Sendall and Kozaczynski in 2003 [184]. In the model-driven community, this assumption is almost a matter of course. The reasons for this is mostly that model transformations in general-purpose programming languages require a lot of book-keeping and manually specified pattern matching. However, this

statement is hardly empirically validated and therefore treated as an assumotion in this thesis. In particular, we are not aware of an existing empirical study that shows that a dedicated model transformation language is better suited for model transformation tasks than a general-purpose programming language. We designed an empirical experiment template to create empirical evidence [127], but this experiment has not yet been carried out due to time constraints.

However, a multitude of approaches [50, 221, 111, 92, 129] shows that model transformation languages can be implemented in internal languages [67]. An investigation on how to reuse modularity concepts and tool support using internal model transformation languages can be found in prior work [101].

A6: A model synchronization is an uni- or bidirectional model transformation that operates on existing models rather than creating fresh ones. Usually, each execution of a model transformation produces a new target model. In the presence of heterogeneous models where the information contained in an existing target model cannot fully be reconstructed by the source model (and vice versa), creating a fresh target model is not desirable as it would lead to information loss. In such a scenario, it is often desirable to reuse existing target models where possible and fit their structure such that it is consistent with the source model with regard to the model transformation. In case of a bidirectional model transformation, this repair of inconsistencies may also be performed in the opposite direction, i.e. that the source model is changed to obtain a consistent state. [11]

Because model transformations, especially model synchronizations, need to perform side-effects that are forbidden in model analyses and the differences in language suitability, we assume that there is a significant difference between model analyses and transformations such that techniques for incremental model analyses cannot immediately be applied for model transformations.

However, there are interesting commonalities between model transformations and model analyses: In a typical model transformation, one is often

[11] Model synchronization is also often referred to as consistency preservation. A very good overview on the challenges and possible solutions to this problem have been provided by Kramer in his PhD thesis [128].

interested which model elements need to be transformed, often depending on other model elements. This analysis is necessary for the execution of the model transformation. Therefore, these analyses are an integral part of a model transformation.

A7: A model analysis and a model transformation can be composed internally or externally. The usage of libraries for often used functionality is a very important feature of most modern programming languages. Developers may save time and reduce bugs by reusing already implemented functionality. A common example of reused functionality in the context of model analyses are query libraries (cf. Section 1.6). For model transformations, internal composition is still a subject of research [218], but also a desirable goal.

As external composition, we regard the reuse of existing model analyses or transformations as a black box through chaining. With this technique, an existing analysis or transformation based on a different metamodel can be reused by first transforming the input model to this metamodel and then processing the transformed model further.

1.3. Research Questions (RQ)

In this section, the research questions tackled by this thesis are explained. All these research questions contribute to the three general goals identified in Section 1.1.

For an overview, the goals and research questions are depicted in Figure 1.1. RQ I–RQ III contribute to the goal of an increased performance at lowest cost for developers. RQ IV reviews how the response times can be reduced by utilizing the target platform type system.

In the following, the research questions are described in more details:

RQ I: How to incrementalize methods on a high abstraction level? Existing incrementalization approaches operate at the instruction level of their respective programming languages. This inevitably means that any incrementalization of a method is tied to its batch specification. This is not necessarily optimal since the batch implementation may implicitly include assumptions

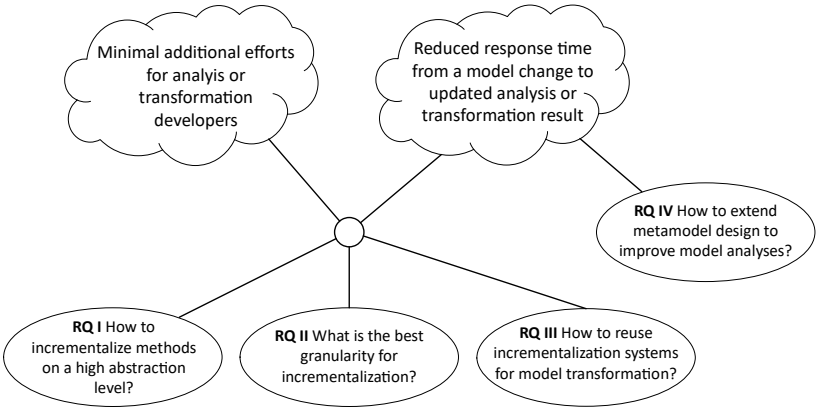


Figure 1.1.: Goals and research questions of the dissertation. The goals are printed in clouds, the research questions are depicted in ellipses.

that are not necessary for the semantics of the method. For example, the batch implementation of a method may iterate a collection in sequence, though the order of the elements is not relevant. Here, a way to describe the (optimized) incrementalization of a method would be a clear advantage.

This research question can be decomposed into the following questions:

RQ I.1: How to design an interface for extensions to support incrementality? Similar to traditional batch analysis frameworks being tuned for performance, it is desirable to enable developers of frameworks to tune their frameworks for incremental execution. For some analyses like the connectivity in a graph, algorithmic solutions for their incremental behavior exist. As these analyses may be specified by third-party component vendors, there must be an interface to express the incremental behavior of analyses.

RQ I.2: How to ensure the correctness of incremental execution in the presence of user extensions? If an incrementalization system is extended by explicit incrementalizations provided by third parties, it is unclear how to ensure that the resulting incrementalized analysis is still correct.

RQ I.3: How does the performance of a general-purpose incrementalization systems with manual incrementalization relate to specialized incrementalization frameworks? The application of incrementality is tied to the goal of a performance improvement. If no performance improvement can be gained, there is no point having an incrementalized model analysis. For the implied importance of performance, it is important that any method call incrementalizations can achieve a performance comparable to specialized incrementalization frameworks such as IncQuery [26] for incremental pattern matching.

Another way to think of this research question is how we enable incrementalization in the presence of internal composition of model analyses and use this internal composition to improve the performance of the resulting incremental model analysis.

RQ II: What is the best granularity for incrementalization? Depending on the complexity of the analysis[12], incrementalization may lead to large DDGs that consume a lot of memory. It is therefore desirable to partition the model in a more coarse-granular way than into individual model elements and update a larger part of the analysis when an elementary model change happens inside such a model partition.

This research question can again be decomposed into the following research questions:

RQ II.1: How to find suitable model partitions? As of today, incrementalization approaches mostly operate on individual model elements. This enables incrementalization systems to only reevaluate those parts of a model analysis that are affected by a given elementary change. However, due to the memory overhead, the time taken to traverse and update the DDG may outweigh the any benefits drawn from the reduced computational complexity arising from reusing prior results. Therefore, it can be useful to group multiple model elements in model partitions and consider elementary model changes in this partition rather than in an individual model element. However, this raises the question how this can be done.

[12] Possibly as part of a model transformation

RQ II.2: How to find suitable model analysis partitions? Similar to RQ II.1, it is also unclear how the model analysis should be partitioned. Ideally, this partition minimizes the cross-connection accesses between model partitions.

RQ II.3: How to incrementalize cross-partition references within a model analysis partition? If suitable partitions are found both for the model and the analysis, this raises the question when to reevaluate this part of the analysis. If the analysis partition is associated with a model partition, elementary model changes within that model partition will likely lead to reevaluate the analysis. However, if there is an elementary model change that references a different model partition (such that the referenced other model partition may change), it is unclear how to react on such a change.

RQ II.4: How to automate the choice of strategies for analysis partitions? Given that there are multiple possibilities for a model partition and likewise for an analysis partition, this implies an additional overhead for the developer of a model analysis to actually make this choice. In order to take away this new burden from the analysis developer, it would be appreciated to automate this design process and obtain the best model and analysis partition automatically where best may be relative to one or multiple quality dimensions such as response time or memory consumption.

RQ III: How to reuse incrementalization systems for model transformation? Model-transformations play an important role in model-driven engineering. Sometimes, they are even called its "heart-and-soul" [184]. Therefore, an incremental execution to reduce the response time from a model change to an updated target model is also desirable for model transformations. Such an incremental transformation is also a foundation for reusing existing incremental model analyses specified for other metamodels.

This research question is decomposed as follows:

RQ III.1: How to incrementalize unidirectional model transformations? Many model transformations are unidirectional in the sense that they transform models from a source metamodel to a target metamodel. An incremental execution of such a model transformation simply has to monitor the

source metamodel for any changes and propagate these changes to the target model.

RQ III.2: How to execute bijective bidirectional model transformations incrementally? Bijective model transformations can be executed either from a source to a target metamodel or vice versa. In both cases, the transformation target can be entirely recomputed from the source model, but a difficulty of such model transformations is typically how to express both directions of the transformations at once, i.e. with minimal overhead to the transformation developer.

RQ III.3: How to execute heterogeneous model synchronizations incrementally? Lastly, there are also model transformations that can be executed in both directions but neither the target model can be generated from the source model nor vice versa. Rather, there is some semantic overlap between the metamodels that has to be kept consistent in instances of the involved metamodels [128]. An important criterion for model synchronization is that the synchronization is hippocratic, which means that an change in one of the models that does not break the consistency criteria also does not cause any propagation in the other model.

In an analogy to RQ I, this research question can be thought of how to enable incrementalization in the context of external composition of model analyses and model transformations.

RQ IV: How to extend metamodel design to improve model analyses? So far, most languages for MDE are applicable for classical two-level modeling conforming to the EMOF standard. However, it is not clear whether metamodeling using this standard yields the best metamodels with respect to incremental model analysis, or whether metamodel extensions could lead to better metamodels. Here, especially the ideas of Deep Modeling have claimed to improve metamodels by avoiding accidental complexity but also the modeling options defined in the Complete MOF (CMOF) standard are worth an investigation.

This research question again can be decomposed into the following research questions:

RQ IV.1: How can a meta-metamodel be extended to better make use of type system guarantees? CMOF defines a concept of association redefinitions that is however not available in the EMOF subset and not implemented in common meta-metamodels such as Ecore. However, an implementation of redefinition or refinements compatible to platform type systems could use guarantees of the target platform to simplify model analyses.

RQ IV.2: How can Deep Modeling be achieved while reusing the tools for traditional strict two-level metamodeling? This research question mainly seeks a formalism how Deep Modeling can be described that does not break with existing tool support. In particular, the rationale behind this question is to avoid a need for a second incrementalization system, but to reuse an incrementalization system for traditional two-level modeling approaches also for Deep Modeling.

RQ IV.3: How to adapt an incremental computation system for Deep Modeling techniques? This research question deals with the performance of model analyses in an incremental computation system and how this system needs to be adapted for such a purpose.

1.4. Contribution Summary (C)

The thesis presents an automated approach to incrementalize functional code with a high level of abstraction and optimization based on the containment hierarchy. This incrementalization system is further applied to model transformations. Furthermore, the thesis investigates how and to what degree the efficiency of analyses can be improved by using more expressive modeling foundations, particularly enabling Deep Modeling. These contributions are validated and evaluated by implementations, case studies and empirical experiments.

An overview of the contributions made in this dissertation is depicted in Figure 1.2. Each contribution consists of a formalization and an implementation integrated to .NET Modeling Framework (NMF). In the remainder of this section, the contributions and their main results are described in a bit more detail.

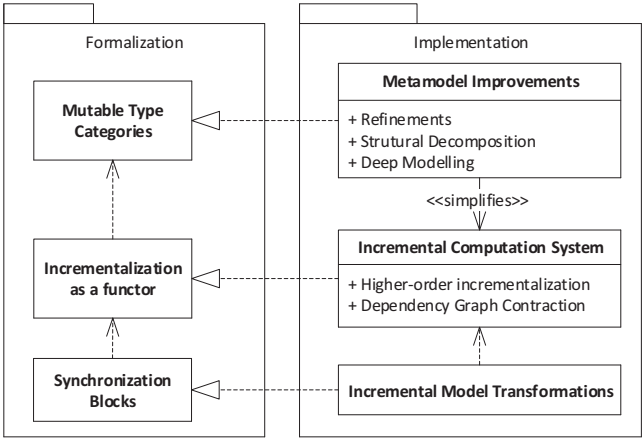


Figure 1.2.: Contributions of this dissertation

C I: Improvements of Incremental Computation The thesis improves the implicit incrementalization process in several ways:

C I.1: Integration of Dynamic Algorithms We formally represent the process of incrementalization as a functor from category theory. While the use of category theory for incrementalization has been suggested before [42], this happened rather to simplify the implementation of incremental computation systems. Chapter 4 proposes an approach to use this insight for integrating custom dynamic algorithms into incrementalization systems. This gives developers of analysis frameworks an opportunity to tune their framework for incremental execution. With the help of the formalization, the thesis proves an important correctness theorem which enables to guarantee the correctness of incremental computation if all elementary instructions are correctly incrementalized. Therefore, this contribution tackles RQ I.

C I.2: Incrementalization Strategies utilizing Generalized Model Changes along Containment Hierarchies Chapter 5 proposes a set of four different incrementalization strategies to realize the incremental derivation of model analyses. These different strategies span a design-space how the incrementalization of a given analysis can be configured. By automatically exploring this

design-space, the incrementalization process can be optimized with regard to the response times from a change to an updated result in a given example model. This contribution tackles RQ II.

C II: Synchronization Blocks To reuse an incrementalization system for incremental, bidirectional model transformations, Chapter 6 proposes the approach of synchronization blocks. We prove that using this new algebraic construct, inconsistencies can be fixed incrementally and bidirectionally and the synchronization operator is hippocratic[13]. Synchronization blocks give an answer to RQ III.

C III: Meta-metamodel extensions The thesis proposes several meta-metamodel extensions to enable the design of better metamodels with respect to the performance of analyses based on them, tackling RQ IV:

C III.1: Refinements and Structural Decomposition Chapter 7 proposes two extensions to existing two-level modeling approaches: Refinements and Structural Decomposition. With the help of these two extensions, metamodels can be simplified such that several analyses become obsolete as they are guaranteed by the underlying type system.

C III.2: Non-intrusive Deep Modeling Chapter 8 proposes an approach to make Deep Modeling techniques available only through structural decomposition of references and attributes. This approach does not only have advantages in the reuse of existing tool support, it also circumvents some restrictions of existing Deep Modeling approaches. In particular, the approach is able to express metamodels with a composite instantiation patterns which level-adjuvant Deep Modeling techniques cannot.

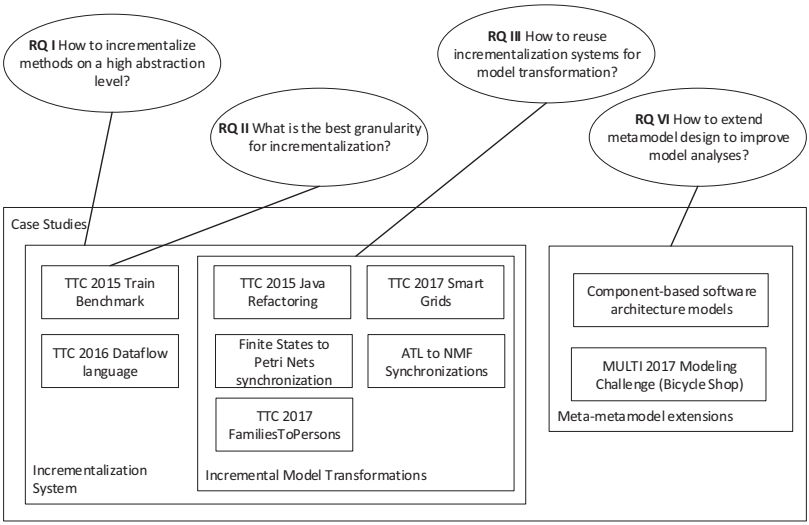


Figure 1.3.: Validation of Research Questions

1.5. Validation Overview

An overview of the validation conducted in this thesis is depicted in Figure 1.3. The validation for incremental model analyses (C I) and incremental model transformations (C II) is done through multiple case studies. Because several insights from the contributions come in the form of correctness proofs, the validation concentrates on the applicability of the proposed formalisms and the speedup that can be achieved in the various systems. Overall, this part of the validation consists of seven case studies. However, the sizes and the complexity of these case studies varies. Some of the case studies are taken from multiple editions of the Transformation Tool Contest (TTC), often inspired from realistic projects. Therefore, multiple solutions of other tools are available which leads to the possibility to compare the contributions of this thesis with other tools.

[13] Hippocraticness of a synchronization operator means that if source and target model are already consistent, the synchronization operator does not change either source or target model.

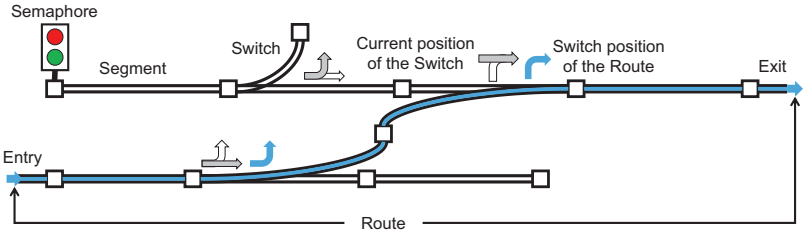


Figure 1.4.: A visualization of the railway network model as used in the TTC 2015 Train Benchmark case [197]

Because all of the case study solutions use the incrementalization system of C I.1, insights regarding RQ I can be drawn from all of these case studies. Similarly, all case study solutions that involve model transformations can be used to draw insights to RQ III. C I.2 requires heterogeneous change sequences to be applicable which were not available for the selected case studies. We therefore extended the Train Benchmark in this direction, but this means that this is the only case study that we use to validate RQ II.

Contribution C III is validated using case studies from the MULTI 2017 modeling challenge and a Deep Modeling version of the Palladio Component Model (PCM). In addition, Section 7.5 discusses the concepts of contribution C III.1 using examples from the domain of industrial production automation.

1.6. Running Example

Throughout large parts of this dissertation, a synthetic example analysis is used to explain issues and solutions to the problems dealt with in the thesis. The example is taken from the TTC Train Benchmark [197]. Though only a synthetic benchmark, this example case demonstrates both practical use cases and also many of the problems attached to incremental computation.

One of the tasks in this benchmark is to select the switches along routes in a railway net that are set incorrectly according to signal positions. The railway network is described in a model conforming to a railway metamodel created by Szárnyas et al. [197]. An illustration of an instance model for an railway network excerpt is depicted in Figure 1.4.

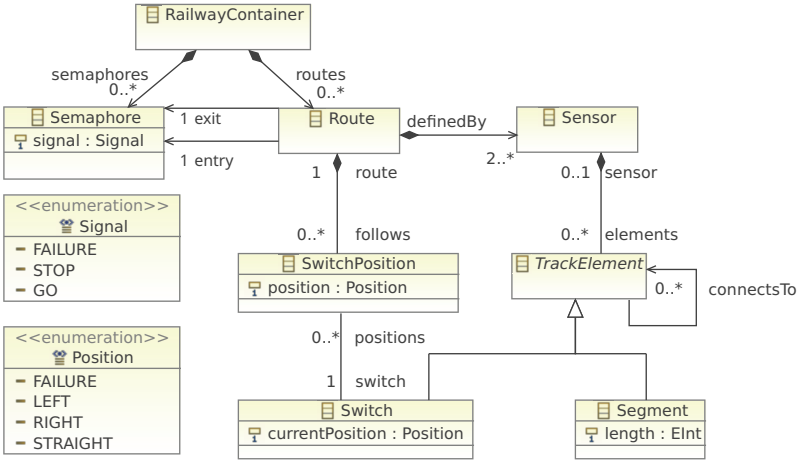


Figure 1.5.: An excerpt of the railway metamodel used in the Train Benchmark [197]

The railway network essentially consists of many segments, switches, semaphores and routes. Each route starts and ends at a semaphore and is defined by a list of switch positions which define where a train following this route should go. An excerpt of the metamodel is depicted in Figure 1.5.

One wants to make sure that if the entry semaphore shows the signal *GO*, all switches along the route should be set accordingly to the route description. The benchmark iteratively finds and fixes some violations of this and some other validation constraints.

A possible solution to this analysis is the NMF solution which can be found in Listing 1.1. According to the peer-review process in the TTC, this solution was the most understandable, even for developers not familiar to the C# language.

Line 1 takes as input a collection of all routes in the network. Line 2 selects those routes that have an entry semaphore set and that entry semaphore shows a *GO* signal. Line 3 selects the switch positions along those routes that define which switches in the network have to be set to what position. Line 4 restricts the set of switch positions in the result set of the query to those where the position of the corresponding switch does not match the required

```
1 var faultyPositions = from route in routes
2   where route.Entry != null && route.Entry.Signal == Signal.GO
3   from swP in route.Follows
4   where swP.Switch.CurrentPosition != swP.Position
5   select swP;
```

Listing 1.1: Query to find inaccurate switch positions in a collection of `routes`

position. Lastly, Line 5 specifies that the result set of the query should only contain the switch position elements.

While this solution is very hard to beat in terms of understandability and conciseness, using standard C#, the entire model has to be reevaluated whenever a model element changes. Furthermore, one has no information when the analysis should be reevaluated and a new result has to be compared with the last one in order to understand which switch positions are wrong that were not wrong before.

As a consequence, as soon as performance gets an issue, developers may start introducing cache objects, e.g. to save the routes with an entry semaphore set to the signal *GO* and dynamically registering hooks when the position of switch positions changes. However, this is a laborious and error-prone procedure as one may easily forget some cases when to update these caches. For example, one may easily forget to remove the hooks when a `SwitchPosition` element is removed from one of the routes with *GO* signal. While this maintains correctness, it slowly decreases the performance over time and is therefore hard to detect[14].

Presumably the most dramatic consequence of such an analysis inflated by caches is that domain experts likely have no longer a chance to proofread the code. Meanwhile, the code in Listing 1.1 is likely to be understood by railway experts as well, meaning they could identify possible flaws in the understanding of what this analysis should do.

Therefore, the goal of (live) implicit incrementality is to enable the system to execute the analysis from Listing 1.1 incrementally. That is, the system automatically registers event handlers to propagate model changes and issues

[14] Some approaches exist that may automatically detect such performance problems by automatically conducting experiments [215]

a notification is a change to the model caused the result of the analysis to change. However, to achieve good results, it is necessary to provide an explicit incrementalization of commonly used functions such as the query operators `from`, `where` and `select` used in Listing 1.1. To enable incrementalization systems to incorporate such dynamic algorithms is made possible through C I.1.

However, an analysis does not only consist of calls to common functions but also of domain-logic, often used for example for query predicates. In Listing 1.1, these predicates are rather simple, but they can get more complex. In that case, an incrementalization close to an instruction level yields very large and thus memory-consuming dependency graphs and make the incremental algorithm slow. Therefore, C I.2 seeks to shrink the dynamic dependency graphs and therefore reduce the memory impact.

1.7. Structure

This dissertation is divided in six parts. The remainder of Part I introduces the thesis and the used formalization. Chapter 2 introduces the foundations that this thesis is built upon. Chapter 3 introduces Mutable Type Caegories that are used to discuss formal aspects in the remainder of the thesis.

Part II is concerned with efficient implicit incremental model analyses. Chapter 4 formalizes incremental model analyses and draws conclusions on the integration of dynamic algorithms and methods of distributed computing. Chapter 5 discusses how the granularity in which to incrementalize model analyses can be increased and how a suitable granularities can be found automatically and optimized for a particular usage scenario.

Part III presents the approach for incremental model transformations. Chapter 6 takes a look into model analyses implemented in model transformations and how the results from the previous chapters can be reused for incremental model transformations.

Part IV is devoted to the influence of metamodel restrictions to incremental model analyses and how they can be mitigated. Chapter 7 presents an approach to support metamodel design to gain an improved expressiveness that makes many model analyses obsolete as they are implied by type system guarantees. Chapter 8 explores how incremental model analyses are affected

by Deep Modeling and how incrementalization tools can be reused for this modeling paradigm.

Part V validates and evaluates the approaches of all previous parts. Chapter 9 presents a series of case studies that validate and evaluate the approaches of Parts II and III. Chapter 10 reports experiences and lessons learned from conducting these case studies.

Part VI concludes the dissertation. Chapter 11 discusses related work. Chapter 12 lists limitations of the approach and gives an outlook to future work. Chapter 13 summarizes the achievements of this dissertation and draws conclusions.

2. Foundations

This chapter briefly introduces the foundations that this thesis is built upon. Section 2.1 introduces model-driven engineering, the engineering paradigm that this thesis is embedded in. Section 2.2 introduces existing definitions of incrementality. Section 2.3 introduces advanced language features of C# that are necessary to understand how the contributions of this thesis have been implemented. Section 2.4 introduces virtual actors and their implementation in the Microsoft Orleans framework. Section 2.6 introduces the theory of partially ordered sets and lattices. Section 2.7 introduces the theory of lenses, a formal construct for bidirectional transformation. Section 2.8 gives a very short introduction to category theory that will be used to formalize the results of this thesis.

2.1. Model-driven engineering

Model-driven engineering (MDE) is an approach to handle the problem of ever-increasing complexity in software development. Instead of code, domain specific models are the central software artifacts. All other software artifacts like code, documentation or test cases are then (fully or partially) generated from the models using transformations. To make transformations possible, the models have to conform to a formal definition.

The following sections introduce the main artifacts and terminology used in model-driven engineering, as they are relevant for this thesis. Furthermore, the .NET Modeling Framework and its meta-metamodel NMeta are briefly introduced as the contributions of this thesis are implemented integrated into this framework.

For a more detailed introduction to model-driven engineering, centered on its application to software development called model-driven software development (MDSD), we recommend the book by Stahl and Völter [186]. There

is also a model-driven software development process is also standardized by the Object Management Group (OMG) in the MDA standard [145].

2.1.1. Metamodels

As the formal definition of a system representation is once again a model in the domain of metamodeling, it is called a metamodel. It describes the structure of the models that conform to this metamodel. As metamodels in turn are models in the domain of metamodeling, they also have their own metamodel, referred to as the meta-metamodel. Most available meta-metamodels are self-descriptive. This prevents endless conformance sequences. The OMG standardized the meta-metamodel MOF [151]. Due to the lack of full implementations of the MOF standard, the most commonly used subset of MOF is standardized as EMOF meanwhile the full specification is now called CMOF. The presumably most common meta-metamodel in practice is Ecore, an implementation of the EMOF standard. Ecore is implemented as part of the Eclipse Modeling Framework (EMF).

In many applications, the structural description encoded in a metamodel still allows modelers to create models that are not valid in the sense that they cannot correspond to a physical system. To restrict the validity of models, static semantic rules are used to restrict the space of allowable models. These rules are usually expressed through invariants that have to hold for specific objects, often expressed using the Object Constraint Language (OCL) [157]. However, these invariants are typically not enfored automatically. Instead, the modeler has to check[15].

2.1.2. Editors

To create a model representation of a given system, modelers use editors for the metamodels that typically provide a more convenient syntax for the model than its XML Metadata Interchange (XMI) representaion.

There are two fundamentally different types of editors: graphical and textual editors. Graphical editors provide a graphical user interface and show

[15] Some approaches use static analysis to restrict when the changes that could possibly affect a constraint violation [40]

the models as diagrams where nodes and edges are defined by metamodel concepts. The model-driven developer may then edit the models by editing the diagram or creating new diagrams. Textual editors rely on a grammar that describes how models can be described in text. The syntax of these editors, either textual or graphical[16], is referred to as concrete syntax as they describe what is visible to the model developer. In contrast, the abstract syntax describes the abstract concepts that are expressed through the concrete syntaxes. The abstract syntax is defined together with the static semantics in the metamodel. An abstract syntax can have multiple concrete syntaxes as there might be multiple editors to edit instances of the same metamodel. [186]

Metamodel and concrete syntax together form a DSL [186].

Technologies such as Sirius [209] or XText [61] help to reduce the effort to create an editor.

To ensure that a model created in an editor is valid, the editor usually repetitively reevaluates the validity constraints. These can therefore be regarded as model analyses with the result being a boolean value whether the constraint is satisfied. Constraint checks are usually done either after each individual model change or manually triggered by the user. In the former case, static analysis is often used to find out which constraints may be affected by a given change [40, 204].

2.1.3. Model Transformations

The models obtained from the various editors are then transformed to either other models or traditional software artifacts. This process is usually supported by Model transformation languages (MTLs). In many cases, the overall abstraction level of the model representing the whole system is relatively high and thus model transformations transforming these models directly to the desired software artifacts (such as code) are very complex. Moreover, it is often not only the semantics that has to be transformed, it also is the syntax. Therefore, it is a widely adopted approach to split the transformation of the semantics from the transformation of the syntax. Thus, a metamodel is created that describes the structure of the target software

[16] Projective editors such as MPS [210] also allow further types of editors.

artifact. A first model transformation then transforms the input system level model to a model of the target semantics. A second transformation takes the model with the semantics already fit to the target software artifact and only transforms the syntax, i.e. prints the model in the format according to the type of software artifact that is to created. These two types of model transformations fundamentally differ. The former takes models as inputs and creates models. It is referred to as Model-to-Model (M2M) whereas the latter is referred to as Model-to-Text (M2T) or Model-to-Code [53].

The goal of M2T-transformations is to fit a model in a given structure. Such transformations contain a lot of static information like keywords or the structure of the targeted format and are thus mostly formulated as text templates. However, as they aim to transform syntax rather than semantics, it is often difficult to include complex transformation logic into these transformations.

In contrast to M2T, M2M-transformations transform models conforming to one or multiple source metamodels into a model conforming to a target metamodel. Usually, the modeling framework is used to load and save models in their default serialization format. The transformations do not care how the models are serialized. The focus is rather set to the transformation of semantics. However, there are usually correspondence relations between elements of the source model and elements of the target model that are often useful to know. These correspondences are referred to as the model transformation trace [53].

In many cases, one does not only consider the model transformation from the source model to a target model but also the backward transformation. If this is the case, the direction from the source to the target is referred to as *forward* transformation. Model transformation problems that involve both forward and backward directions are often referred as *bidirectional* transformations, abbreviated as BX. A bidirectional model transformation need not be bijective [128].

In the remainder of this thesis, if we say model transformations, we implicitly mean M2M transformations. A couple of well-suited techniques to achieve incremental execution of M2T-transformations can be found in the PhD thesis of Ogunyomi [159].

2.1.4. The .NET Modeling Framework

As Meyerovich suggests, most programmers do not easily change their primary programming languages [149]. When the adoption of MDE implies the adoption of the Java platform, this can block the adoption of MDE in domains where Java is not the primary language to work with. In addition, although MDE has existed for more than a decade now, tool support is still one of the major factors that hampers a wide adoption of MDE in industry [187, 153].

To solve this problem, the author of this thesis has created the .NET Modeling Framework (NMF)[17]. The development of this framework started with the M2M transformation language NMF Transformations Language (NTL) [92] and an XML serialization compliant to the XMI standard. By now, it is an open-source framework of libraries, tools and languages to support model-driven engineering on the .NET platform. NMF contains tools to generate model representations compliant with EMF, supports a model management repository system and allows developers to specify model analyses, model transformations and model synchronizations. To minimize both the language adoption problem and the tool support problem, NMF is entirely based on internal languages that use C# as a host language.

An introductory tutorial for NMF can be found on YouTube[18].

All contributions of NMF were made either by the author of this thesis or one of the students advised in the course of conducting research towards this thesis. Consequently, all contributions made in Parts II and III have been integrated into NMF.

2.1.5. NMeta

NMF contains its own meta-metamodel NMeta to resolve Java platform specific of the Ecore metamodel.

In NMeta, every model element is an instance of `ModelElement` and therefore has an absolute Unique Resource Identifier (URI) which makes it uniquely

[17] `http://github.com/NMFCode/NMF`
[18] `https://youtu.be/NIMYuwTltVs`

identifiable and addressable. This URI is created automatically based on the containment hierarchy of the model elements (the `Parent` reference) and on the URI of the model that contains the model element. Further, model elements also have a relative URI to identify within the scope of their model. Conversely, it is possible to resolve a relative URI starting from a given model element as context or an absolute URI in the context of a model repository.

Model elements are categorized into classes that define the type system of a model (metamodel). An excerpt of NMeta showing the metaclasses responsible for the type system is depicted in Figure 2.1. The type system is similar to the one from Ecore and there is a model transformation from Ecore to NMeta. In particular, a model conforming to an Ecore metamodel M can also be read using the transformed NMeta metamodel M', provided M does not contain generic types, factories or custom XMI handlers.

The metaclasses `StructuredType` and `ReferenceType` describe the structural features of a class. This description of structural features is extracted into separate metaclasses as these features are used also by structures and extensions, not shown in Figure 2.1. A structure is a structured value type, an extension is an implementation of a Universal Modeling Language (UML) stereotype. However, as both of these concepts are not used in Ecore, they are hardly used so far and thus a detailed description is omitted here.

NMeta uses naming conventions from the .NET platform without an explicit prefix, for example the reference `eSuperTypes` is called `BaseTypes`.

Same as CMOF [151, p.43], NMeta only allows one attribute to be the identifier, but makes this constraint more explicit. This identifier is used to identify elements either locally, i.e. in the scope of their container, or globally, i.e. in the scope of the model they reside in. In both cases, an identifier overrides the way a URI is computed for a given model element, though NMF also allows to switch to a referencing scheme based on indices for local identifiers.

2.2. Incrementality

In complexity theory, dynamic problems or dynamic algorithms are stated in terms of changes of input rather then original input. There, incremental algorithms mean algorithms that handle additions (increments), but no deletions. In the context of model-driven engineering, incrementality usually

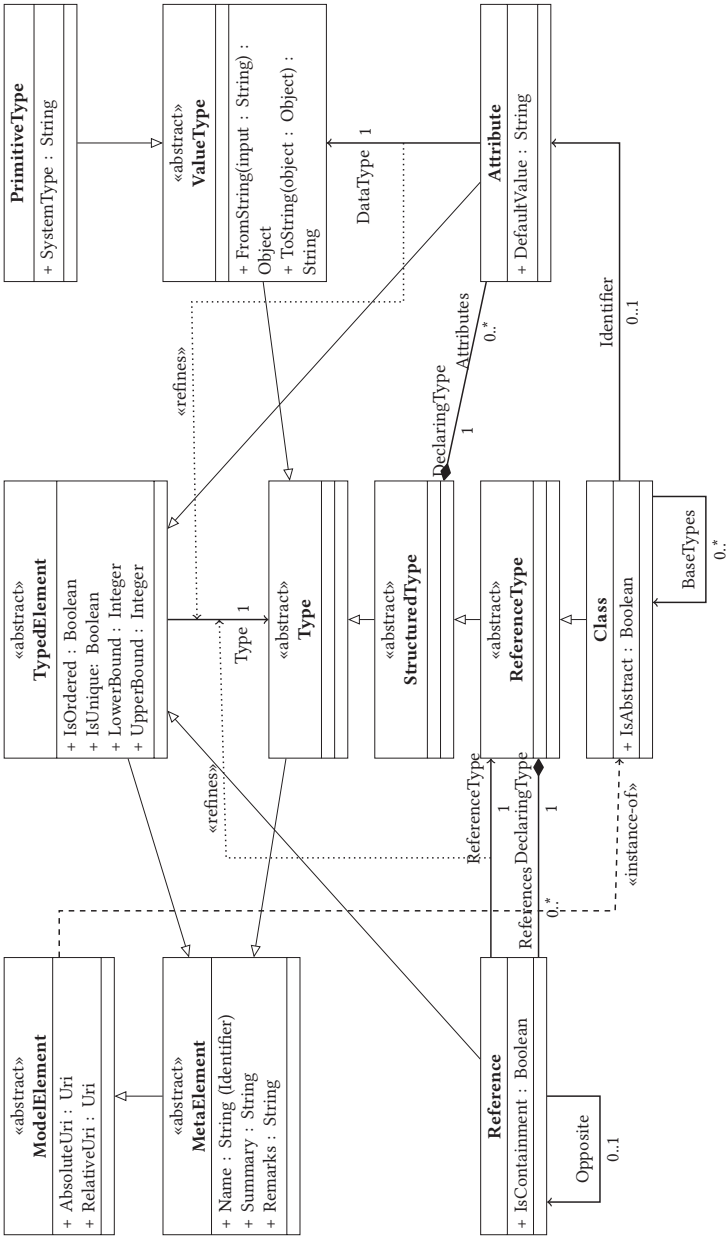


Figure 2.1.: The type system of NMeta (simplified)

refers to incremental advent of changes to a given model. However, these changes may also be decremental, for example deleting model elements or references. Therefore, from a complexity theory point of view, the thesis is about implicitly creating fully dynamic algorithms for arbitrary analyses and transformations.

For model transformations, Czarnecki et al. present a threefold definition of incrementality [53]: A model transformation is *target-incremental* if it incrementally reuses the target model instead of recreating a new every time the transformation is executed. It is *source-incremental* if only changes of the source model are processed instead of the entire model transformation. A transformation *preserves user changes* if changes made by a user after a model has been transformed are not overridden by the transformation engine.

In the terminology of Czarnecki, the model analyses and transformations considered in this thesis are source-incremental but not necessarily target-incremental[19]. Especially for model analyses, it is not a common scenario that a user would change a target model and the changed need to be preserved. Further, target-incrementality is often not important, though it is often a consequence of source-incrementality. However, in Chapter 5, the target-incrementality is slightly relaxed and may be violated to improve performance.

Hearnden et al. [85] suggest an implementation strategy of model transformations that are both source- and target-incremental as live model transformations. Here, the transformation runs continously and monitors changes of the source model that are propagated to the target model in small batches. This idea matches quite closely the understanding of an incremental analysis or transformation followed in this thesis.

Giese and Wagner [74] define incremental model transformations through asymptotic complexity: A model transformation is *fully incremental* if the complexity of propagating a change only depends at most linearly on the size of the change rather than the size of the model. They call a model transformation *effectively incremental* if the complexity of propagating a change depends on the size of the model as well but less then linear such

[19] The incrementalization system presented in Chapter 4 is target-incremental unless combined with the approach from Chapter 5.

as for example logarithmic. In both cases, the complexity is taken in the average case.

The problem with this definition is that the question whether a model transformation is incremental depends on the average impact of changes, i.e. which parts of a model transformation have to be recomputed. Except for trivial models, there is a huge set of potential model changes. The frequency of their occurrence is unclear and cannot be deduced from the analysis or transformation.

Furthermore, in the general case of arbitrary analyses, the impact of a change is very hard to analyze. Meanwhile, the definition by Giese and Wagner makes the assumption that model transformations always have linear complexity. Especially for model transformations that perform matchings of multiple models, this is not true. For model analyses, we also do not want to make this assumption.

Therefore, whether a given analysis is fully incremental or effectively incremental depends on the usage, in particular on the frequency of changes.

2.3. Advanced C# Language Features

Throughout the thesis, a couple of rather advanced language features of the C# programming languages are used that are hardly present in other programming languages and therefore hardly known for developers not familiar with C#. However, a basic understanding of these features is necessary to understand how the contributions of this thesis are implemented. Hence, they are briefly described in the remainder of this section.

2.3.1. Expression Trees

Starting with version 3.0, the C# language supports a feature called *expression trees*. This feature allows developers to obtain a model of a function instead of already compiled code. This model already contains information from semantic analysis. That is, missing type information is inferred and references to methods or other members are already resolved. These analysis steps are done at compile-time, only the last step, the output of intermediate language code, is changed.

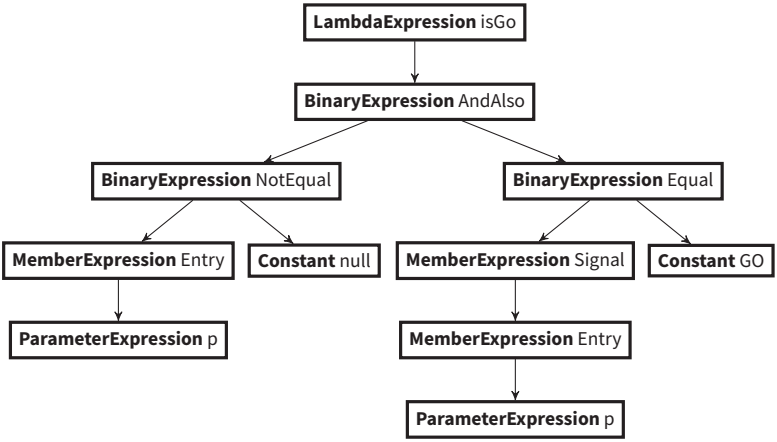


Figure 2.2.: The expression tree for the lambda expression to obtain whether the entry signal of a route shows *GO*.

To enable this feature, all the programmer has to do is to use a dedicated function type called `Expression<>` around the regular function type `Func<>`. As an example, we depicted the predicate whether the entry semaphore of a route shows GO in Listing 2.1.

```
1 Expression<Func<Route, bool>> isGo =
2   route => route.Entry != null && route.Entry.Signal == Signal.GO;
```

Listing 2.1: Lambda Expressions in C#

The result of an assignment as in Listing 2.1 is an *expression tree* that is depicted in Figure 2.2.

This expression tree closely resembles the abstract syntax tree of the function specified by the developer, yet already with the results from semantic analysis. That is, each node knows its type, member expressions carry a reference to a reflection object referencing the member they are accessing and implicit conversions as well as type inference are considered in the tree.

```
1 var faultyPositions = routes
2   .Where(route => route.Entry != null && route.Entry.Signal == Signal.GO)
3   .SelectMany(route => route.Follows,
4               (route, swP) => new { route = route, swP = swP })
5   .Where(_t => _t.swP.Switch.CurrentPosition != _t.swP.Position)
6   .Select(_t => _t.swP);
```

Listing 2.2: Query to find inaccurate switch positions in a collection of routes (as treated by the compiler)

2.3.2. Language Integrated Query and the C# Query Syntax

Language Integrated Query (LINQ) is a framework integrated into the .NET framework to allow developers to specify queries against objects, XML documents or even databases integrated into their primary programming language, which is often C# or VB.NET [147]. For this purpose, Microsoft defined the Standard Query Operators (SQOs) which essentially is a query Application Programming Interface (API) consisting of extension methods, i.e. static methods with a special annotation such that they are treated like member methods of their first parameter's type.

Examples of SQO methods are the query operators `Select`, `Where` or `SelectMany`. A comprehensive list of API methods and their semantics can be obtained online[20].

The compilers of the .NET languages use this API to offer a special and dedicated syntax for it. In case of C#, this is the C# query syntax. An example of this syntax can be seen in Listing 1.1 on page 23.

At compile time, the compiler converts the query syntax into a chain of methods, generating anonymous types where necessary. For example, Listing 1.1 is internally handled very close to Listing 2.2[21]. This style of writing queries is also directly supported in the language and is often called the method chaining syntax.

The SQO methods are resolved using the normal name resolution rules of the respective language. Depending on the type of `routes` and the extension

[20] `https://msdn.microsoft.com/en-us/library/bb394939.aspx`, retrieved 15 Feb 2017
[21] The names of compiler-generated identifiers and classes are more sophisticated

```
1 var faultyPositions = routes
2   .Where(route => route.Entry != null && route.Entry.Signal == Signal.GO)
3   .SelectMany(route => route.Follows)
4   .Where(swP => swP.Switch.CurrentPosition != swP.Position);
```

Listing 2.3: Query to find inaccurate switch positions in a collection of routes (method chaining syntax)

methods available in the current scope, the chosen SQO methods in Listing 2.2 may be different. Therefore, developers may change the semantics of how such a query is executed by providing different implementations of the SQO methods.

To resolve these extension methods, the compiler actually allows the predicates to be required either as regular compiled functions, but it also allows expression trees. This is how LINQ translates queries specified in C# to database queries in SQL [147].

Note that the method chain generated by the C# compiler is (at least at the time of writing) not optimal[22]. For instance, a better method chaining translation of Listing 1.1 would be the method chaining syntax depicted in Listing 2.3 that requires only three SQO calls instead of four in Listing 2.2.

The fact that the implementation of SQO methods can be exchanged makes the syntax very appealing for incremental analyses: The query syntax is very concise and easy to understand. In particular, the SQO API allows predicates to be specified either as compiled functions or as expressions: In the latter case, the compiler generates a fully typed abstract syntax tree of the provided lambda expression instead of generating the entire code.

The Language Integrated Query (LINQ) is an example of an SQO implementation. It uses the option to obtain predicates as typed abstract syntax trees and translates these expressions to other languages such as SQL in order to execute the query on a database. The result of the query is then converted to objects through an integrated object-relational mapper. Since the translation of the query expression is encapsulated in a component, LINQ provides a common interface for querying data from a vast variety of information

[22] Here, optimality is with regard to the number of SQO method calls generated.

sources. This system is also extensible so that new information sources can be added by just implementing a new provider implementation.

However, in the context of this dissertation, LINQ is not relevant. This is because the interfaces that LINQ is built upon only consider a batch scenario where data is retrieved and then the connection to the data source is lost[23].

The query syntax of C# can be thought of as built-in monad support as queries can be considered as extensions of collections to a monad (cf. Section 2.8). The syntax is tailored for monads that (like in Haskell) focus on the bind function (as an extension method called `SelectMany`). The framework design guidelines of the .NET framework [52] even state that the query syntax should only be used by collections. As these guidelines are used across all the .NET framework, it would be confusing for developers to break them.

2.4. Virtual Actors

Gul Agha has proposed the actor model to overcome problems typically induced by parallel computation such as deadlocks [4]. In this model, actors asynchronously send messages to other actors while the implementation of these messages itself is synchronous.

The actor model has been implemented in the Microsoft Orleans framework [38, 39] which also extended the model to virtual actors where virtual means that the existence of an actor (called grain) is independent of a direct representation in memory (activation). Rather, if no activation exists, the actor is transparently loaded from persistent storage into a silo, a container of grains.

While a silo is bound to a physical machine, grains may interact with grains from other silos, enabling support for distributed systems. An auto-tuning agent is able to move grains across silos in order to increase locality and raise performance. Since the framework has direct support for a deployment to Windows Azure, this can be used in elastic cloud computing scenarios.

[23] The object-relation mapper allows to save changes to objects in the data source, but changes made in the database are not reflected locally.

The Orleans framework has been used for large distributed applications such as the online services for the Halo 5 computer game [28], distributed matrix-vector multiplication or the Horton project [180] for large graph processing.

2.5. The Palladio Component Model (PCM)

PCM is the metamodel used in the Palladio simulation tool suite [21]. Its goal is to describe the architecture, deployment and usage of a component-based system in a level of detail that is sufficient to analyze the system for its non-functional properties but on the other hand higher than the actual implementation. While the metamodel was originally created to predict performance, it has been extended to also target reliability [30], energy consumption [192] or security [183]. In its current state, PCM is entirely implemented using strict two-level metamodeling. The metamodel consists of more than 200 classes, divided in more than ten packages. It further builds upon several other metamodels that model for instance stochastic expressions.

PCM internally consists of four models [21]:

- The repository model contains the components and their interfaces. For each service offered by a component, it also contains a rough behavior description called service effect specification (SEFF) that models the resource demands when executing this service.
- The composition model specifies which of the components in the repository are instantiated in a system architecture and how they are connected to each other.
- The allocation model specifies how the instantiated components of a system's architecture are deployed to available resources.
- The usage model specifies how the system is used, for example what services are typically called by users in which intervals. This model is again outside the scope of this paper and thus not described in more detail.

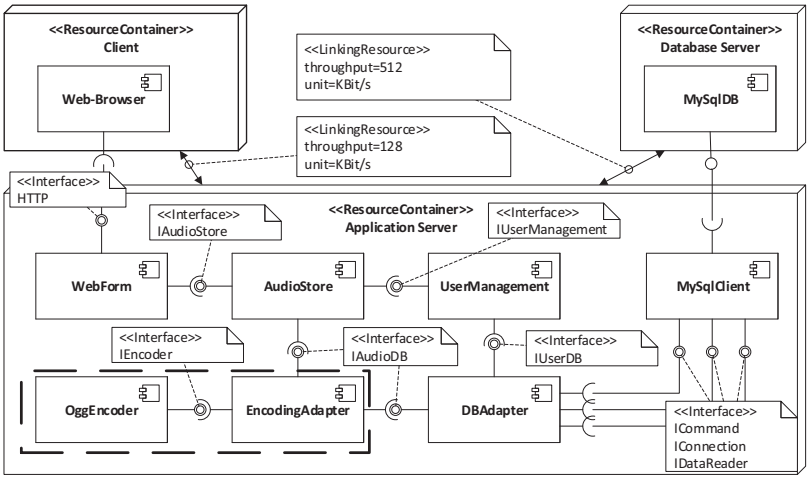


Figure 2.3.: An example e-commerce system in PCM cf. [21]

Figure 2.3 shows an example e-commerce system modeled in Palladio where these models have been merged into a single view.

Because PCM only uses two-level modeling, it contains a range of helper constructs to represent e.g. instantiated components as instantiation itself is not available. The components in this diagram form the architecture of the e-commerce application. In PCM, the types of these components are stored in a repository to make it easier to reuse these components in a different application. In 2.3, all component types in the underlying repository are instantiated exactly once such that the component instances in Figure 2.3 have the same names as their corresponding component types. The surrounding boxes denote the resource containers to which these components are deployed.

To get a rough idea of how the assembly of a system is realized in PCM, we have depicted a simplified metamodel excerpt in Figure 2.4. An instantiated component (*AssemblyContext*) is realized as a class that holds a reference to its actual type, namely the component that is instantiated in the assembly. Then, the connections of such an instantiated component are realized using separate connector elements (*AssemblyConnectors*) that specify to which other assembly context the current context is connected to for a given re-

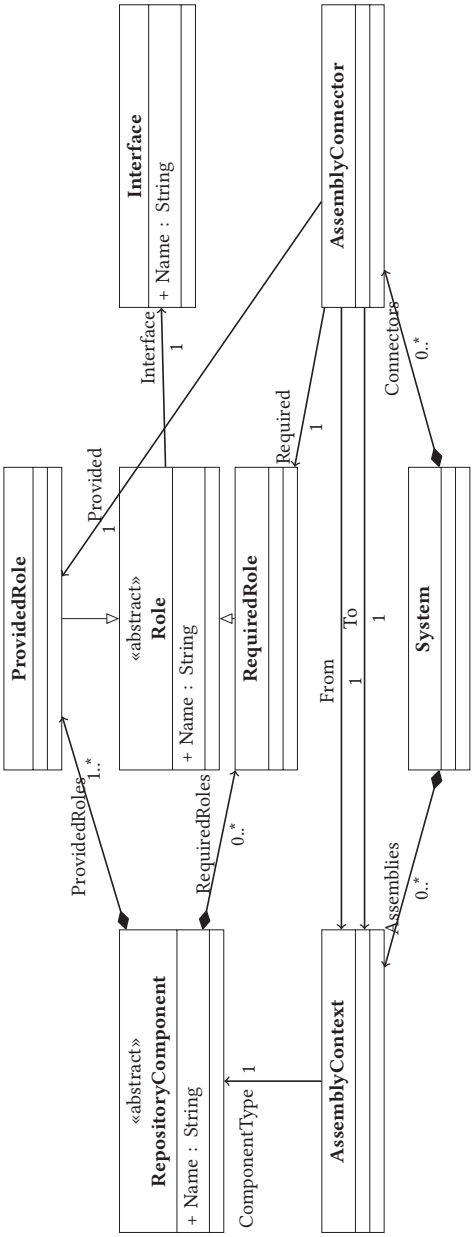


Figure 2.4.: System assembly in PCM (simplified excerpt)

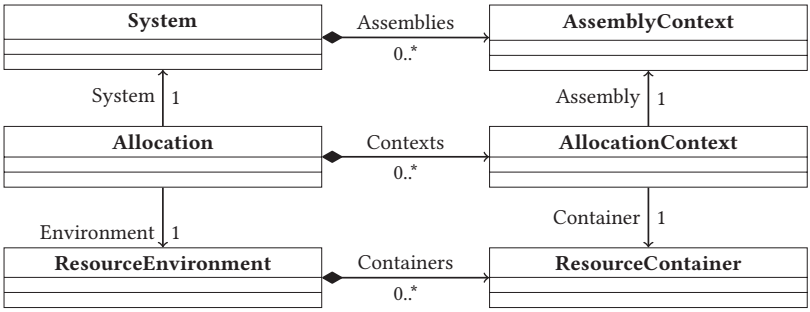


Figure 2.5.: System deployment in PCM (simplified excerpt)

quired interface. OCL constraints ensure that for each required interface of an assemblies component, the system architecture contains a connector that connects this assembly to another assembly whose component provides this interface.

A system then also may have provided roles (the composition is actually shared by a common base class not depicted in Figure 2.4) that serve as interface of the system towards the user. A different kind of connector elements is then used to map these system provided roles to provided roles of the inner assemblies.

Components may either be basic components – components with no detailed inner structure – or composite components. Composite components are essentially a combination of systems and repository components and represent components that realize their provided interfaces through delegation to other components. In the example of Figure 2.3, one may want to bundle the two components for encoding data streams together in a composite component. The advantage of such a composite component is that the decision to use the *OggEncoder* component for encoding is encapsulated in the composite component and can be exchanged independent of how often this composite component is instantiated.

The metamodel excerpt responsible for the deployment of assembly contexts is roughly depicted in Figure 2.5. An allocation essentially consists of allocation contexts that each describe to which resource container (e.g., machine) an assembly context should be deployed. The resource containers are modeled in a resource environment that specifies the resources available at each

resource container as well as their connection through linking resources such as e.g. network.

2.6. Partially Ordered Sets and Lattices

This section gives a brief introduction to the theory of partially ordered sets and lattices. These theories will be used in the remainder for a formalization of of type systems and their applications. The definitions can be found in many text books on partial orders. The definitions depicted here are taken from Crole [49].

Definition 1 (Partial order)**.** A partial order $\leq$ on a given set X is a binary relation which is reflexive, transitive and anti-symmetric. This means, for any given $x, y, z \in X$ the relation $\leq$ fulfills the following properties:

$$x \leq x \qquad \text{(reflexive)}$$
$$x \leq y \wedge y \leq z \Rightarrow x \leq z \qquad \text{(transitive)}$$
$$x \leq y \wedge y \leq x \Rightarrow x = y \qquad \text{(anti-symmetric)}$$

Definition 2 (Partially ordered set)**.** A partially ordered set $(X, \leq)$ is a set X equipped with a partial order $\leq$. If the partial order $\leq$ is clear from the context, we also simply write X.

Definition 3. Let X be a partially ordered set. Then an upper bound $\hat{s}$ of a subset $S \subset X$ of X is an element such that for each $s \in S$ we have that $s \leq \hat{s}$. Likewise, a lower bound for S is an element $\check{s}$ for which we have that for every $s \in S$ it holds that $\check{s} \leq s$.

Definition 4 (Join and Meet)**.** A join of a subset $S \subset X$ of X is the supremum of S with respect to $\leq$, denoted as $\bigvee S$. This means, $\bigvee S$ is an upper bound and for any other upper bound $\hat{s}$ of S, we have that $\bigvee S \leq \hat{s}$. Likewise, a meet of S is an infimum of S with respect to $\leq$ denoted as $\bigwedge S$, i.e. it is a lower bound of S and for every other lower bound $\check{s}$ we have that $\check{s} \leq \bigwedge S$.

A join (meet) of a subset consisting only of two elements a and b is also denoted as $a \vee b$ ($a \wedge b$). Joins and meets of finite subsets are called finite joins and meets.

Definition 5 (Lattice, Semilattice)**.** A lattice is a set equipped with a partial order such that any finite meet and join exists. If only finite meets (joins) exist, then it is called a meet-semilattice (join-semilattice).

Example 1. Any totally ordered set such as the real numbers are lattices and therefore of course also meet- and join-semilattices.

Example 2. For a given set X, the powerset $\mathbb{P}(X)$ with set-inclusion is a lattice. For a given set $S \subset \mathbb{P}(X)$, the join $\bigvee S = \bigcup S$ is the union of these sets, while the meet $\bigwedge S = \bigcap S$ is their intersection.

2.7. Lenses

Lenses are a an algebraic construct originally introduced by Foster et al. [64] to solve the view-update-problem for tree structures. They operate on a set of tree structures $\mathcal{V}$ and are able to compute updates to the original tree when views have changed.

Definition 6 (Lens)**.** A *lens* l is a pair of two partial functions $l \nearrow: \mathcal{V} \rightharpoonup \mathcal{V}$ called the Get-function of l and $l \searrow: \mathcal{V} \times \mathcal{V} \rightharpoonup \mathcal{V}$ called the Put-function of l. The intuition is that $l \nearrow$ computes a view on an element while $l \searrow$ applies changes to the view back to the original element.

Definition 7 (Well-behavedness)**.** Let C and A be subsets of $\mathcal{V}$. A lens is called well-behaved and total from C to A if it maps arguments of C to results of A ($l \nearrow (C) \subset A$ and $l \searrow (A \times C) \subset C$) and complies with the following laws:

$$l \searrow (l \nearrow (c), c) = c \quad \text{for all } c \in C \tag{GetPut}$$

$$l \nearrow (l \searrow (a, c)) = a \quad \text{for all } (a, c) \in A \times C. \tag{PutGet}$$

Intuitively, these laws state that when no modification is performed in the view, the Put function should not modify the original element; otherwise, it should store this information such that recomputing the view would not change the results.

Definition 8. The composition operator ; puts two lenses l and k in sequence:

$$(l;k) \nearrow : c \mapsto k \nearrow (l \nearrow (c))$$
$$(l;k) \searrow : (a,c) \mapsto l \searrow (k \searrow (a, l \nearrow (c)), c).$$

Proposition 1. The composition $l;k$ of a well-behaved total lense l from A to B and a well-behaved, total lense k from B to C is a well-behaved, total lense from A to C.

2.8. A very short Primer to Category Theory

The goal of this section is to introduce the category theory that is used in this thesis for reference purposes. The concepts are not explained as suitable explanations can be obtained from many textbooks on category theory. The interested reader is referred to Lawvere and Rosebrugh [137] or Crole [49]. The definitions from this section are taken from the latter.

2.8.1. Categories

Definition 9 (Category)**.** A category C consists of a collection $ob\,C$ of objects and collections of morphisms between objects of C equipped with an associative operator $\circ$. The morphisms between objects $A, B \in ob\,C$ are denoted as $Mor_C(A,B)$ or $Mor(A,B)$ if C is clear from the context. Furthermore, for each object A, the identity id_A must exist and for each $f \in Mor(A,B)$, it must hold that $f \circ id_A = f = id_B \circ f$.

Remark 1. The associtativity means that for any $f \in Mor(A,B)$, $g \in Mor(B,C), h \in Mor(C,D)$ where $A,B,C,D \in ob\,C$ that $(h \circ g) \circ f = h \circ (g \circ f)$.

Remark 2. In category theory, equations are often visualized as diagrams. For this, one uses objects of the category as nodes of the graph and morphisms between these objects as edges. Due to composition of morphisms, any paths in this graph are also morphisms in the category. The terminology that such

a diagram commutes is equivalent to saying that the morphisms generated by following multiple paths between objects are the same.

Definition 10. A category C is called *small* if its objects $ob\,C$ form a set and likewise every class of morphisms $Mor(A, B)$ for any objects A and B is a set.

Example 3 (Trivial Category)**.** The trivial category $\mathcal{T}(S)$ for a given set S is a category with $ob\,\mathcal{T} = S$ and the only morphisms are the identities for each object.

Example 4 (Sets)**.** One of the most important categories is the category $\mathcal{S}$ of sets. Here, the morphisms are the mappings between sets and the identity for a given set A is the identity mapping on A.

The category $\mathcal{S}$ is not small. If it was small, then its objects would form a set and this set of objects would have to be an element of itself. This is forbidden in axiomatic set theory.

Example 5 (Type systems)**.** As shown by Crole [49], every algebraic type system corresponds to a (cartesian-closed) category. In the running example, the railway network type system is a category, where the objects are the types such as `Semaphore`, `Switch` and `Route`. The morphisms are the reflexive-transitive closure of the model properties between these types, such as for instance the *position* of a switch, but also combinations such whether the position of the first switch in some collection of switches matches a given (constant) value.

Definition 11. Let C be a category. The opposite category C^{op} has the same objects as C and for each objects A, B, we have that $Mor_{C^{op}}(A, B) = Mor_C(B, A)$.

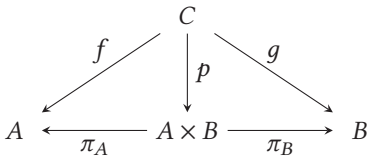
Example 6. Every partially ordered set $(S, \leq)$ gives rise to a category C by setting $ob\,C = S$ and for each objects A, B of C, we have that $Mor(A, B) \neq \emptyset :\Leftrightarrow A \leq B$. In particular, $Mor(A, B)$ contains at most one element.

Example 7. The category $\mathcal{G}$ of groups consists of the collection of groups together with the group homomorphisms. A similar construction is possible for a range of algebraic constructs such as rings or fields.

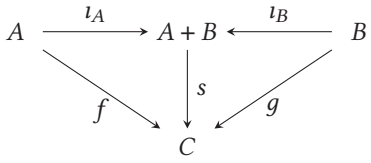
Remark 3. The goal of this range of examples is to show the generality of categories. However, this has a downside that so far only few theorems are known that hold on this very general level. Such a theorem simply has to hold for a very diverse range of mathematical entities. Thus, the main benefit from category theory is its universal language.

In particular, category theory defines a range of properties defined on this very abstract level, called universal properties. To prove whether a given category possesses such a property usually requires to use the inner structure of this category but the language offered by category theory can be very beneficial for communication.

Definition 12 (Product, Sum)**.** Let A and B be objects of a category $\mathcal{C}$. The product of A and B in $\mathcal{C}$ is an object $A \times B$ of $\mathcal{C}$ together with two projection morphisms $\pi_A : A \times B \to A$ and $\pi_B : A \times B \to B$ such that for every object C and every pair of morphisms $f : C \to A$ and $g : C \to B$, there is a unique morphism $p : C \to A \times B$ such that $f = \pi_A \circ p$ and $g = \pi_B \circ A$. That is, the following diagram commutes:

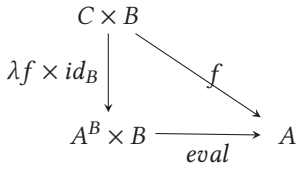


A sum of objects A and B in $\mathcal{C}$ simply is the product of A and B in $\mathcal{C}^{op}$. That is, it is an object $A + B$ together with two morphisms $\iota_A : A \to A + B$ and $\iota_B : B \to A + B$ such that for every object C and every pair of morphisms $f : A \to C$ and $g : B \to C$, there is a unique morphism $s : A + B \to C$ such that $f = s \circ \iota_A$ and $g = s \circ \iota_B$. That is, the following diagram commutes:



Proposition 2. Sums and products for given objects A and B in C are unique up to isomorphism, in case they exist.

Definition 13 (Exponential)**.** Let C be a category such that for each objects A and B their product exists. Then the exponential of A and B is an object A^B together with a morphism $eval : A^B \times B \to A$ such that for any morphism $f : C \times B \to A$, there is a unique morphism $\lambda f : C \to A^B$ such that for every $c \in C$ and $b \in B$, $f(c, b) = eval(\lambda f(c), b)$. That is, the following diagram commutes:



Definition 14 (Initial object, terminal object)**.** An initial object $\bot$ of a category C is an object such that for every object A in C, there exists exactly one morphism from $\bot$ to A.

Conversely, a terminal object $\top$ of a category C is an object such that for every object A in C, there exists exactly one morphism from A to $\top$.

An initial object of C is a terminal object of C^{op} and vice versa.

Proposition 3. Initial and terminal objects are unique up to isomorphism, i.e. if A and B are initial objects of the same category, then there is an isomorphism from A to B.

Example 8. In the category $\mathcal{S}$ of sets, the initial object is the empty set. The terminal objects are the sets that contain exactly one element.

Example 9. In programming, initial and terminal objects usually correspond to the type of `void` and `null`. This is because for any given type, there is exactly one morphism that returns `null` (because there is no other choice) and there is exactly one morphism from `void` to that type (that does not have a specification because the type `void` allows no instances).

Definition 15. A category C is called cartesian-closed if it satisfies the following properties:

- It contains an initial and a terminal object.
- For any objects A and B, the product $A \times B$ exists.
- For any objects A and B, the exponential A^B exists.

Example 10. The category $\mathcal{S}$ of sets is cartesian-complete. The binary product of two sets A and B is the set-theoretic product $A \times B$ and the exponential A^B is the set of functions $B \to A$.

2.8.2. Functors and Monads

Definition 16 (Functor)**.** A (covariant) functor $\mathcal{F} : C \to \mathcal{D}$ between categories C and $\mathcal{D}$ is a mapping between the objects of C and $\mathcal{D}$ and the morphisms such that for each objects A and B and $f \in Mor(A, B)$ in C, we have that $\mathcal{F}(f) \in Mor_{\mathcal{D}}(\mathcal{F}(A), \mathcal{F}(B))$. Further, a functor has to respect composition, i.e. if $f : A \to B$ and $g : B \to C$, then it must hold that $\mathcal{F}(g \circ f) = \mathcal{F}(g) \circ \mathcal{F}(f)$ and $\mathcal{F}(id_A) = id_{\mathcal{F}(A)}$ in $\mathcal{D}$ and for each object A in C.

A contravariant functor of C is a covariant functor of C^{op}. However, contravariant functors will not be a subject of consideration in this thesis. Whether a functor is covariant or contravariant is usually clear from context so that one often omits this information.

A functor $\mathcal{F} : C \to C$ for some category C is called an endofunctor.

Example 11 (Identity functor)**.** An important functor is the identity functor $id_C : C \to C$ for a category C that maps each object $A \in C$ to itself and likewise each mapping $f \in Mor(A, B)$ to itself.

Example 12. Further, if A is an object of a small category C, then $Mor(A, -)$ is a functor from C to $\mathcal{S}$. It maps each object B of C to $Mor(A, B)$ and each mapping $f \in Mor(B, C)$ to $Mor(A, f) : Mor(A, B) \to Mor(A, C), \phi \mapsto f \circ \phi$.

Example 13. There are three prominent collection functors on $\mathcal{S}$:

1. The powerset functor $\mathbb{P} : \mathcal{S} \to \mathcal{S}$ sends each set to its powerset and for each morphism $f : A \to B$ we have that

$$\mathbb{P}(f) : \mathbb{P}(A) \to \mathbb{P}(B), S \mapsto f(S) := \{f(s) | s \in S\}.$$

2. The multiset functor $\mathbb{M} : \mathcal{S} \to \mathcal{S}$ sends each set S to the set of multisets with elements of S, i.e. to a function $S \to \mathbb{N}_0$ that assigns each element a multiplicity in the multiset. A morphism $f : A \to B$ is mapped to

$$\mathbb{M}(f) : \mathbb{M}(A) \to \mathbb{M}(B), m \mapsto (b \mapsto \sum_{a \in f^{-1}(\{b\})} m(a)).$$

3. The Kleene closure $* : \mathcal{S} \to \mathcal{S}$ maps each set A to its Kleene closure $A*$ which is the monoid of finite sequences of elements of A. A morphism $f : A \to B$ is mapped to

$$*(f) : A* \to B*, (a_1; \ldots; a_n) \mapsto (f(a_1); \ldots; f(a_n)).$$

All three of these functors will be important in the remainder of this thesis.

Remark 4. Functors are the 'natural' mapping constructs between categories. This is because indeed, the collection of categories forms the category $\mathcal{C}at$ where the morphisms between categories $\mathcal{C}$ and $\mathcal{D}$ (which are themselves objects of $\mathcal{C}at$) are the functors $\mathcal{F} : \mathcal{C} \to \mathcal{D}$.

Definition 17 (Natural transformation)**.** A natural transformation $\eta : \mathcal{F} \to \mathcal{G}$ between two functors $\mathcal{F}, \mathcal{G} : \mathcal{C} \to \mathcal{D}$ is a set of mappings $\eta_A \in Mor(\mathcal{F}(A), \mathcal{G}(A))$ for each $A \in \mathcal{C}$ such that for each $A, B \in ob\,\mathcal{C}$ and $f \in Mor(A, B)$ it holds that $\eta_B \circ \mathcal{F}(f) = \mathcal{G}(f) \circ \eta_A$. That is, the following diagram commutes:

$$\begin{array}{ccc} \mathcal{F}(A) & \xrightarrow{\mathcal{F}(f)} & \mathcal{F}(B) \\ {\scriptstyle \eta_A}\downarrow & & \downarrow{\scriptstyle \eta_B} \\ \mathcal{G}(A) & \xrightarrow[\mathcal{G}(f)]{} & \mathcal{G}(B) \end{array}$$

If all η_A are isomorphisms, η is called a natural isomorphism between $\mathcal{F}$ and $\mathcal{G}$.

Remark 5. With natural transformations, category theory offers a formal definition of naturality that is provable.

Example 14. An important example of a natural transformation between functors is the identity transformation on a given functor $\mathcal{F}$. For each object A in C, the transformation component for A is simply the identity, i.e. $(id_{\mathcal{F}})_A = id_{\mathcal{F}(A)}$.

Definition 18 (Monad)**.** A monad $\mathcal{T} : C \to C$ is a functor equipped with two natural transformations $\eta : id_C \to \mathcal{T}$ and $\mu : \mathcal{T}^2 \to \mathcal{T}$ such that $\mu \circ \mathcal{T}\mu = \mu \circ \mu\mathcal{T}$ and $\mu \circ \mathcal{T}\eta = \mu \circ \eta\mathcal{T} = id_{\mathcal{T}}$.

Example 15. The powerset functor from Example 13 can be extended to a monad with the following natural transformations: Let A be a set in $\mathcal{S}$, then

$$\eta^{\mathbb{P}}_A : A \to \mathbb{P}(A), a \mapsto \{a\}$$
$$\mu^{\mathbb{P}}_A : \mathbb{P}(\mathbb{P}(A)) \to \mathbb{P}(A), S \mapsto \bigcup S.$$

$\eta^{\mathbb{P}}$ is natural since $f(\{a\}) = \{f(a)\}$ for $a \in A$ and $f : A \to B$. The naturality of $\mu^{\mathbb{P}}$ follows from $\bigcup_{S\in\wp} f(S) = f(\bigcup_{S\in\wp})$. This time, $\wp \in \mathbb{P}(\mathbb{P}(A))$ is a subset of subsets of A.

To proof that $\mathbb{P}$ is a monad, it is even necessary to go one more functor level deeper than before. Thus, let $X \in \mathbb{P}^3(A)$ be a subset of subsets of subsets of A. Then

$$(\mu \circ \mu\mathbb{P})_A(X) = \bigcup_{\wp\in X} \cup_{S\in\wp} S = (\mu \circ \mathbb{P})_A(X)$$

and for each $S \in \mathbb{P}(A)$

$$(\mu \circ \eta\mathbb{P})_A(S) = S = \bigcup_{a\in S} \{a\} = (\mu \circ \mathbb{P}(\eta))_A(S).$$

Hence, $\mathbb{P}$ is a monad.

Remark 6. The difference between $\mu \circ \mathbb{P}\mu$ and $\mu \circ \mu\mathbb{P}$ is in the order the sets of sets of sets are flattened. In the first case, the inner sets of sets of A are flattened first, while in the latter the outer sets of sets of $\mathbb{P}(A)$ are merged first.

Likewise, the operator $\mu \circ \mathbb{P}\eta$ first packages every item $a \in S$ into a subset $\{a\}$ and then flattens these subsets, meanwhile $\mu \circ \eta\mathbb{P}$ simply maps S to $\{S\}$ and flattens this one-element set of sets to S.

For both equations, the representation suffers from the fact that the mathematical set union notation already ignores the order in which unions are computed as this does not matter in $\mathcal{S}$.

Example 16. Like the powerset monad, the Kleene functor can also be extended to a monad using the natural transformations η^* and μ^* as follows:

$$\begin{aligned}
&\eta^*_A : A \to A*, a \mapsto a;\\
&\mu^*_A : A** \to A*,\\
&\quad (a^1_1; \ldots; a^1_{n_1}); \ldots; (a^m_1; \ldots; a^m_{n_m});\\
&\quad \mapsto a^1_1; \ldots; a^1_{n_1}; a^2_1; \ldots; a^{m-1}_{n_{m-1}}; a^m_1; \ldots; a^m_{n_m};
\end{aligned}$$

Similarly, the functor $\mathbb{M}$ can be extended to a monad through the natural transformations $\eta^{\mathbb{M}}$ and $\mu^{\mathbb{M}}$ as follows:

$$\begin{aligned}
&\eta^{\mathbb{M}}_A : A \to \mathbb{M}(A), a \mapsto (b \mapsto \delta(a, b))\\
&\mu^{\mathbb{M}}_A : \mathbb{M}^2(A) \to \mathbb{M}(A), M \mapsto (a \mapsto \sum_{m \in \mathbb{M}(A)} M(m) \cdot m(a).
\end{aligned}$$

3. Mutable Type Categories

In this chapter, we introduce the notion of mutable type categories. The rationale behind this formalization is to extend the static category-theory based formalization of algebraic type systems as given by Crole [49] with a notion of state in order to formally denote a reaction of state changes in incremental computing.

Section 3.1 explains the structure of mutable type categories. Section 3.2 introduces the notion of stateless methods and stateless types used in the thesis. The next sections define the semantics and present some results for several modeling features. In particular, Section 3.3 inheritance, Section 3.4 collections, Section 3.5 composition references and the composition hierarchies defined by them, Section 3.6 opposite references.

3.1. Types and State as a Category

The goal of this section is to introduce a formalization of type systems based on category theory. In particular, this section will introduce a formalization of basic properties of a type system which are then completed in later sections.

The basic idea is to interpret types in a type system as objects of a category, similar to Croles mapping of algebraic type systems to categories [49]. Each type object represents the set of possible objects of this type. In contrast to Crole, however, we consider the mutable state of objects at runtime. This has a multitude of consequences: First, the value of a member access of an object may be different, depending on the state in which the member was accessed, but the identity of the object is assumed to stay the same. Second, the added complexity in the formalization by Crole to cope with generic methods and functions is not necessary because at runtime, each method is bound already to a type, i.e. we do not have to take open generic type definitions into

account. Therefore, we basically rest on the proof made by Crole that any algebraic type system is equivalent to a cartesian-closed category.

While this foundation on algebraic type theory is useful for many practical applications, we need to keep in mind that model elements have an identity that stays the same even though its attributes or references may change. In particular, models are mutable. This is because according to the general model theory by Stachowiak [185], models always correspond to an original or concept whose identity does not change either.

The goal of considering the state is to analyze the impact when this state has changed. Therefore, we are also interested in actions that will change the global states. These operations can be represented as a series of elementary model manipulations which in turn are inversions of elementary model accesses. Hence, we will be interested not only in elementary model manipulations but also in their inversions. A similar approach was taken by Foster et al. with their Lenses framework [64], later extended by Diskin to Delta-Lenses [56].

To account for multiple objects having an interrelated state as, e.g., through opposite references, we model this state as a global state Ω on which we do not make any assumption. This is inspired by the universe Ω commonly used in stochastics. The intuition is that attributes of an object can change over time, just like random variables in stochastics can change over multiple states in the state space.

The reason for a very rough model of a global state is that an elementary model change may change the state not only of the model element that is worked with but also many others. An example here are opposite references where setting a reference of one model element implicitly also sets the opposite reference at another model element. Furthermore, a global state space enables a unified consideration of changes regardless of where the change originates.

The state space Ω can be seen as the space of possible memory states where we abstract from temporary data needed only to compute a given method. Thus, Ω can be thought of the set of sequences over an alphabet (e.g., $\{0, 1\}$) with finitely many non-zero entries. In particular, an element of the function set $\Omega \to A$ simply is a typed pointer, very similar to a typed random variable.

One of the merits of category theory is that it often does not require an in-depth understanding of the inner structure of objects but rather reasons on their behavior, i.e. the value or the uniqueness of certain morphisms. This is useful for us, because it enables a formalization at a very high level of abstraction that yields a good flexibility for a later implementation. In particular, we do not make any assumptions on the structure of the state space Ω or a given type A except that there is a relation that checks whether a given object has a certain type. We use the element notation $a \in A$ to depict that an object a is an instance of A. Further, we identify A with the set of its instances.

To take the global state into account, the basic idea is to extend a static type system (which can be thought of as a category $\mathcal{T}$ through the mapping defined by Crole [49]) with this global state space. The resulting category has as objects the canonical product of objects of $ob\,\mathcal{T}$ (i.e., types) and the global state space Ω.

We are particularly interested in the incrementalization of side-effect free morphisms as per the following definition:

Definition 19 (Side-effect free methods)**.** The idea of the definition of side-effect free methods is that they do not change the global state. In particular, a function $f : A \times \Omega \to B \times \Omega$ is side-effect free if and only if for all $(a, \omega) \in A \times \Omega$, it holds that

$$\pi_\Omega(f(a, \omega)) = \omega$$

where π_Ω is the canonical projection to the state of the result[24]. Since side-effect free morphisms do not change the global state, we sometimes identify the result of the result of a side-effect free function with the resulting object and discard the state as the latter did not change.

Example 17. For any type A, the identity on $A \times \Omega$ is side-effect free.

Proposition 4. A composition of side-effect free morphisms is side-effect free.

[24] It is common to index projections with indices. However, in the scope of this thesis, projections will be indexed with the space they are projecting to, as there is no case of confusion.

Definition 20 (Mutable Type Category)**.** A Mutable Type Category (MTC) C for a set of types $ob\,\mathcal{T}$ and a state space Ω is a category that consists of tuples $ob\ C := \{A \times \Omega \mid A \in ob\,\mathcal{T}\}$ as objects and morphisms $Mor(A \times \Omega, B \times \Omega)$ between two types A and B as functions $A \times \Omega \to B \times \Omega$. We further demand that the restriction of C to side-effect free morphisms C_Ω forms a cartesian-closed category.

Remark 7. If C is a category, then the restriction C_Ω with $ob\,C = ob\,C_\Omega$ and $Mor_{C_\Omega}(A \times \Omega, B \times \Omega) = \{f \in Mor_C(A \times \Omega, B \times \Omega) | f \text{is side-effect free}\}$ is a category in any case because the composition of side-effect free morphisms is side-effect free. Demanding that C_Ω is cartesian-closed means that products and sums exist for which only side-effect free methods must be taken into account.

The reason for this is that for methods with side-effects, the order in which they are executed matters which makes it hard to reason on product and sum types. If we restrict the methods to side-effect free methods, we can simply assume that the product of objects $A \times \Omega$ and $B \times \Omega$ in C_Ω is $A \times B \times \Omega$, because the global state is not touched in C_Ω. Meanwhile, we require the surrounding type system C in order to perform changes of the global state.

Definition 21 (Notation)**.** In the remainder of the thesis, we use a slightly simplified notation where we write $f : A \to B$ for $f \in Mor(A \times \Omega, B \times \Omega)$ when it is clear from context that A and B are types. We also say that $A \in C$ to denote that $A \times \Omega \in ob\ C$.

Further, a functor $\mathcal{F}$ applied to a given object $A \times \Omega$ in C must be an object $\mathcal{F}(A \times \Omega) = A' \times \Omega$. We notate this type A' as $A' = \mathcal{F}(A)$ such that $\mathcal{F}(A \times \Omega) = \mathcal{F}(A) \times \Omega$. We refer to changes of the global state as set-theoretic functions $\Delta\omega \in \Delta\Omega := \Omega \to \Omega$.

If we know that a morphism $f : A \to B$ is side-effect free, we often treat it as a function $A \times \Omega \to B$, because it is clear that the state will not change.

Definition 22. In a mutable type category C, a state change $\Delta\omega$ can be extended to a transformation $C \to C$

$$\Delta\omega_A : A \to A, (a, \omega) \mapsto (a, \Delta\omega(\omega))$$

that simply applies this state change but leaves the value intact. In the remainder we use this inclusion if this is clear from the context.

Definition 23 (In-model Lens)**.** In this interpretation, a (well-behaved) in-model lens (m-lens) $l : A \hookrightarrow B$ between types A and B of a mutable type category C consists of a side-effect free GET morphism $l \nearrow: A \to B$ and a partial morphism $l \searrow: A \times B \to A$ called the PUT function that satisfy the following conditions:

- For each $a \in A, \omega \in \Omega$, the morphism $l \searrow$ is defined for the tuple $(a, l \nearrow (a, \omega))$ and we have that

$$l \searrow (a, l \nearrow (a)) = (a, \omega) \qquad \text{(GetPut)}$$

- If $l \searrow$ is defined for a tuple $(a, b, \omega) \in A \times B \times \Omega$, then we have that

$$l \nearrow (l \searrow (a, b, \omega)) = (b, \tilde{\omega}) \quad \text{for some } \tilde{\omega} \in \Omega. \qquad \text{(PutGet)}$$

The first condition is a direct translation of the original PutGet law. Meanwhile, the second line is slightly weaker than the original GetPut because the global state may have changed. In particular, we allow the PUT function to change the global state.

Example 18. We want to show how references and attributes such as defined in MOF translate to Mutable Type Category (MTC) morphisms. Let $\mathit{entry} : \texttt{Route} \hookrightarrow \texttt{Semaphore}$ be the reference of `Route` that specifies a routes entry semaphore and likewise $\mathit{signal} : \texttt{Semaphore} \hookrightarrow \texttt{Signal}$ the attribute containing the current signal of the semaphore.

The corresponding excerpt from the metamodel is depicted in the class diagram of Figure 3.1.

The setter morphism $\mathit{signal} \searrow$ has full domain, i.e.

$$\mathcal{D}(\mathit{signal} \searrow) = \texttt{Semaphore} \times \texttt{Signal} \times \Omega.$$

This means that all possible instances of `Signal` are valid for a given semaphore.

However, we would typically want to restrict the semaphores that can be used as entries of a route to those semaphores that are in the same railway container.

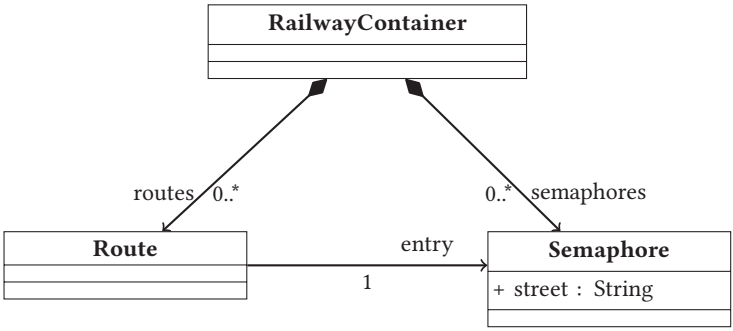


Figure 3.1.: The metamodel excerpt of the railway network metamodel used in Example 18

We can formulate this constraint formally by limiting the domain of the $entry \searrow$ morphism:

$$\mathcal{D}(signal \searrow) = \{(r, s, \omega) \in \texttt{Route} \times \texttt{Semaphore} \times \Omega \\ | s \in semaphores \nearrow (parent(r, \omega))\}.$$

Here, we used a *parent* morphism that simply returns the parent of a given route in the containment hierarchy. Such a morphism is usually generated by the modeling framework.

Remark 8. Whether and to which degree the domain of a lens PUT morphism is enforced by the type system is not clear. From a very technical point of view, the morphism may still be defined but an implementation may also throw an exception in such a case.

Remark 9. In general, setters may do more than just setting the value of their getter functions but change the global status entirely. The rationale is that setters may also influence other morphisms, especially the backward references. In the case of Example 18, assume that a `Semaphore` also holds the backwards reference to the `Route` instances that start at this semaphore. Then, changing the entry of a route should remove the route from the list of routes starting at the old entry and add the route to the list of routes starting at the new entry.

3.2. Stateless Types and Stateless Methods

Our formalization of MTCs specifically targets to understand the impact of changes. Thus, for a given morphism $f : A \to B$, we want to understand which state transitions $\Delta\omega \in \Delta\Omega$ cause the method return value $f(a, \Delta\omega(\omega))$ for a given $a \in A$ to change, i.e. be different than $f(a, \omega)$. For this, a very important kind of methods are stateless methods that we want to study in this section.

Definition 24 (Stateless Morphism)**.** Let $\mathcal{C}$ be a MTC, A and B objects of $\mathcal{C}$ and $f : A \to B$. Then f is stateless if and only if it is side-effect free and for every $a \in A$ and every $\omega_1, \omega_2 \in \Omega$ we have that

$$\pi_B(f(a, \omega_1)) = \pi_B(f(a, \omega_2)).$$

Example 19. For each object A of $\mathcal{C}$, the identity id_A on A is stateless since for any $a \in A$ and $\omega_1, \omega_2 \in \Omega$ we have that

$$\pi_A(id_A(a, \omega_1)) = a = \pi_A(id_A(a, \omega_2)).$$

Example 20. Let $A \times B$ be a tuple type. Then the projection morphisms $\pi_A : A \times B \to A$ and $\pi_B : A \times B \to B$ are stateless as extensions of the projections in $\mathcal{S}$.

Example 21. For numbers and boolean types, arithmetic operators $+, -, *, /, \%$ are stateless, same as equality and inequality and comparison operators. For more complex types such as collections, concatenations can be implemented stateless by just saving references to the concatenated lists instead of copying elements. While in the latter, updates to the original lists are lost (which is why the state matters), they are kept if the lists are only referenced.

Example 22. For any side-effect free morphism $f : A \times B \to C$, its exponential mate $\lambda f : A \to C^B$ is stateless because the function f is only evaluated in through the *eval* morphism.

Proposition 5. The composition of stateless morphisms is stateless.

Proof. Let $\mathcal{C}$ be a MTC type system and $f : A \to B$ and $g : B \to C$ stateless morphisms, $a \in A$ and $\omega_1, \omega_2 \in \Omega$, then we have that

$$\begin{aligned}(g \circ f)(a, \omega_1) &= g(f(a, \omega_1)) \\ &= g(\pi_B(f(a, \omega_1)), \omega_2) \\ &= g(\pi_B(f(a, \omega_2)), \omega_2) \\ &= g(f(a, \omega_2)) \\ &= (g \circ f)(a, \omega_2).\end{aligned}$$

□

3.3. Inheritance

Object-oriented design has proved its usefulness in many applications throughout the past decades. An important concept in the object-oriented world is the concept of inheritance between classes as a tool to specify generalizations and specializations. Inheritance has many facets and many semantic issues are related to it, suggesting when and how to use inheritance.

From a runtime perspective, this discussion is not necessary since the decision whether or not a given type inherits from another is a design question that already has an answer. In particular, the difference between inheritance and true subtyping is meaningless at runtime. If a type A inherits from a different type B, this implies that instances of A are also instances of type B and morphisms of B to another type C are also morphisms of A. We formalize this understanding of inheritance in the following definition.

Definition 25. Inheritance is a partial order relation $\preceq$ on the types of a MTC $\mathcal{C}$ such that forms a join-semilattice on the types of $\mathcal{C}$. Further, $A \preceq B$ for types A and B implies that all instances of A are also instances of B.

Remark 10. The inheritance relation $\preceq$ may not exactly match the intuitive understanding of 'A inherits B' but rather its reflexive, transitive closure. Definition 25 allows multiple inheritance that is forbidden in Java and .NET for classes. Therefore, model classes are usually implemented in interfaces.

3.4. Collections

Attributes or references in type systems often may have more than a single value. Such references are often called multi-valued. There are multiple kinds of multi-valued attributes in order to describe different behavior. Some of them already have been formalized. The most important behaviors are that all elements of a collection must be unique or that all elements of a collection have a defined order. This section aims to give these multi-valued attributes a formal semantics.

Proposition 6. Let C be a MTC. Then the (lazy) powerset functor $\mathcal{P}$ defines an endofunctor on C_Ω as follows: For any type A, $\mathcal{P}$ maps this object to pointers to sets of elements of A, i.e. $\mathcal{P}(A) = \Omega \to \mathbb{P}(A)$. We also demand that these subsets are finite, i.e. for each $s \in \mathcal{P}(A)$ and $\omega \in \Omega$, $|s(\omega)| < \infty$. Further, a morphism $f : A \to B$ is mapped through specification below.

$$\mathcal{P}(f) : \mathcal{P}(A) \to \mathcal{P}(B), \quad (S, \tilde{\omega}) \mapsto ((\omega \mapsto \{f(a, \omega) | a \in S(\omega)\}), \tilde{\omega}).$$

Here, the resulting mapping is clearly finite for each $\omega \in \Omega$. In particular, $\mathcal{P}(f)$ is stateless, regardless of whether f originally was stateless or not.

Proof. To show that $\mathcal{P}$ defines an endofunctor on C_Ω, we need to show that $\mathcal{P}(id_A) = id_{\mathcal{P}(A)}$ and for any side-effect free morphisms $f : A \to B, g : B \to C$ that $\mathcal{P}(g \circ f) = \mathcal{P}(g) \circ \mathcal{P}(f)$.

Thus, let $S \in \mathcal{P}(A)$ and $\tilde{\omega} \in \Omega$. We then have that

$$\mathcal{P}(id_A)(S, \tilde{\omega}) = \omega \mapsto \{id_A(a, \omega) | a \in S(\omega)\} = \omega \mapsto S(\omega) = id_{\mathcal{P}(A)}(S, \tilde{\omega})$$

and furthermore

$$\begin{aligned}
\mathcal{P}(g \circ f)(S, \tilde{\omega})(\omega) &= \{(g \circ f)(a, \omega) | a \in S(\omega)\} \\
&= \{g(f(a, \omega)) | a \in S(\omega)\} \\
&= \{g(b, \omega) | b \in \{f(a, \omega) | a \in S(\omega)\}\} \\
&= \{g(b, \omega) | b \in \mathcal{P}(f)(S, \tilde{\omega})(\omega)\} \\
&= (\mathcal{P}(g) \circ \mathcal{P}(f))(S, \tilde{\omega})(\omega).
\end{aligned}$$

□

Remark 11. The fact that $\mathcal{P}$ is restricted to C_Ω instead of being applicable to the entire C means that we may only use side-effect free morphisms as inputs to the monad. This has the reason that otherwise, it would not be clear in which order the state changes would have to be evaluated. We will restrict the other collection types in the same way.

Proposition 7. In the above situation, the endofunctor $\mathcal{P}$ defines a monad with the following transformations for a given object A of C_Ω:

$$\eta^{\mathcal{P}}_A : A \to \mathcal{P}(A), a \mapsto (\omega \mapsto \{a\})$$
$$\mu^{\mathcal{P}}_A : \mathcal{P}^2(A) \to \mathcal{P}(A), \wp \mapsto (\omega \mapsto \bigcup_{S \in \wp} S(\omega)).$$

Proof. We need to show that the transformations $\eta^{\mathcal{P}}$ and $\mu^{\mathcal{P}}$ are natural and fulfill the requirements for a monad. Therefore, let $f : A \to B$ be a side-effect free morphism, $a \in A$, $\wp \in \mathcal{P}^2(A)$ and $\tilde{\omega} \in \Omega$. We then have that

$$\begin{aligned}(\eta^{\mathcal{P}}_B \circ f)(a, \tilde{\omega}) &= \omega \mapsto \{f(a, \tilde{\omega})\} \\ &= \mathcal{P}(f)(\omega \mapsto \{a\}, \tilde{\omega}) \\ &= (\mathbb{P}(f) \circ \eta^{\mathbb{P}}_A)(a, \tilde{\omega}).\end{aligned}$$

Furthermore,

$$\begin{aligned}(\mu^{\mathcal{P}}_B \circ \mathcal{P}^2(f))(\wp, \tilde{\omega}) &= \omega \mapsto \bigcup_{S \in \wp(\omega)} \{f(a, \omega) | a \in S(\omega)\} \\ &= \omega \mapsto \{f(a, \omega) | a \in \bigcup_{S \in \wp(\omega)} S(\omega)\} \\ &= (\mathcal{P}(f) \circ \mu^{\mathcal{P}}_A)(\wp, \omega).\end{aligned}$$

This shows that $\eta^{\mathcal{P}}$ and $\mu^{\mathcal{P}}$ are natural.

Now, similar to Example 15, let $X \in \mathcal{P}^3(A)$ be a subset of subsets of subsets of A. Then

$$(\mu^{\mathcal{P}} \circ \mu^{\mathcal{P}}\mathcal{P})_A(X) = \omega \mapsto \bigcup_{\wp \in X(\omega)} \bigcup_{S \in \wp(\omega)} S(\omega) = (\mu^{\mathcal{P}} \circ \mathcal{P}\mu^{\mathcal{P}})_A(X)$$

and for each $S \in \mathcal{P}(A)$

$$(\mu^{\mathcal{P}} \circ \eta^{\mathcal{P}}\mathcal{P})_A(S) = (\mu^{\mathcal{P}} \circ \mathcal{P}\eta^{\mathcal{P}})_A(S) = \omega \mapsto \bigcup_{a \in S(\omega)} \{a\} = S.$$

□

Definition 26. Let $f : A \to B$ be a side-effect free morphism in C. Then the inverse f^{-1} is a mapping $f^{-1} : \mathbb{P}(B) \to \mathbb{P}(A \times \Omega)$ such that we have for each $S \in \mathbb{P}(B)$ that $(a, \omega) \in f^{-1}(S) :\Leftrightarrow f(a, \omega) \in S$.

The existence of this mapping is guaranteed by basic set theory. The purpose of this definition here is only to be clear about the notation. We will need this inverse of morphisms for non-unique multi-valued morphisms.

Proposition 8. Let C be a MTC. Then the multi-powerset $\mathcal{M}$ defines an endofunctor on C_Ω through the following mapping for each $A \in ob\, C$:

$$\mathcal{M}(A) = \Omega \to (A \to \mathbb{N}_0).$$

Here, we also demand that the domain of a multiset is finite for every $\omega \in \Omega$, i.e. $|m(\omega)^{-1}(\mathbb{N})| < \infty$ for any $\omega \in \Omega$[25].

That is, $\mathcal{M}(A)$ is a function returning a multiset with elements of A for every state of the state space Ω. The elements of $m \in \mathcal{M}(A)$ for a given state $\omega \in \Omega$ is given by $m(\omega)^{-1}(\mathbb{N})$ and for each element this set, the multiplicity is given by $m(\omega)(a)$. We will shorten that an element a is contained in the multi-set through the notation $a \in m(\omega) :\Leftrightarrow m(\omega)(a) > 0$.

Let $f : A \to B$ be a morphism in C, then $\mathcal{M}(f)$ is given by the following definition:

$$\begin{aligned}\mathcal{M}(f) :&\mathcal{M}(A) \to \mathcal{M}(B),\\ &(m, \tilde{\omega}) \mapsto ((\omega \mapsto (b \mapsto \sum_{(a,\omega) \in f^{-1}(\{b\}) \cap A \times \{\omega\}} m(\omega)(a))), \tilde{\omega})\end{aligned}$$

with $\mathcal{D}(\mathcal{M}(f)(m, \tilde{\omega})(\omega)) = f(\mathcal{D}(m(\omega)), \omega)$.

In particular, $\mathcal{M}(f)$ is stateless regardless whether f was stateless.

[25] We denote $\mathbb{N}$ as the set of natural numbers starting with 1.

Proof. To show that $\mathcal{M}$ is a functor, we also need to show that $\mathcal{M}(id_A) = id_{\mathcal{M}(A)}$ for any object A in $\mathcal{C}_\Omega$ and for every side-effect free morphisms $f : A \to B, g : B \to C$ we have that $\mathcal{M}(g \circ f) = \mathcal{M}(g) \circ \mathcal{M}(f)$.

Thus, let $m \in \mathcal{M}(A)$ and $\omega, \tilde{\omega} \in \Omega$. We then have that

$$\begin{aligned}
\mathcal{M}(id_A)(m,\tilde{\omega})(\omega) &= b \mapsto \sum_{(a,\omega) \in id^{-1}(\{b\}) \cap A \times \{\omega\}} m(\omega)(a) \\
&= b \mapsto \sum_{(a,\omega) \in \{b\} \times \Omega \cap A \times \{\omega\}} m(\omega)(a) \\
&= b \mapsto m(\omega)(b) \\
&= m(\omega) \\
&= id_{\mathcal{M}(A)}(m,\tilde{\omega})(\omega).
\end{aligned}$$

Furthermore,

$$\mathcal{M}(g \circ f)(m,\tilde{\omega})(\omega) = c \mapsto \sum_{(a,\omega) \in (g \circ f)^{-1}(\{c\}) \cap A \times \{\omega\}} m(\omega)(a).$$

Here, we have that for every $c \in C$ that

$$\begin{aligned}
(a,\omega) \in (g \circ f)^{-1}(\{c\}) \cap A \times \{\omega\} &\Leftrightarrow g(f(a,\omega),\omega) = c \\
&\Leftrightarrow (f(a,\omega),\omega) \in g^{-1}(\{c\}) \cap B \times \{\omega\}.
\end{aligned}$$

For each $(b,\omega) \in g^{-1}(\{c\}) \cap B \times \{\omega\}$, the origin pairs (a,ω) for which $b = f(a,\omega)$ are precisely given by $f^{-1}(\{b\}) \cap A \times \{\omega\}$.

Hence, for any $m \in \mathcal{M}(A), c \in C$ and a given pair $(a,\omega) \in (g \circ f)^{-1}(\{b\})$ we conclude

$$\begin{aligned}
\mathcal{M}(g \circ f)(m,\tilde{\omega})(\omega)(c) &= \sum_{(b,\omega) \in g^{-1}(\{c\}) \cap B \times \{\omega\}} \; \sum_{(a,\omega) \in f^{-1}(\{b\}) \cap A \times \{\omega\}} m(\omega)(a) \\
&= \sum_{(b,\omega) \in g^{-1}(\{c\}) \cap B \times \{\omega\}} \mathcal{M}(f)(m,\tilde{\omega})(\omega)(b) \\
&= \mathcal{M}(g)(\mathcal{M}(f)(m,\tilde{\omega}),\tilde{\omega})(\omega)(c).
\end{aligned}$$

This shows that $\mathcal{M}(g \circ f) = \mathcal{M}(g) \circ \mathcal{M}(f)$. □

Proposition 9. In the situation of Proposition 8, the functor $\mathcal{M}$ defines a monad on C_Ω equipped with the following natural transformations:

$$\eta_A^{\mathcal{M}} : A \to \mathcal{M}(A), a \mapsto (\omega \mapsto (b \mapsto \delta(a,b) := \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases}))$$
$$\mu_A^{\mathcal{M}} : \mathcal{M}^2(A) \to \mathcal{M}(A), M \mapsto (\omega \mapsto (b \mapsto \sum_{m \in \mathcal{M}(A)} M(\omega)(m) \cdot m(\omega)(b))).$$

Proof. We first show that indeed, $\eta^{\mathcal{M}}$ and $\mu^{\mathcal{M}}$ define natural transformations. Therefore, let $f : A \to B$ a side-effect free morphism. We need to show that $\eta_B^{\mathcal{M}} \circ f = \mathcal{M}(f) \circ \eta_A^{\mathcal{M}}$ and $\mu_B^{\mathcal{M}} \circ \mathcal{M}^2(f) = \mathcal{M}(f) \circ \mu_A^{\mathcal{M}}$.

Thus, first let $a \in A, b \in B$ and $\omega, \tilde{\omega} \in \Omega$. We then have

$$\begin{aligned} \eta_B^{\mathcal{M}}(f(a,\tilde{\omega}))(\omega)(b) &= \delta(f(a,\omega),b) \\ &= \sum_{(\tilde{a},\omega) \in f^{-1}(\{b\}) \cap A \times \{\omega\}} \delta(a,\tilde{a}) \\ &= \mathcal{M}(f)(\eta_A^{\mathcal{M}}(a),\tilde{\omega})(\omega)(b). \end{aligned}$$

Furthermore, for $M \in \mathcal{M}^2(A), b \in B$ and $\omega, \tilde{\omega}$ we have

$$\begin{aligned} &\mu_B^{\mathcal{M}}(\mathcal{M}^2(f)(M,\tilde{\omega}))(\omega)(b) \\ &\quad = \sum_{m \in \mathcal{M}(B)} \mathcal{M}^2(f)(M,\tilde{\omega})(\omega)(m) \cdot m(\omega)(b) \\ &\quad = \sum_{m \in \mathcal{M}(B)} \sum_{(\tilde{m},\omega) \in \mathcal{M}(f)^{-1}(\{m\}) \cap \mathcal{M}(A) \times \{\omega\}} M(\omega)(\tilde{m}) \cdot m(\omega)(b). \end{aligned}$$

For such a multi-set $\tilde{m} \in \mathcal{M}(A)$, it holds that

$$(\tilde{m},\omega) \in \mathcal{M}(f)^{-1}(\{m\}) \cap \mathcal{M}(A) \times \{\omega\} \Leftrightarrow \mathcal{M}(f)(\tilde{m},\omega) = m.$$

Furthermore, the sets $\{m\}$ are pairwise disjunct for $m \in \mathcal{M}(B)$ and thus

$$\bigcup_{m \in \mathcal{M}(B)} \mathcal{M}(f)^{-1}(\{m\}) = \mathcal{M}(f)^{-1}(\mathcal{M}(B)) = \mathcal{M}(A) \times \Omega.$$

As a consequence,

$$\begin{aligned}
&\mu_B^{\mathcal{M}}(\mathcal{M}^2(f)(M,\tilde{\omega}))(\omega)(b)\\
&\quad= \sum_{m\in\mathcal{M}(B)} \sum_{(\tilde{m},\omega)\in\mathcal{M}(f)^{-1}(\{m\})\cap\mathcal{M}(A)\times\{\omega\}} M(\omega)(\tilde{m})\cdot m(\omega)(b)\\
&\quad= \sum_{\tilde{m}\in\mathcal{M}(A)} \sum_{(a,\omega)\in f^{-1}(\{b\})\cap A\times\{\omega\}} M(\omega)(\tilde{m})\cdot \tilde{m}(\omega)(a)\\
&\quad= \sum_{(a,\omega)\in f^{-1}(\{b\})\cap A\times\{\omega\}} \sum_{\tilde{m}\in\mathcal{M}(A)} M(\omega)(\tilde{m})\cdot \tilde{m}(\omega)(a)\\
&\quad= \mathcal{M}(f)(\mu_A^{\mathcal{M}}(M),\tilde{\omega})(\omega)(b).
\end{aligned}$$

We can exchange the summation since for any $M \in \mathcal{M}^2(A)$, the term $M(\omega)(\tilde{m})$ is only non-zero for finitely many elements because M only contains finitely many elements in any global state ω. Likewise, $\tilde{m}$ only contains finitely many elements and thus the summation is finite and can be exchanged.

This concludes the naturality of $\mu^{\mathcal{M}}$.

To show that the equations $\mu^{\mathcal{M}} \circ \mathcal{M}\eta^{\mathcal{M}} = \mu^{\mathcal{M}} \circ \eta^{\mathcal{M}}\mathcal{M} = id_{\mathcal{M}}$ and $\mu^{\mathcal{M}} \circ \mathcal{M}\mu^{\mathcal{M}} = \mu^{\mathcal{M}} \circ \mu^{\mathcal{M}}\mathcal{M}$, let $m \in \mathcal{M}(A)$ and $X \in \mathcal{M}^3(A)$. Then we have for $a \in A$ and $\omega \in \Omega$

$$\begin{aligned}
(\mu_A^{\mathcal{M}} \circ \mathcal{M}\eta_A^{\mathcal{M}})(m)(\omega)(a) &= \sum_{\tilde{m}\in\mathcal{M}(A)} \delta(m,\tilde{m})\cdot m(\omega)(a)\\
&= m(\omega)(a)\\
&= \sum_{\tilde{m}\in\mathcal{M}(A)}\\
&= (\mu_A^{\mathcal{M}} \circ \eta_A^{\mathcal{M}}\mathcal{M})(m)(\omega)(a)
\end{aligned}$$

and

$$\begin{aligned}
(\mu_A^{\mathcal{M}} \circ \mathcal{M}\mu_A^{\mathcal{M}})(X)(\omega)(a) &= \sum_{M\in\mathcal{M}^2(A)} \sum_{m\in\mathcal{M}(A)} X(\omega)(M)\cdot M(\omega)(m)\cdot m(\omega)(a)\\
&= (\mu_A^{\mathcal{M}} \circ \mu_A^{\mathcal{M}}\mathcal{M})(X)(\omega)(a).
\end{aligned}$$

This concludes that $\mathcal{M}$ is a monad. □

Proposition 10. Let C be a MTC. Then the Kleene closure $\mathcal{K}$ defines an endofunctor on C_Ω where for each object $A \in C_\Omega$ and each side-effect free morphism $f : A \to B$ we have that

$$\mathcal{K}(A) = \Omega \to A*$$

and

$$\begin{aligned}&\mathcal{K}(f) : \mathcal{K}(A) \to \mathcal{K}(B),\\ &\qquad (\omega \mapsto (a_1; \ldots; a_n), \tilde{\omega}) \mapsto (\omega \mapsto (f(a_1, \omega); \ldots; f(a_n, \omega))).\end{aligned}$$

In particular, again $\mathcal{K}(f)$ is stateless regardless whether f was stateless.

Proof. Let $a \in \mathcal{K}(A)$ and $\omega, \tilde{\omega} \in \Omega$. Then we have that for the sequence $a(\omega) = (a_1, \ldots, a_n)$ that

$$\mathcal{K}(id_A)(a, \tilde{\omega})(\omega) = (id_A(a_1, \omega); \ldots; id_A(a_n, \omega)) = (a_1; \ldots; a_n).$$

Furthermore, for morphisms $f : A \to B$ and $g : B \to C$:

$$\begin{aligned}\mathcal{K}(g \circ f)(a, \tilde{\omega})(\omega) &= (g(f(a_1, \omega), \omega), \ldots, g(f(a_n, \omega), \omega))\\ &= \mathcal{K}(g)(\hat{\omega} \mapsto (f(a_1, \hat{\omega}), \ldots, f(a_n, \hat{\omega})), \tilde{\omega})(\omega)\\ &= \mathcal{K}(g)(\mathcal{K}(f)(a, \tilde{\omega}), \tilde{\omega})(\omega)\\ &= (\mathcal{K}(g) \circ \mathcal{K}(f))(a, \tilde{\omega})(\omega).\end{aligned}$$

This concludes the proof. □

Proposition 11. In the situation of Proposition 10, $\mathcal{K}$ can be extended to a monad on C_Ω through the following transformations for a given object A of C_Ω:

$$\begin{aligned}&\eta^{\mathcal{K}}_A : A \to \mathcal{K}(A), a \mapsto (\omega \mapsto (a))\\ &\mu^{\mathcal{K}}_A : \mathcal{K}^2(A) \to \mathcal{K}(A),\\ &\qquad a \mapsto (\omega \mapsto a^{(1)}_1; \ldots; a^{(1)}_{m_1}; a^{(2)}_1; \ldots; a^{(n-1)}_{m_{n-1}}; a^{(n)}_1; \ldots; a^{(n)}_{m_n})\end{aligned}$$

where in the latter equation $a(\omega) = (a^{(1)}; \ldots; a^{(n)})$ and similarly $a^{(i)}(\omega) = (a_1^{(i)}; \ldots; a_{m_i}^{(i)})$.

Proof. We omit the proof here since it is rather uninteresting and technical. □

Remark 12. The fact that all of the collection monads presented lift morphisms to stateless morphisms in the monad is utilized by query frameworks such as provided by the SQOs. It means that the state in which a query was defined has no influence on the query result but only the state when evaluating the query has.

Proposition 12. The transformation $\imath^{\mathcal{K}} : \mathcal{K} \to \mathcal{M}$ with the components

$$\begin{aligned} &\imath_A^{\mathcal{K}} : \mathcal{K}(A) \to \mathcal{M}(A), \\ &\quad (\omega \mapsto (a_1; \ldots; a_n)) \mapsto (\omega \mapsto (a \mapsto |\{i \in \{1, \ldots, n\} | a_i = a\}|)) \end{aligned}$$

is natural.

Proof. Let $f : A \to B$ be a morphism. We need to show that $\imath_B^{\mathcal{K}} \circ \mathcal{K}(f) = \mathbb{P}(f) \circ \imath_A^{\mathcal{K}}$. For this, fix $\omega \in \Omega$ and let $a \in \mathcal{K}(A)$ be a list of elements in A such that $a(\omega) = (a_1, \ldots, a_n)$. We then have that $b \in B$

$$\begin{aligned} (\imath_B^{\mathcal{K}} \circ \mathcal{K}(f))(a, \tilde{\omega})(\omega)(b) &= |\{i \in \{1, \ldots, n\} | f(a_i, \omega) = b\}| \\ &= |\{i \in \{1, \ldots, n\} | (a_i, \omega) \in f^{-1}(\{b\}) \cap A \times \{\omega\}\}| \\ &= \sum_{(\tilde{a}, \omega) \in f^{-1}(\{b\}) \cap A \times \{\omega\}} |\{i \in \{1, \ldots, n\} | a_i = \tilde{a}\}| \\ &= (\mathcal{M}(f) \circ \imath_A^{\mathcal{K}})(a, \tilde{\omega})(\omega)(b). \end{aligned}$$

This shows the naturality of $\imath^{\mathcal{K}}$. □

Proposition 13. The transformation $\imath^{\mathcal{P}} : \mathcal{P} \to \mathcal{M}$ with the components

$$\imath^{\mathcal{P}}_A : \mathcal{P}(A) \to \mathcal{M}(A), A \mapsto (\omega \mapsto \left(a \mapsto \left\{ \begin{array}{cc} 1 & \text{if } a \in A(\omega) \\ 0 & \text{otherwise} \end{array} \right\}\right))$$

is **not** natural.

Proof. Informally, the reason that $\jmath^{\mathcal{P}}$ is not natural is that it makes a difference when to apply $\imath^{\mathcal{P}}$ if a morphism f is not injective for a given $\omega \in \Omega$. Let for example $a_1, a_2 \in A$ such that for a given $\omega \in \Omega$ we have that $f(a_1, \omega) = f(a_2, \omega) = b$ for some $b \in B$. If we now consider the constant list $\mathcal{K}(A) \ni a := \tilde{\omega} \mapsto (a_1, a_2)$, then we have that

$$\imath^{\mathcal{P}}_A(a)(\omega) = \tilde{a} \mapsto \left\{ \begin{array}{cc} 1 & \tilde{a} \in \{a_1, a_2\} \\ 0 & \text{otherwise} \end{array} \right\}$$

and consequently

$$\mathcal{M}(f)(\imath^{\mathcal{P}}_A(a), \tilde{\omega})(\omega)(b) = \sum_{(\tilde{a},\omega) \in f^{-1}(\{b\}) \cap A \times \{\omega\}} \left\{ \begin{array}{cc} 1 & \tilde{a} \in \{a_1, a_2\} \\ 0 & \text{otherwise} \end{array} \right\} = 2.$$

On the other hand, we have that $\imath^{\mathcal{P}}_B(\mathcal{P}(f)(a, \tilde{\omega}))(\omega)(\tilde{b}) \leq 1$ for any $\tilde{b} \in B$ due to the construction of $\imath^{\mathcal{P}}$. □

Remark 13. The above proposition inevitably raises the question what the consequence of this negative result is, especially in comparison to Proposition 12. The consequence is that developers must always be clear whether they are operating on $\mathcal{M}$ or $\mathcal{P}$ as the results may differ, not only in terms of performance (as deduplication usually is an expensive operation), but also concerning correct results. A possible solution can be to append a deduplication after each computation that is done on sets but we argue that this is easy to forget as the necessity seems unclear. Furthermore, such an additional deduplication degrades performance.

On the other hand, the monad $\mathcal{P}$ is not so important in practical applications, at least not the functor applications to morphisms.

3.5. Composition

Many type systems include a notion of composition. As an example, the UML distinguishes between associations, aggregations and compositions[26]. The main difference between these references is the relation of instances if they are connected through one or the other type of reference. If an object a is connected to an object b through an association, this has no influence on either a or b in the sense that both a and b can still be associated to any other instance. If however, $a \in A$ is connected to an instance $b \in B$ through a composition, this means that a contains the instance b. No other element $\tilde{a}$ may contain b through the same or any other composition reference. For this composition, the owner a is unique for every b, if it exists.

Definition 27 (Composition Hierarchy)**.** Let $\mathcal{C}$ be a MTC. A composition hierarchy $\boldsymbol{C}$ is a set of side-effect free morphisms that are called composition morphisms such that the following properties hold: For every composition morphisms $f : A \to B$ and $g : C \to B$, $a \in A, c \in C$ and $\omega \in \Omega$ such that $f(a, \omega) = g(c, \omega)$, either there exists a composition morphism $\tilde{f} : C \to A$ such that $g = f \circ \tilde{f}$ and $(a, \omega) = \tilde{f}(c, \omega)$ or a composition morphism $\tilde{g} : A \to C$ such that $f = g \circ \tilde{g}$ and $(c, \omega) = \tilde{g}(a, \omega)$. Furthermore, for each type A, the identity is a composition morphism.

Remark 14. Note that Definition 27 demands the implication only if f is evaluated in the same state for a and c. In particular, it is allowed to move model elements through the composition hierarchy.

Proposition 14. Let $f : A \to B, g : B \to C$ be composition morphism in the MTC $\mathcal{C}$. Then the morphism $g \circ f$ is also a composition morphism.

Proof. Let $h : D \to C$ be a composition from some type D such that for some $a \in A, d \in D$ and $\omega \in \Omega$ we have that $(g \circ f)(a, \omega) = h(d, \omega)$. Because $(g \circ f)(a, \omega) = g(f(a, \omega)$ and g is a composition, we can distinguish two cases:

[26] Even in the most recent version UML 2.5, the difference between these types is not entirely clear and we rather refer to the presumably most common usage.

1. There exists a composition morphism $\widetilde{gf} : D \to B$ such that $h = g \circ \widetilde{gf}$ and $f(a, \omega) = \widetilde{gf}(d, \omega)$. In this case, we can apply that f is also a composition morphism and again have two cases to consider:
   a) There exists a composition morphism $\tilde{f} : D \to A$ such that $\widetilde{gf} = f \circ \tilde{f}$ and $(a, \omega) = \tilde{f}(d, \omega)$. In this case, we see that
   $$h = g \circ \widetilde{gf} = g \circ f \circ \tilde{f} = (g \circ f) \circ \tilde{f}$$
   and further $(a, \omega) = \tilde{f}(d, \omega)$ as required.
   b) There exists a composition morphism $\tilde{f} : A \to D$ such that $f = \widetilde{gf} \circ \tilde{f}$ and $(a, \omega) = \tilde{f}(d, \omega)$. In this case, we see that
   $$g \circ f = g \circ \widetilde{gf} \circ \tilde{f} = h \circ \tilde{f}$$
   and further $(a, \omega) = \tilde{f}(d, \omega)$ as required.
2. There exists a composition morphism $\tilde{h} : B \to D$ such that $g = h \circ \tilde{h}$ and $(d, \omega) = \tilde{h}(f(a, \omega))$. In this case, we have that $g \circ f = h \circ (\tilde{h} \circ f)$ as required[27].

□

3.6. Opposites

EMOF supports the notion of opposite references, a feature widely used in the model-driven community as it has proved to be very useful[28]. It is also the reason for the very flexible approach we made in Section 3.1 for elementary morphism: The goal was simply to allow such morphisms as opposite references. In this section, we will formally define what opposite references are and present a few results.

[27] Strictly, we do not know that $\tilde{h} \circ f$ is a composition. However, in practice, most compositions are assemblies of finitely many atomic composition morphisms that are not the identity and we therefore apply the presented disassembly as many times as required.

[28] A reference and its opposite can also be viewed as the two ends of a bidirectional reference. However, this point of view is more difficult to capture formally.

Definition 28 (Opposites)**.** Let $f : A \to \mathcal{M}(B)$ and $g : B \to \mathcal{M}(A)$ be side-effect free morphisms in the MTC C. Then, f is the opposite of g, if for every $a \in A, b \in B$ and $\omega \in \Omega$ we have that

$$f(a,\omega)(\omega)(b) > 0 \Leftrightarrow g(b,\omega)(\omega)(a) > 0.$$

If $\tilde{f} : A \to B$ or $\tilde{g} : B \to A$, then these morphisms can be inserted into the above definition through the application of $\eta^{\mathcal{M}}$, i.e. $\tilde{f}$ is an opposite for $\tilde{g}$ if $\eta^{\mathcal{M}} \circ \tilde{f}$ is an opposite of $\eta^{\mathcal{M}} \circ \tilde{g}$.

Proposition 15. Let $f : A \to B$ and $g : B \to A$ morphisms of a MTC C. Then f is an opposite of g if and only if for every $a \in A, b \in B$ and $\omega \in \Omega$ we have that

$$f(a,\omega) = b \Leftrightarrow g(b,\omega) = a.$$

Proof. We have that for every $a \in A, b \in B$ and $\omega \in \Omega$ we have that

$$\begin{aligned}(\eta^{\mathcal{M}} f)(a,\omega)(\omega)(b) > 0 &\Leftrightarrow \delta(b, f(a,\omega)) > 0 \\ &\Leftrightarrow f(a,\omega) = b.\end{aligned}$$

This proposition also holds for g for symmetry reasons. Thus, we have the equivalence of the following four statements:

$$\begin{array}{ccc} f(a,\omega) = b & \Longleftrightarrow & g(b,\omega) = a \\ \Updownarrow & & \Updownarrow \\ (\eta^{\mathcal{M}} f)(a,\omega)(\omega)(b) > 0 & \Longleftrightarrow & (\eta^{\mathcal{M}} g)(b,\omega)(\omega)(a) > 0 \end{array}$$

This shows the claim. □

Proposition 16. Let C be a type system and A a type. Then the identity on A is an opposite of itself.

Proof. The claim follows straight from the definition. □

Proposition 17. Let $f : A \to B, g : B \to C, \tilde{f} : B \to A, \tilde{g} : C \to B$ be morphisms such that f is an opposite of $\tilde{f}$ and g is an opposite of $\tilde{g}$. Then also $g \circ f$ is an opposite of $\tilde{f} \circ \tilde{g}$.

Proof. We have that for $a \in A, b \in B$ and $\omega \in \Omega$ that

$$\begin{aligned}(g \circ f)(a, \omega) = b &\Leftrightarrow g(f(a, \omega)) = b \\ &\Leftrightarrow f(a, \omega) = \tilde{g}(b, \omega) \\ &\Leftrightarrow a = \tilde{f}(\tilde{g}(b, \omega), \omega) \\ &\Leftrightarrow a = (\tilde{f} \circ \tilde{g})(b, \omega).\end{aligned}$$

□

Part II.

Implicit Incremental Model Analyses

4. Efficient Incremental Computation Systems

The goal of this chapter is to let general-purpose incrementalization systems work on the same abstraction level the developer of a given analysis is using when designing a model analysis rather than on the instruction set of the programming language (C I.1). The rationale behind this goal is that model analyses nowadays often use library methods wherever possible to reduce cost and the probability of bugs by reusing modules that are usually well tested and potentially optimized for performance. As an example of such a frequently used library, we consider an implementation of query methods. To achieve this goal, this Chapter develops a formalism how incrementalization can be described in order to understand how it can be overridden. The latter concept is then implemented and applied to a query framework.

The remainder of this chapter is structured as follows: Section 4.1 introduces a formalization of incrementalization as functors from category theory. Section 4.2 discusses how to use insights from this formalization to integrate dynamic algorithms in incrementalization systems. Section 4.3 presents the implementation of these concepts in the incrementalization system NMF Expressions. Section 4.4 shows an example usage of the integration approach to integrate dynamic algorithms for the SQOs. Section 4.5 discusses how the approach used in this chapter can be used also to combine incrementality with distributed computing as another technique to overcome scalability problems of model analyses. Lastly, Section 4.6 summarizes the insights and achievements from this chapter.

4.1. Incrementalization as a Functor

In this section, the application of MTC in formalizing incremental computation systems is shown. The goal of a formalization for incremental computation systems given an analysis morphism $f : M \longrightarrow R$ is some object of a type $\mathcal{I}(R)$. This object will represent the running live analysis (cf. Section 2.2). From this object, we would like to query the current analysis result and apply any model changes, such as through natural transformations

$$value : \mathcal{I}(R) \longrightarrow R \text{ and } apply : \mathcal{I}(R) \times \Delta\Omega^{29} \longrightarrow \mathcal{I}(R).$$

In this situation, the *value* function is meant to return the current value of an incremental value instance of $\mathcal{I}(R)$ while *apply* applies a change of the global state to the incremental value. The idea is that this application could be used to propagate changes to the analysis result. The type $\mathcal{I}(R)$ is dependent on R to maintain type-safety while the system $\mathcal{I}$ is independent of the analysis result type.

As a trivial example, consider the check whether a semaphore is set to *GO* in the running example. Let `Signal` : `Semaphore` $\hookrightarrow$ `Signal` be the property access returning the current signal of a semaphore. Further we have the morphism $\neq$: `Object` $\times$ `Object` $\longrightarrow$ `bool` and the constant value `GO` can be extended to a constant morphism $GO : \top \longrightarrow$ `Signal` which simply returns the signal `GO` regardless of the state that is provided as a parameter. Thus, we can formulate the expression as

$$\begin{aligned} &\texttt{isGo} : \texttt{Semaphore} \longrightarrow \texttt{bool}, \\ &\qquad (s, \omega) \mapsto_{\neq} (Signal \nearrow (s, \omega), GO(\omega)). \end{aligned}$$

In implementations, such a representation can be easily retrieved from a typed abstract syntax tree.

With the help of mutable type categories, we can formalize incremental analyses using functors, i.e. we can simply formalize $\mathcal{I}$ from above as a functor from $\mathcal{C}_\Omega$ to itself[30]. The functor $\mathcal{I}$ then maps each type A in $\mathcal{C}$ to some $\mathcal{I}(A)$ in $\mathcal{C}$ for which we demand that natural transformations *value* :

[29] We will define the semantics of this construct in Proposition 18

[30] Thanks to assumption A1, we can restrict ourselves to side-effect free morphisms

$\mathcal{I} \to Id_C$ and $apply : \mathcal{I} \times \Delta\Omega \to \mathcal{I}$ exist. We also require $\mathcal{I}$ to respect products and exponentials, i.e. for any objects A, B in C_Ω, we have that $\mathcal{I}(A \times B) = \mathcal{I}(A) \times \mathcal{I}(B)$ and $\mathcal{I}(A \to B) = \mathcal{I}(A) \to \mathcal{I}(B)$, respectively. We can then apply $\mathcal{I}$ to our analysis f and yield a function $\mathcal{I}(f) : \mathcal{I}(M) \to \mathcal{I}(R)$. This function is then used to automatically update analysis results from a changed model underneath which we formalize as an instance of $\mathcal{I}(M)$.

Using the functor $\mathcal{I}$, we can simply apply it to our small sub-expression `isGo` to retrieve

$$\mathcal{I}(\texttt{isGo}) : \mathcal{I}(\texttt{Semaphore}) \to \mathcal{I}(\texttt{bool}).$$

The associativity of the functor guarantees us that we can assemble $\mathcal{I}(\texttt{isGo})$ from the functor applied to the components of `isGo`, i.e. its abstract syntax tree.

Moreover, the general approach of using functors to change the way how a given function is executed is independent of the exact structure of the type $\mathcal{I}(A)$ for a given type A. This provides several degrees of freedom for implementations.

Although functors already suffice to represent incremental execution systems, it is useful to consider monads. One reason for this is that in order to apply $\mathcal{I}(f)$ to an instance $m \in M$, one needs an instance $m' \in \mathcal{I}(M)$. Since the incrementalization system should be independent of the model type M, such a method should be available as a transformation $Id_C \to \mathcal{I}$. Semantically, an element of a given type can be regarded as an incremental value that simply never changes, i.e. as a constant. This definition matches the requirements for the unit transformation η of a monad.

For a given fixed model element, the value for a given property may change over time so that the property value can be understood as an incremental value. A useful thing one would like to achive is to also apply such a function to incremental values of the model element type and still retrieve an incremental value instead of an incremental value of an incremental value. Such a simplification can be offered by the μ transformation of a monad.

What remains to discuss is whether the naturality of η and μ is useful in this scenario. For η, this naturality means that we could either apply a function on a value and then regard the result as an incremental value (by regarding it as a constant) or lifting the input to the monad (by regarding it as a constant) and then run the incremental derivation of the function on it. This is clearly

not the case. Consider for example a property access as function. The value of the property may change over time (if the property is assigned a new value) whereas the constant value obtained by lifting the property access result once does not change. Thus, naturality is something that we explicitly do not want to have for η and we have to be very careful when to apply it.

This situation is different for μ, as this function is only used to combine the incrementality of two levels into one. However, we are typically not interested in *why* the result of a model analysis changed and it suffics to know *that* the value has changed. Therefore, it is viable to lose track of whether the outer or the inner incremental value has caused a value to change.

For a formal definition, we first define what we mean by $\mathcal{I} \times \Delta\Omega$ in the following rather technical proposition.

Proposition 18. Let C be a MTC and $\mathcal{I}$ an endofunctor on C_Ω. Then, the point-wise tuple $\mathcal{I} \times \Delta\Omega$ that consists of objects

$$ob\,\mathcal{I} \times \Delta\Omega = \{\mathcal{I}(A) \times \Delta\Omega | A \in ob\,C\}$$

and morphisms

$$Mor_{\mathcal{I}\times\Delta\Omega}(A, B) = \{(\mathcal{I}(f), \Delta\omega) | f \in Mor_C(A, B), \Delta\omega \in \Delta\Omega\}$$

is a category and the mappings $\mathcal{F}(A) = \mathcal{I}(A) \times \Delta\Omega$ and $\mathcal{F}(f) = (\mathcal{I}(f), Id_\Omega)$ form a functor $C_\Omega \to \mathcal{I} \times \Delta\Omega$. We identify this functor with the assigned category if this is clear from the context.

Proof. As composition operator on $\mathcal{I} \times \Delta\Omega$, we choose the mapping

$$(g, \Delta\omega_2) \circ (f, \Delta\omega_1) = (g \circ f, \Delta\omega_2 \circ \Delta\omega_1).$$

This is clearly associative since its components are. The identity for an object $\mathcal{I}(A) \times \Delta\Omega$ of $\mathcal{I} \times \Delta\Omega$ is given by $(Id_{\mathcal{I}(A)}, Id_{\Delta\Omega})$, which is the image of $\mathcal{F}$ for the identity on A. Furthermore, let $f : A \to B$ and $g : B \to C$ be morphisms in C_Ω, we have that

$$\begin{aligned}\mathcal{F}(g \circ f) &= (\mathcal{I}(g \circ f), Id_{\Delta\Omega}) \\ &= (\mathcal{I}(g), Id_{\Delta\Omega}) \circ (\mathcal{I}(f), Id_{\Delta\Omega}) \\ &= \mathcal{F}(g) \circ \mathcal{F}(f).\end{aligned}$$

This concludes the proposition which therefore explains the meaning of the naturality for the *apply* transformation since we used $\mathcal{F}$ in its definition. □

Finally, we arrive at the following definition for an incremental computation system:

Definition 29 (Incremental Computation System)**.** Let Ω be a set of global states. Let C be a MTC. Then an incremental computation system $\mathcal{I} : C_\Omega \to C_\Omega$ for C is a functor for which natural transformations $value : \mathcal{I} \to Id_{C_\Omega}$ and $apply : \mathcal{I} \times \Delta\Omega \to \mathcal{I}$ exist. We further demand a natural transformation $\mu : \mathcal{I}^2 \to \mathcal{I}$ and a (non-natural) transformation $\eta : Id \to \mathcal{I}$ with stateless components exist such that

$$\begin{aligned} \mu \circ \mathcal{I}\mu &= \mu \circ \mu\mathcal{I}, \\ \mu \circ \mathcal{I}\eta &= \mu \circ \eta\mathcal{I} = id_{\mathcal{I}}, \\ value \circ \eta &= Id_C \\ apply \circ (-, Id_\Omega) &= Id_{\mathcal{I}}. \end{aligned}$$

The last equation means that *apply* does not change neither the given incremental value nor the global state if the identity on the state space is passed in.

Further, we demand that applying a state change to constants does not have an effect, i.e we have that for each $\Delta\omega \in \Delta\Omega$ that

$$apply \circ (\eta, \Delta\omega) = \eta \circ \Delta\omega.$$

Here, we used the inclusion defined in Definition 22.

Remark 15. For a given type A, the *value*-transformation shall return the current value of an incremental value $a \in \mathcal{I}(A)$. The *apply*-transformation applies a given state change to an incremental value. η plays the role of an elevation of a given instance of a type A to a constant of that type. μ is required to simplify nested modifiable references, for example an attribute reference of a modifiable reference. The first two validity constraints mean that an incrementalization system actually is a monad with the slight exception that η does not have to be natural as this would be too restrictive: If a result of a computation is elevated to a constant, this must not yield the same

result as performing the computation incrementally on constant arguments - the latter may change due to state changes as well.

The *value*-transformation can (and should) be natural so that there is no difference in the result whether the current value of a modifiable reference is processed incrementally or not. Besides that, an incremental processing also means that the result are refreshed upon a state change.

The last two constraints mean that the value of a constant should always be the original instance the constant was created from. The last constraint implies that if no changes are made to the global state, modifiable references must not change.

For the correctness, we want incremental values giving us the same analysis results as we would obtain through batch mode execution. This is formalized by the below definition.

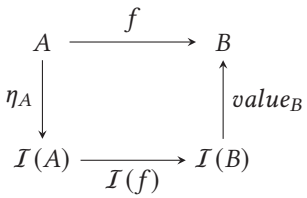
Definition 30 (Correctness of Incremental Computation Systems)**.** An incremental computation system $\mathcal{I}$ on the category $\mathcal{C}$ is correct if for every A and B in $\mathcal{C}$, every side-effect free morphism $f : A \times \Omega \to B$ in $\mathcal{C}_\Omega$ and every state change $\Delta\omega \in \Delta\Omega$ the following holds:

$$value_B \circ \mathcal{I}(f) \circ \eta_A = f \qquad \text{(Initialization)}$$

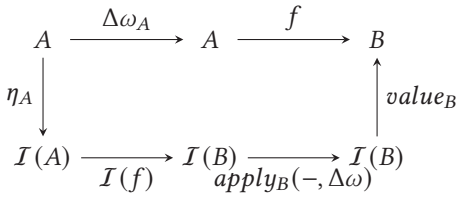
and

$$value_B \circ apply_B \circ (\mathcal{I}(f) \circ \eta_A, \Delta\omega) = f \circ \Delta\omega_A. \qquad \text{(Updates)}$$

as mappings $A \times \Omega \to B$. This corresponds to the following commutative diagram for (Initialization):



The equation (Updates) corresponds to the following commutative diagram:

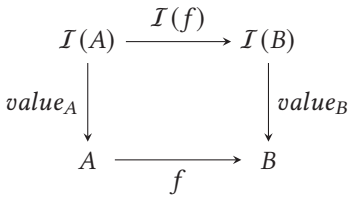


This means, if we create an incremental value for a given analysis and immediately query the current value, we get the same as if we just executed the original analysis (Initialization). Before we do that, we can apply a state change $\Delta\omega \in \Delta\Omega$ to the incremental value and then it should give us the same value as if we were obtaining the analysis result value again from scratch (Updates).

The key observation here is that while on the right hand of (Updates), the analysis function f is only used after the state change $\Delta\omega$ is applied, the left hand of the equation first evaluates $\mathcal{I}(f)$ before applying the change using $apply_B$. As a consequence, we already know the analysis f when we apply $\Delta\omega$ and can use abstractions of f to update caches.

Theorem 1. Let $\mathcal{I}$ be an incremental evaluation system for the MTC C. Then $\mathcal{I}$ is correct.

Proof. Let $f : A \to B$ an arbitrary morphism in C_Ω and $\Delta\omega \in \Delta\Omega$ be a state change. We begin by proving that (Initialization) holds for f. We first observe that the following diagram commutes due to the naturality of *value*:



We then have that

$$\begin{aligned} &value_B \circ \mathcal{I}(f) \circ (\eta_A \times Id_\Omega) \\ &\quad = f \circ value_A \circ (\eta_A \times Id_\Omega) \\ &\quad = f \circ (Id_C)_A = f. \end{aligned}$$

To proof the updates, we see that the following diagram commutes due to the naturality of *apply*:

$$\begin{array}{ccc} \mathcal{I}(A) \times \Delta\Omega & \xrightarrow{(\mathcal{I}(f), Id_{\Delta\Omega})} & \mathcal{I}(B) \times \Delta\Omega \\ \Big\downarrow{\scriptstyle apply_A} & & \Big\downarrow{\scriptstyle apply_A} \\ \mathcal{I}(A) & \xrightarrow[\mathcal{I}(f)]{} & \mathcal{I}(B) \end{array}$$

Thus,

$$\begin{aligned} &value_B \circ apply_B(\mathcal{I}(f) \circ \eta_A, \Delta\omega) \\ &\quad = value_B \circ apply_B \circ (\mathcal{I}(f), Id_{\Delta\Omega}) \circ (\eta_A, \Delta\omega) \\ &\quad = value_B \circ \mathcal{I}(f) \circ apply_A \circ (\eta_A, \Delta\omega) \\ &\quad = f \circ value_A \circ apply_A \circ (\eta_A, \Delta\omega) \\ &\quad = f \circ value_A \circ \eta_A \circ \Delta\omega_A \\ &\quad = f \circ \Delta\omega. \end{aligned}$$

This concludes the proof. □

Remark 16. Theorem 1 essentially shows that the correctness of an incrementalization system is a consequence of the naturality of the *value* and *apply* transformations. These naturalities can be checked for each morphism separately and thus enable to deduce the correctness of an entire incrementalization system from the correct incrementalization of elementary morphisms. As a reason, the commutative diagrams that are required for the naturality of a transformation can be easily stacked together as long as the functor conforms to the law that $\mathcal{I}(f \circ g) = \mathcal{I}(f) \circ \mathcal{I}(g)$. Thus, if a transformation is natural for morphisms f and g, it automatically is natural for $f \circ g$.

4.2. Integrating Dynamic Algorithms into Incremental Analyses

This section describes how arbitrary analysis frameworks can be tuned for implicit incremental computation systems (C I.1). If applied correctly, such an extension is entirely transparent for the developer of a model analysis.

Many analyses are based on recurring problems with dedicated algorithmic solutions for the incremental (dynamic) and non-incremental case, often based on graph theory. In the literature, the APIs for both kinds of algorithm are different: The API for the dynamic algorithm usually extends the API for the non-incremental case by operations that propagate input changes. For our approach, this is problematic because we assume a model analysis to be strictly separated from the model manipulation. In particular, we do not want to make the model manipulation aware that there is an incremental analysis going on. Rather, the analysis has to adapt to the changed model automatically. Therefore, the goal of this section is to describe how algorithms need to be reified for incrementalization.

For this, we first explain why different algorithms are necessary in the incremental case and then present the approach how such problems must be reified for incrementalization.

4.2.1. Choice of Algorithms

As an example for graph algorithms beyond queries, we have chosen connectivity analysis to explain our approach. This means, we analyze whether two nodes in a graph are connected, i.e. whether there is a path between them.

In batch mode, one would typically use a *Union-Find* data structure that is created in $\Theta(n + m\alpha(n))$ time [200] and answers connectivity queries in $O(\log n)$ time where n is the number of vertices, m is the number of edges and α is the inverse Ackermann function [201]. This amounts to $\Theta(n + m\alpha(n))$ when we answer at most $O(\log n)$ connectivity queries. As Tarjan showed, this solution has optimal asymptotic complexity [200].

The Union-Find data structure essentially adds a parent-pointer to each vertex pointing to a representative of its strongly-connected component.

These pointers are followed until an element is found which references itself. Then, two vertices are in the same cluster iff their pointers ultimately point to the same element. The data structure is created by iterating through all edges and making sure that vertices connected by an edge are always in the same cluster.

The Union-Find data structure does not support decremental updates, i.e. when edges are removed from the graph the entire data structure has to be rebuilt. However, there is also a fully dynamic connectivity algorithm by Holm et al. [109]. Fully dynamic here means that the problem size may increase (incremental) or decrease (decremental) and the algorithm handles this input change asymptotically faster than recreating the entire data structure (cf. Section 2.2).

As Holm et al. suggest [109], one can create and maintain a data structure of dynamic spanning forests in the graph, thus answering connectivity queries in amortized $O(\log n)$ time while requiring amortized $O(\log^2 n)$ time for updating the data structure when edges are inserted or deleted. This yields a total time of $O(\log^2 n)$ to update analysis results on model changes if at most $O(\log n)$ connectivity queries must be answered. This is faster than recreating a Union-Find data structure for each change which requires $\Omega(n + m\alpha(n))$ time as the existing data structure cannot be reused in case of edge deletions.

The key observation here is that the incremental algorithm in this case, maintaining a dynamic spanning forest, is entirely different to the batch mode approach of using a Union-Find data structure. While Tarjans Union-Find data structure efficiently answers connectivity queries, the dynamic spanning forest by Holm allows to be updated even if edges are deleted from the graph.

However, although Holms dynamic spanning forest algorithm is known for more than a decade, it can be doubted that many analysis developers are aware of it or even can implement it. For a developer of an analysis, it is more common to simply use an implementation of connectivity analysis provided by a library, without a deeper understanding of the algorithm that is used behind the scenes. The rationale behind our approach is that the developer of that library probably knows the dynamic algorithms available, but requires a way to implement that algorithm in a way such that an incrementalization system is able to pick up this implementation.

4.2.2. Reification of the Problem for Incrementalization

Incrementalization approaches that work tied to the batch implementation cannot see the algorithmic problem and therefore are not aware that it has an elegant solution in an incremental setting which is entirely different to the best solution in the batch scenario. Further, most graph algorithms are specified in imperative code that modifies some internal state in loops where some loop invariant ensures the correctness. On an instruction level, the state space for these internal states is very large and invariants are hard to identify automatically. Therefore, to achieve an efficient result, the incrementalization has to chose a different algorithm for an incremental execution.

Because finding a suitable incremental algorithm for a given problem can easily take decades of research (such as in the case of graph connectivity analysis), finding the best incremental algorithm cannot be done automatically. However, when creating a framework for connectivity analysis, developers of this framework may know such an alternative solution as they are aware of what the method is supposed to do. Our approach enables them to create a custom incremental derivation of the connectivity analysis that uses Holms dynamic forests instead of a Union-Find data structure. The analysis developer can simply reuse the connectivity analysis as a building block and the incrementalization automatically decides whether to run the connectivity analysis using Tarjans *Union-Find* data structure or Holms dynamic spanning forest, depending on whether the analysis is executed in batch mode or incrementally.

The basic idea is to enable developers to provide a custom implementation of the functor application of framework functions. An explicit functor application is only required once for each generic analysis method such as connectivity analysis while it may be used in a multitude of analyses.

The advantage of our formalization of incrementality as a functor is that the correctness of the whole analysis simply follows from the requirement that functors respect functional composition, i.e. for morphisms $f : A \to B, g : B \to C$ we have that $\mathcal{I}(f \circ g) = \mathcal{I}(f) \circ \mathcal{I}(g)$.

In terms of programming languages, this means that a function f from $A_1 \times \ldots \times A_n \to B$ with n parameters must be mapped to a function $\mathcal{I}(f) : \mathcal{I}(A_1) \times \ldots \times \mathcal{I}(A_n) \to \mathcal{I}(B)$. Here, the advantage of the category over previous approaches is that if any of the A_i is a function type, then the

incrementalization of $f(a_1, \ldots, a_n)$ may access the incrementalization $\mathcal{I}(a_i)$ of this function since the functor treats functions and objects the same. This allows to provide explicit incrementalizations for higher-order functions.

For example, consider again Listing 1.1 on page 23. The query in this listing is translated into calls to SQO methods such as `SelectMany`, `Where` or `Select`. These method calls need to be mapped to their incremental derivatives, i.e. functions to which the incremental computation system $\mathcal{I}$ has been applied.

Calls to these higher-order functions need to be mapped to calls of the incremental derivatives, i.e. functions to which the incremental computation system $\mathcal{I}$ has been applied. An easy specification method is possible in languages that keep metadata such as Java or C#. The metadata of a function can then simply contain a reference to the incremental derivative, e.g. through .NET attributes or Java annotations.

While this approach is a straight-forward outcome of the formalization of incrementality as a functor (cf. Section 4.1), it has a strong impact on the API design of analysis frameworks. In algorithmics, fully dynamic algorithms such as the connectivity algorithm presented by Holm et al. [109] are often designed with an API that mixes the functional specification of the algorithm (in the example a function returning whether two vertices are connected) and an API to adjust the data structure to updated input (in the example methods that insert or remove edges from the graph). As a consequence of our approach, the latter is no longer necessary and thus the API gets cleaner. Instead of explicit commands, the adoption to an updated input is implicitly in the functor implementation of the algorithm.

In the example of connectivity analysis, we can (and have to) reduce the API to the two elements below.

- `Connectivity<T>(`$T*$ `vertices,` $T \to T*$ `edges) : Connectivity<T>` creates a new data structure to decide whether two elements of type T are connected where the underlying graph is given by a set of vertices and for each vertex the incident edges. Here, $T*$ denotes the Kleene closure, i.e. a collection of type T.
- `AreConnected(`T `a,` T `b) : bool` as an instance method of the resulting data structure determines whether the vertices `a` and `b` are connected.

In batch mode, the method `Connectivity` creates a *Union-Find* data structure as proposed by Tarjan. On this data structure, the method `AreConnected` checks for two instances of the domain, whether the parent pointers are pointing to the same element.

In the incrementalized version, the result of `Connectivity` is an incremental value of a connectivity object created using Holms dynamic spanning trees, inheriting from the same abstract base class. This methods gets as an input an incremental value for the vertices in the graph and an incremental value for the method describing the outgoing edges. This object will react on changes in the vertices appropriately by adding or removing edges in the dynamic forest. If for example the value for the parameter `edges` changes entirely, it may also simply return a new Connectivity object, meaning that the present dynamic forest is discarded.

The method `AreConnected` of the incremental dynamic spanning tree implementation compares the root nodes for both involved trees and looks whether they match. Furthermore, it hooks an event handler to react on changes to the dynamic forest and reruns the check afterwards. The resulting incremental boolean value represents whether this has any effect on the connection between vertices `a` and `b`.

This can be seen as a separation of concerns in the otherwise query-and-command like interface of fully dynamic algorithms. In this version, the functionality is exposed in a purely functional manner whereas the state management is entirely hidden from the developer when the analysis is run in incremental mode.

4.3. An Extensible Implicit Incremental Computation System

The basic idea of NMF Expressions is to implement incremental expression evaluation by creating a DDG where each executed instruction is reflected by a node in the DDG. As usually many instructions are necessary to compute an analysis, these graphs become very large and may easily consume enormous amounts of memory. This makes the integration of manually incrementalized functions tremendously important to avoid that graph traversal outweighs the savings in terms of incremental computation.

In the remainder of this section, we first introduce the overall concept in Section 4.3.1, discuss the incrementalization at instruction-level and its correctness in Section 4.3.2, incrementalization of higher-order functions in Section 4.3.3 and the extensibility in Section 4.3.4.

4.3.1. Overview

To implement an incremental computation system, one of the first decisions to make is how state changes in $\Delta\Omega$ should be mapped to the type system in order to provide an implementation of the `apply` transformation. We have chosen an implicit representation through an event. That is, whenever a state change occurs, all DDG nodes execute `apply`, immediately.

In a modeling environment, the state changes in $\Delta\Omega$ are model changes that can be recorded using standard notification APIs. NMF reuses the notification API that is common in the .NET platform, available through the interfaces `INotifyPropertyChanged` and `INotifyCollectionChanged`. Because the implementation only makes use of these two interfaces, it can also be used with model classes that are not generated from a metamodel but written directly[31].

Our implementation uses a generic interface `INotifyValue` for the mapping of types to decouple the monad[32] as much as possible from concrete implementations. Interfaces in .NET offer support for covariance. Thus, inheritance is transported to the monad in a type-safe way which is important given the hard implementation of generics in .NET[33].

The transformations η and μ are straight-forward to implement as extensions methods. The unit transformation η simply converts a value to a constant; the transformation μ simplifies an incremental value of an incremental value essentially by chaining the *value*- and *apply*-transformations. The functor

[31] The support for these two interfaces can even be generated automatically using aspect-oriented programming [68]

[32] The unit transformation of an incrementalization system is not natural, therefore incrementalization is not a monad in the sense of category theory, only a functor. However, monads are often defined slightly different in functional programming where the naturality of the unit transformation is not required.

[33] As a consequence of this hard implementation, the machine code for a generic method may depend on the generic type parameters which makes it necessary to know them.

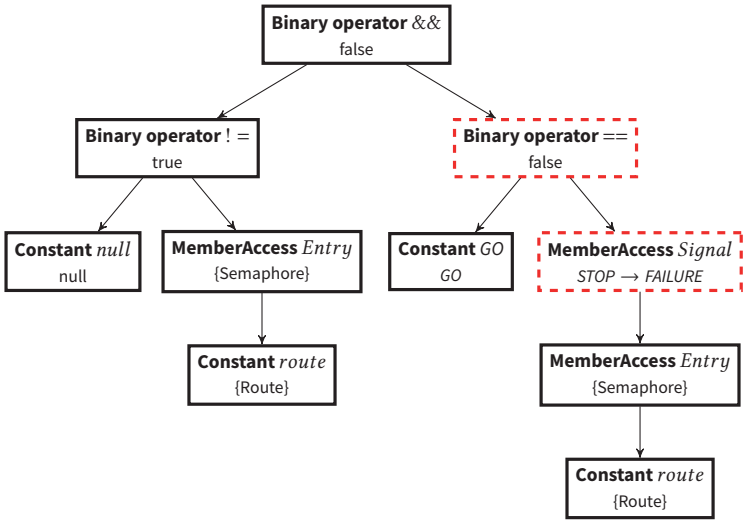


Figure 4.1.: The DDG for the predicate *route.Entry*`!=null` && *route.Entry.Signal* == *Signal.GO* and nodes that must be reevaluated when changing the signal to *FAILURE* in red and dashed.

itself is not as easy. In our implementation, we decompose methods into their abstract syntax trees and incrementalize every element of it separately, making use of the law that $\mathcal{I}(f \circ g) = \mathcal{I}(f) \circ \mathcal{I}(g)$.

For this to work, we require a decomposition of the model analysis into instructions. We obtain this decomposition at run time through the feature of the C# language to compile methods to expression trees (cf. Section 2.3.1). The usage of this language feature to build internal DSLs has been discussed first by Martin Fowler [67, p. 455] under the term *Parse Tree Manipulation*.

The resulting DDG is essentially a copy of the expression tree[34] and contains a node for each executed instruction, including the type of instruction as well as the data passed in. It is therefore much larger than comparable DDGs created by self-adjusting computation [1] that uses explicit incrementalization primitives to make the nodes as big as possible. However, it has the

[34] Unlike the original expression tree that we obtain from the compiler, the DDG nodes have generic types to optimize performance and type safety.

advantage that we have a direct representation of a method call which makes it easier to exchange such nodes with an explicit incrementalization for the given method.

If any node in the expression tree changes its value, this change is propagated up to the root of the tree that represents the value of the whole tree. Along this way, the propagation is stopped as soon as the value for a sub-expression does not change.

For example, the expression `route != null && route.Entry.Signal == Signal.GO` does not change its value if the entry semaphore of the route has a failure while showing *STOP* (depicted in Figure 4.1). The member access node to the entry semaphore does not change because the identity of the semaphore is still the same. However, the signal property of that semaphore changed. This change raises an event, fetched by the member access node and further propagated through the dependency graph. The node for the binary operator == is registered for this event and now gets notified. However, the signal still does not show *GO* and thus the change is no longer propagated.

4.3.2. Incrementalization at Instruction Level

We implemented an incrementalization for each instruction type, each represented in its own class. If a change affects an incremental value, we do not exchange the instance of the DDG node but issue an event such that dependent nodes treat the incremental value as new. The expression tree is then converted using a visitor pattern. For each of the instruction types, their incrementalization has to respect the naturality of *value* and *apply* transformations.

The naturality of *apply* simply means that 1. the creation of the DDG can be done before or after a given change is done to the model without affecting the DDG after the change and 2. the change notification is issued after each event. The former statement is true for all nodes of our implementation, as the implementation is entirely sequential and therefore the creation of a DDG node cannot interfere with the change propagation. As soon as a change happens, all DDG nodes that are affected by this change adapt themselves to the change. The latter statement and the naturality of the *value* transformation have to be discussed for each instruction type individually.

This means that at any given global state, the *value* transformation of a DDG node must match the instruction applied to the *value* of the input DDG node and if this result has changed since the last model manipulation, a change notification must have been issued. This change notification may contain detailed information on the change that may help to propagate it. An implementation for the most common types of instructions is described below.

Constants Constants never change. Thus, the event to inform clients that the value changed is not used. The *value* transformation is also a constant, which is clearly natural.

Member access A member access potentially changes either if the target model element for the member access changes or any change of the target element's properties is recorded through the notification API.

Unary expressions The considered unary expressions are type casts, conversions, unary plus and minus of numbers, logical negation and bitwise inverse. These operators only change their value when their inputs change.

Binary expressions The value of a binary expression potentially changes if either of the operand's values changes. An exception to this rule are the logical shorthand operators. In case of the conditional shorthand && operator, the right operand must not be evaluated if the left operand evaluates to `false`, as it might throw an exception. Thus, the right operand must be attached or detached from the model, depending on the value of the left operand.

Conditional expressions Conditional expressions keep a DDG for the test expression, the true expression and the false expression. Depending on the current value of the test DDG, the DDGs for the true or false expression are dynamically detached. The value of the conditional expression only changes if the value for the attached sub-DDG root node changes.

Method calls, constructors In case we have an abstract syntax tree of the method available such as for Lambda expressions, we recursively deduce a dependency graph template from it. In all other cases, we assume that a method return value only changes if either of its arguments changes, at least unless we are told otherwise, i.e. if the

developer has specified an explicitly incrementalized version. This assumption is reasonable for immutable types, particularly for platform functions like string length or the sinus function to which we do not have access.

Lambda expressions Nested lambda expressions are problematic. Because the function types of the .NET platform are fixed, using a custom function type loses the inherited compiler support[35]. Therefore, the approach of NMF is to perform a lazy incrementalization of lambda expressions. In particular, the lambda expression is only incrementalized when actually needed. If this is the case, the body expression of the function is recursively transformed into the monad as well.

Dynamic dependency graphs consume a lot of memory and are the main reason for incremental computation to have a large memory overhead. Therefore, approaches like the implicit self-adjusting computation by Chen et al. [43] argue that constant operations that do not change their value should not go into the functor since they unnecessarily increase the size of the DDG. To solve this problem, their approach generates methods for each combination of an incremental value[36] and constant value. To circumvent this problem, we introduced a constant propagation, i.e. we do create nodes in the dynamic dependency graph if a value is constant (i.e. there is no change notification provided for it) but reduce operations made on constants to constant values.

Converting the abstract syntax trees at runtime yields the decision whether or not we apply the monad. If so, we can apply the monad and obtain an incremental evaluation. If we do not apply the monad, we can simply use the .NET built-in expression compiler and get a batch mode version of the analysis with low overhead: Because the type of all expressions are already known, it is straight forward and thus very fast to compile an expression tree to intermediate language code[37].

[35] One may circumvent this problem by extending the compiler. Using technologies such as Roslyn, this seems possible and technically viable, even though one then also has to extend the Integrated Development Environment (IDE) support.

[36] called modifiable reference in [43]

[37] However, the generated method should be stored in order to avoid repeated JIT-compilation.

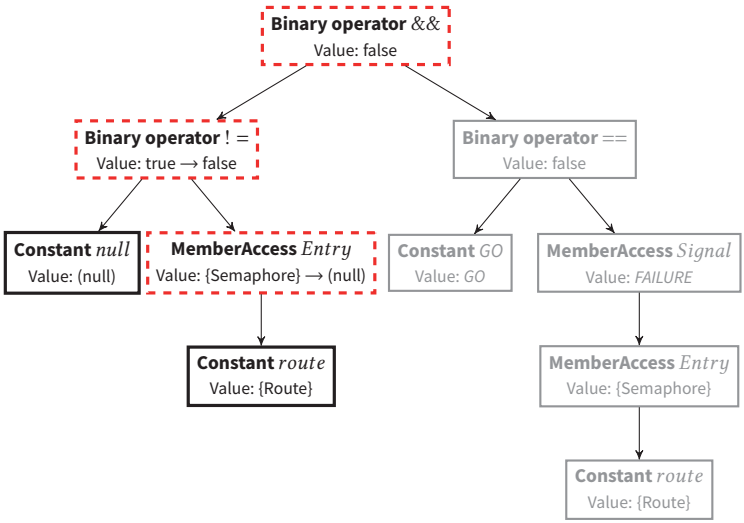


Figure 4.2.: The DDG for the predicate $route.Entry$`!=null` && $route.Entry.Signal == Signal.GO$ and nodes that are disconnected if the entry semaphore is changed to `null`.

4.3.3. Incrementalization of Higher-order Functions

Many model analyses such as the detection of wrongly set switches in the running example include the usage of higher-order functions, i.e. functions that take functions as arguments. This immediately raises the question how an incremental value of a function should look like.

To solve this problem, we use templates of DDGs. The DDG is created for the body of the function, using placeholders whenever an argument is accessed. Upon creation, the entire DDG for a function is in a disconnected state. If arguments are passed to the system, the DDG template is copied, replacing the argument placeholders with the provided DDG nodes. If all parameters are satisfied, the DDG is connected. Otherwise, the copied DDG stays disconnected and realizes the exponential mate, also known as the curried version of the original method.

DDG templates are also used for conditional and shorthand binary expressions. For example, the right side of a shorthand `&&` operator must not be evaluated if the left side already evaluates to `false`. Therefore, in that

case we deactivate the subgraph. For the predicate *route.Entry*`!=null&&` *route.Entry.Signal* == *Signal.GO* of the running example, this is depicted in Figure 4.2.

However, as an incremental analysis is usually meant to run continuously, it is very important that the algorithm is elastic in its memory consumption. This means, the memory of DDG nodes is released once they are no longer needed.

In our implementation, each DDG node has a separate counter to determine whether it should be connected or disconnected, because a DDG node generally does not know where it is used. If this reference counter is incremented to 1, the node automatically connects which means that it increments the reference counter for all of its prerequisite nodes and attaches to the model notification API if necessary. Conversely, if the reference counter is decremented to 0, the DDG node disconnects from the model and decrements the reference counters of prerequisite nodes. However, the implementation still holds a strong reference to the DDG nodes such that they are not collected by the garbage collector. This is because otherwise it would not be possible to connect to the model again.

4.3.4. Extensibility

As a key advantage of the proposed incrementalization system, we enable developers to provide an explicit incrementalization of a given function. If such an explicit incrementalization is provided, the function is no longer seen as a composition of instructions but rather treated as a primitive. For methods that do can change meanwhile the identity of their arguments stays the same, providing such an explicit incrementalization is even mandatory.

We allow users to use different variants of specifying a proxy. For a function $f : A \to B$, the user may either provide a function $\mathcal{I}(f) : \mathcal{I}(A) \to \mathcal{I}(B)$ or a function $f' : A \to \mathcal{I}(B)$. In case of the latter, we lift the provided function to the monad by a node type that changes if either the value of the argument changes or the incremental result for the current argument changes. The rationale behind this decision is that for many functions such as aggregates, it is much easier to specify the latter and does not cause additional overhead since the internal memory has to be reset entirely when the original arguments change.

For the actual specification of the manual incrementalization, we use an annotation called `ObservableProxy`. This annotation specifies a type and a method name, identifying a method that realizes the given extension point.

A problematic situation arises if the method is recursive. Creating a DDG template that contains a recursive method, we have to avoid recursively calling the proxy method. Here we make use of the fact that the DDG template for the method is only needed when the method is actually called. In particular, we use a proxy node that only copies the DDG template for the required method as soon as the node is connected to the model. As this proxy node means additional memory, we require the user to specify whether the proxy method is recursive.

4.4. Incremental Queries as an Example Extension

This section presents an implementation of our concepts for incremental queries. As queries are very popular, this implementation is also part of NMF but separated in its own assembly, proving that the incremental computation system is independent from the query implementation.

Queries can be seen as an extension of collections into a monad [77]. Thus, we only refine this monad to represent changes, i.e. combine them with the `INotifyCollectionChanged` interface commonly used in the .NET platform for collection changes. That is, instead of the usual `IEnumerable` interface, we created a new `INotifyEnumerable` interface for incremental computation and the `IEnumerableExpression` that allows users to switch between batch mode and incremental mode. `IEnumerableExpression` behaves like the `IEnumerable` monad but allows to switch to the `INotifyEnumerable` monad through a method call.

The extension of collections to a monad is supported on the .NET platform through the SQO methods (cf. Section 2.3). For each of these methods, we have defined a manual functor implementation that enables to use them incrementally. The `INotifyEnumerable` monad is already fixed to incremental execution so that we only created proxy implementations for the `IEnumerableExpression` monad or when methods leave the collection monad such

as aggregations. The proxies for the `IEnumerableExpression` monad simply switch to the `INotifyEnumerable` monad and thus enable incremental execution.

The `INotifyCollectionChanged` interface yields a high-level change representation of collections, similar to the proposal of Cai et al. [41], making abstractions from the concrete collection implementation. This change representation enables us to abstract from the index of a changed element in a collection or even the collection implementation. In the example of the average calculation, we use this change notification to update the running sum and element count accordingly.

We implemented the following extension methods that are part of the SQO both for the `INotifyEnumerable` monad and for the `IEnumerableExpression` monad: *All*, *Any*, *Average*, *Cast*, *Concat*, *Contains*, *Count*, *Distinct*, *Except*, *FirstOrDefault*, *GroupBy*, *GroupJoin*, *Intersect*, *IsProperSubsetOf*, *IsProperSupersetOf*, *IsSubsetOf*, *IsSupersetOf*, *Join*, *Max*, *Min*, *OfType*, *OrderBy*, *OrderByDescending*, *Select*, *SelectMany*, *SetEquals*, *Sum*, *ThenBy*, *ThenByDescending*, *Union* and *Where*. The semantics of these extension methods match their definitions from the SQO which are reflected by their names. We implemented the overloads that do not consider element indices that are thus not available on either of our monads. If element indices are considered, an insertion of an element often results in too many changes for incremental execution to be beneficial. In particular, adding or removing an element from a collection of n elements in the average leads to $\frac{n}{2}$ index changes, meanwhile if indices are not considered, only the removed element needs to be adjusted. Furthermore, these overloads are not considered in C# for the query syntax and are thus rather rarely used.

The explicit incrementalizations of higher-order methods such as the `Select` or `Where` operators internally manage DDGs for any element in the underlying collection. If an element in the collection is added, a new DDG is created to obtain an incremental value for the predicate the operator is using. If the element is removed from the collection, the DDG is no longer needed and removed from the node[38]. If the result of one of the element DDGs change, the corresponding change of the operator is deduced and then propagated.

[38] The implementations are aware of cardinalities larger than 1 so that effectively, the DDG is removed if the cardinality of the removed element is 0.

4.5. Distributed Incrementality using Virtual Actors

Saving intermediate results of a prior computation and managing the events appropriately requires a memory overhead. Depending on the complexity of the analysis, this memory overhead may exceed the memory capabilities of a single machine. One solution is to distribute the incremental analysis across multiple machines.

An essential result of this chapter is that the incremental semantics of a method can be annotated to a method. This section investigates how this approach can be extended to also describe the distribution of incremental methods in the same way. For this, the classes realizing the incremental analyses are mapped to the concept of virtual actors, as this model allows a simple distribution, with many features such as dynamic load adaptation and autotuning available.

This section summarizes the results of a master thesis by Benjamin Wanner [214] that I supervised. Further information can be found in the original thesis. The remainder of this section is structured as follows: The decision and strategy to map incremental analyses to virtual actors is explained in Section 4.5.1. Section 4.5.2 discusses the resulting application model. Section 4.5.3 explains how the mapping to virtual actors is carried out.

4.5.1. Distributed Incremental Analyses through Virtual Actors

The core idea is to distribute an incremental query using a virtual actor model. The actor model simplifies distributed and concurrent software development. Because the actor implementations are single-threaded, many problems often attached to concurrent programming such as data races and deadlocks can be avoided by construction. Furthermore, the virtual actor model allows a simple abstraction on which node a particular actor is executed.

For the implementation, we decided to use the Microsoft Orleans framework [38, 39] for virtual actors for the following reasons:

Fully transparent communication with actors: The communication with a virtual actor (called grain in Orleans) is the same regardless of its physical location, so the client does not need to know on which host (silo) a specific grain is executed. Requests are automatically re-routed to the correct silo.

Streams work uniformly across silos and grain client: The code to send data via a stream stays the same, independent of the communication target's location. Thus, for communication purposes only one interface has to be maintained.

Cloud deployment capabilities: For reproducible benchmarking, Orleans can easily be deployed to Microsoft Azure. The Azure tools available allow debugging of the developed Orleans application in many virtual compute nodes even on a local machine, which simplifies debugging.

Stateful actors with persistent storage: It is transparent to the application which grains are in memory and which ones have to be loaded from persistent storage. This gives additional scalability, as the amount of data stored in grains is not limited by the amount of main memory available. The only constraint is that each grain instance has to fit in a single machines main memory since the same grain activation always needs to be executed within one machine.

Further, Microsoft Orleans is completely open source and actively maintained.

4.5.2. Application Model

Same as with incrementality, the goal of this section is to hide the complexity of distributed computing from the user which in our case is the analysis developer. His task remains to specify a model analysis in a batch fashion as if it were executed locally with the exception that a cluster configuration is provided and a source where the cluster gets the model from. Furthermore, any changes to the model are also sent to the cluster so that the analysis may update its results incrementally.

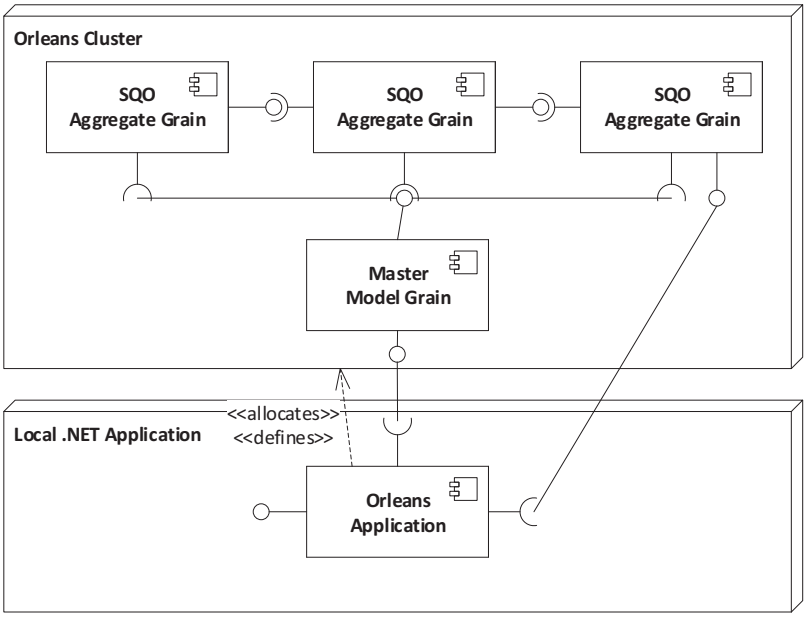


Figure 4.3.: Workflow of a distributed incremental analysis offloading incremental analysis to a Microsoft Orleans cluster (cf. [214])

The application model for such a distributed incremental model analysis is depicted in Figure 4.3.

For this purpose, the cluster defines a master model grain that stores the model and makes it available to grains that perform the incremental analysis. As a reason, the incremental processing of a SQO operator is stateful, i.e. the results may change even though the inputs did not. Since such a model access is required on each silo, the model has to be replicated for each silo.

Multiple distributed implementations of SQO operators connect with the master model to obtain the data and process changes. The exact set of these operator grains and their configuration (i.e. the used predicates) is determined by the client application that initially defines the query and thus implicitly the grains that process the query.

Finally, the grain representing the entire model analysis offers an Orleans stream of updated analysis results. The client application then may subscribe to this stream and read the results.

4.5.3. Mapping Incremental Analyses to Virtual Actors

Applying the virtual actor model to incremental model analyses yields the problem that it is unclear how the nodes in a dynamic dependency graph should be mapped to actors. A naive approach would map every node in the dependency graph to an actor. However, the communication between actors induces a certain amount of overhead: In the used Microsoft Orleans framework, this overhead consists of checking whether the referenced actor resides on the same machine and then possibly creating a new TCP connection. This is significantly more overhead than a simple in-memory method call. Given that most nodes in the dynamic dependency graph are inexpensive, this yields the risk that any advantages drawn from the incrementality may be lost by the distribution model[39].

Therefore, a more coarse granularity is required in which model analyses can be distributed. Particularly for queries, such a more coarse granularity can be offered by the SQO methods. In the implementation of Benjamin Wanner[40], an SQO call returns a grain, called aggregate grain. To process multiple items in parallel, this aggregate grain has only administrative tasks and organizes other node grains that perform the actual incrementalization of the SQO. Each of these node grains processes a subset of the input collection. This multiplexing is only done for the first SQO method call. Any further SQO call automatically connects its worker grains to worker grains of the previous SQO aggregate grain using the aggregate node as dispatcher.

This architecture is depicted in Figure 4.4. The resulting architecture for a given query is referred to as a stream processing chain.

The node grains realize the incremental processing by connecting to a per-silo model grain from which they receive model change notifications. These per-silo model grains are automatically synchronized from a master model grain to which client applications issue model changes.

[39] A more detailed analysis and performance studies can be found in the original thesis [214].
[40] `https://github.com/NMFCode/NMF/tree/distributed-expressions-orleans`

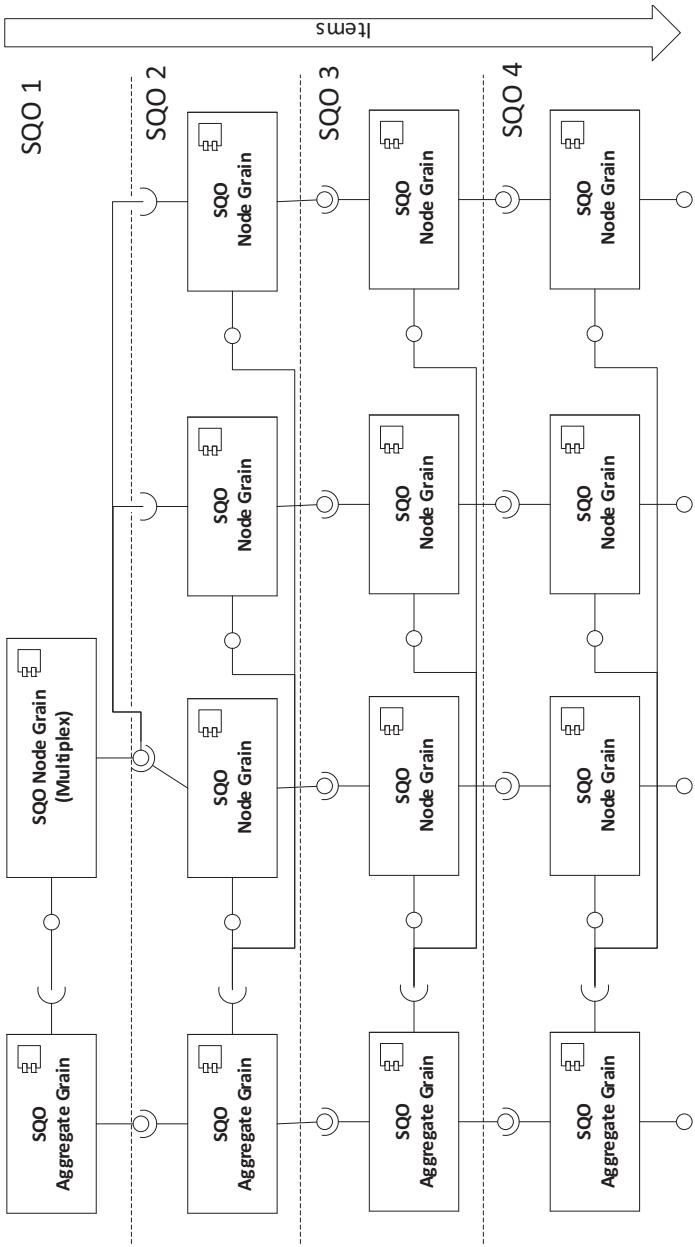


Figure 4.4: An example stream processor chain (cf. [214])

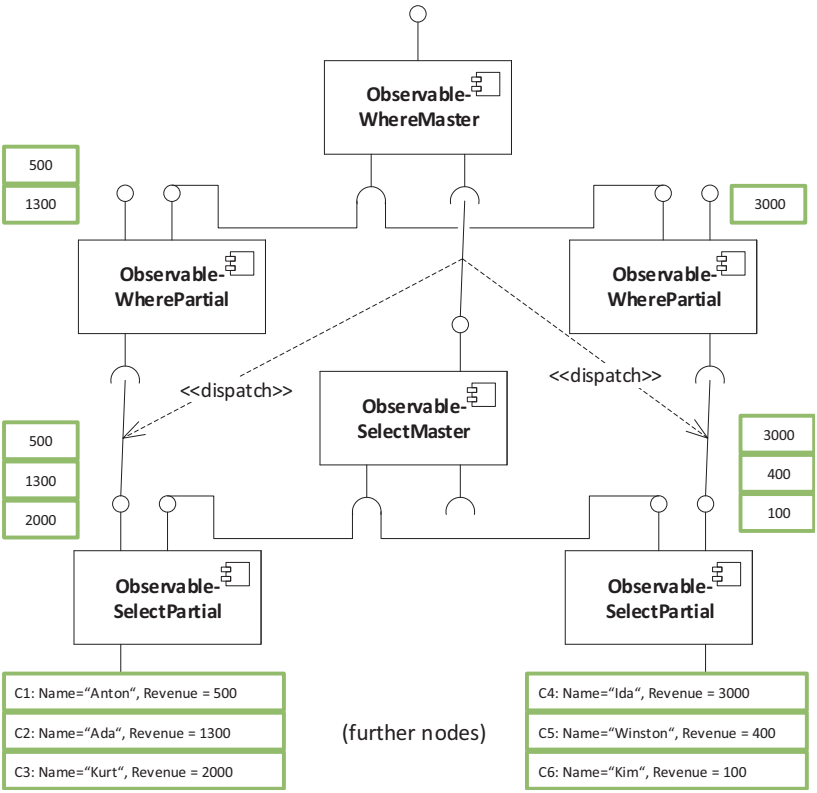


Figure 4.5.: Dispatching collection items between node grains (cf. [214])

The connection of node grains between each other is visualized in Figure 4.5 for a hypothetical example that realizes the query that selects the revenue of customers and selects those that are higher than 1000. For the selection, an aggregate node called `ObservableSelectMaster` is created that receives the incoming collection of customer objects and dispatches these collection items to its node grains, denoted with the implementation names `ObservableSelectPartial`. These nodes contain the dynamic dependency graphs of the subset of the input collection they are responsible for. For instance, the node grain on the left is responsible for the customers with names Anton, Ada and Kurt.

A subsequent SQO call, such as the `where` operator that comes next in the example, then has to take the processing results as inputs and distribute them among its worker grains. To avoid creating a bottleneck collecting the results of multiple node grains, the aggregate grain creates as many node grains as the aggregate before and establishes a 1:1 connection. Instead of enumerating the resulting (incremental) collection from the previous aggregate grain, the aggregate dispatches the node grains directly such that any updates of the previous node grain is sent to the subsequent node grain directly.

In the example of Figure 4.5, if the revenue of the customer named Anton changes from 500 to 1100, the corresponding node grain for the select operation issues a change notification that the element 500 has been replaced by the value 1100 via an Orleans stream. This change notification is sent directly to the node grain of the `where` operator (the `ObservableWherePartial` on the left of the figure) that processed the mapping of this customer before. If the `where` operator was the last SQO call in the query, the aggregate grain (the `ObservableWhereMaster` instance) collects the current result set from its node grains into a stream and pass any changes to the client.

4.6. Summary

In this chapter, we introduced a novel approach to formalize incremental computation systems as functors from category theory. This formalization allows us to generically prove the correctness of integrating custom method incrementalizations. Such an incrementalization only has to respect the naturality of the *value* and *apply* transformations. If this is the case, incrementalizations of any analysis using this method are correctly incrementalized.

From this formalization, we deduced a methodology how implicit incremental computation systems can be made extensible. Thus, they can make use of abstractions incorporated in analysis frameworks also for incremental computation, encouraging modular analyses reusing analysis frameworks. Our approach gives framework developers a tool at hand which they can use to offer implicit incrementality to their users that is tuned to their framework. For the developer of an analysis, this combines the understandability of a batch specification with the efficiency gained from framework abstractions. This saves the error-prone process of manual incrementalization and keeps the analysis more readable, thus maintainable.

The theory of incrementalization gives an answer to RQ I.2 because it clearly and formally define on a high abstraction level what requirements a user extension has to fulfill to be integrated into an incrementalization system. Theorem 1 shows the correctness of such an integration, i.e. a correct incrementalization of analyses that use a method that is manually overridden.

In abstract terms, the interface for such a user extension consists of providing developers a way (our implementation uses an annotation) to manually override the incrementalization of a given method. This manual incrementalization has to use the same implementation of the incrementalization functor $\mathcal{I}$ with regard to the types, but apart from that, the algorithmic decision how to implement such a user extension is not restricted. In particular, developers may use entirely different algorithms for the implementation than those they would use for a batch implementation. However, this may require a reification of the algorithm to match the incrementalization process. Applied to a fully dynamic connectivity algorithm, our approach reduces the API to the purely functional specification, hiding the state management from analysis developers. This part of this chapter is our answer to RQ I.1.

Furthermore, we have shown that the general approach of this chapter, to enrich the metadata of methods with information on how to incrementalize them, can also be used to combine incrementality with distributed computing and therefore ovecome memory limitations caused by large DDGs.

To give a complete answer to RQ I, we need to evaluate our approach since RQ I.3 is still open. This evaluation is done in Chapter 9.

5. Using Containments to Optimize Incremental Model Analyses

In Chapter 4, we introduced a concept how function calls can be incrementalized in terms of the function instead of in terms of its implementation. While this is helpful in case the analysis uses many generic methods such as higher-order functions, the approach is not applicable for analyses that are composed of many inexpensive operations such as arithmetic operations. However, the latter case is a frequent pattern for complex domain logic. Here, the dependency graph is still large and requires a lot of memory if not contracted. Such a contraction is the goal of C I.2 that is presented in this chapter.

To achieve such a contraction automatically, the goal of this chapter is to take parts of the model analysis in the original batch implementation and recompute these parts when relevant parts of the model change. In order to decrease the memory consumption, the detection of these model parts must be fast when changes are to be propagated. However, it may rely on static analysis results gathered when the analysis is introduced into the system. To simplify this detection process, we take the containment hierarchy of the model into account: Elementary model changes propagate along the containment hierarchy up to the root element and the rationale of the approach presented in this chapter is to statically analyze a function whether these change notifications suffice to detect all model changes that may affect a given function and what types of elementary changes are required.

If it does not, then we propose several incrementalization strategies how such a situation can be mitigated.

However, the contraction also is a conservative approximation to the changes that happen in the model. As a possible consequence, larger parts of the

analysis may have to be recomputed for too many changes. Thus, the contraction may lead to slower response times to changes, depending on the usage scenario of the incremental analysis. Since we lack the tools to accurately predict when which type of contraction or no contraction leads to the best results, we present an approach to perform an automated design space exploration of possible configurations. This approach is implemented in a tool called Incerator that is also part of NMF.

The remainder of this chapter is structured as follows: Section 5.1 formally presents the contraction strategy and gives a proof on its correctness. Section 5.2 explains our approach to generalize changes along the composition hierarchy and its implementation. Section 5.3 proposes an algorithm to obtain an approximate trigger coverage in the presence of generalized change notifications. Section 5.4 presents the incrementalization strategies based on triggers and coverages. Section 5.5 explains our approach to optimize the performance of incremental model analysis through systematic design-space exploration of available incrementalization configurations. Section 5.6 briefly introduces an implementation of this approach in Incerator. Lastly, Section 5.7 summarizes the insights and achievements of this chapter.

5.1. Covering Triggers for Incremental Values

In this section, we present the formal methods to contract the dependency graph.

This means, we try to extract an explicit incrementalization of a given method automatically through conservative approximation. Ideally, the non-functional properties of this generated explicit incrementalization are better than the instruction-level one. Here, conservative means that we may reevaluate an entire function more often then necessary. On the contrary, as a result of the contraction, the function is only represented as a single node in the DDG instead of one for each instruction which is why this conservative approximation may be advantageous for the overall system.

To ensure correctness, we need to study when an incremental value is changed. Hence, for a particular type A in a MTC C we consider the relation $Ch \subset \Delta\Omega \times \mathcal{I}(A)$ defined as follows:

$$\begin{aligned}\Delta\omega\ Ch\ a &:\Leftrightarrow (\Delta\omega, a) \in Ch \\ &:\Leftrightarrow \exists_{\omega\in\Omega} value_A(a,\omega) \neq value_A(apply_A(a,\omega,\Delta\omega)).\end{aligned}$$

Informally, a global state change $\Delta\omega$ changes an incremental value a if $\Delta\omega$ must be propagated to a as its value with respect to the *value* transformation may have changed. If the state change does not change a, then *apply* does not have have to be executed at all since it only returns its argument. Thus, a goal in the implementation will be to make as sparse as possible conservative approximations to Ch. We obtain these approximations through static analysis.

To get to such approximations, we analyze the consequences of a state change. We begin with simple property changes as in the following definition.

Definition 31. Let $f : A \to B$ be a morphism, $a \in A$ be an instance of A and $\Delta\omega \in \Delta\Omega$ be a state change. We say that the property access f of a is *affected* by $\Delta\omega$ (through state ω) if there is a global state $\omega \in \Omega$ such that $f(a,\omega) \neq f(a, \Delta\omega(\omega))$.

Definition 31 only captures single-valued properties well. Changes of mutable collections, where changing the contents of a collection does not change the identity of the collection, are not reflected. We take this into account with the next definition.

Definition 32. Let $c \in \mathcal{M}(A)$ be a bag of type A and $\Delta\omega \in \Delta\Omega$ be a state change. We say that the collection c is *affected* by $\Delta\omega$ (through state ω) if there is a global state ω and an element $a \in A$ such that $c(\omega)(a) \neq c(\Delta\omega(\omega))(a)$.

In many modeling frameworks, collections are no model elements themselves but model elements in a collection are rather connected to another model element (through a reference). Therefore, we combine Definitions 31 and 32 in another definition:

Definition 33. Let $f : A \to \mathcal{M}(B)$ be a multi-valued morphism, $a \in A$ be an instance and $\Delta\omega$ a state change. Then we say that f is *collection-affected*

by $\Delta\omega$ for a through state ω if either f itself is affected by $\Delta\omega$ through state ω or the collection $f(a, \omega)$ is affected by $\Delta\omega$ through state ω.

Remark 17. In most cases, collection properties of model elements are stateless, meaning that they always return the same collection. In that case, saying that f is collection-affected for a by $\Delta\omega$ means that the collection implementing f for a is affected by $\Delta\omega$.

Proposition 19. A morphism $f : A \to B$ is affected by a global state change $\Delta\omega \in \Delta\Omega$ for an object a if and only if its embedding $\mu^{\mathcal{M}} \circ f : A \to \mathcal{M}(B)$ is affected by $\Delta\omega$ for a.

Proof. Let ω be the global state such that $f(a, \omega) \neq f(a, \Delta\omega(\omega))$. Then,

$$\mu^{\mathcal{M}}(f)(a, \omega)(\omega)(f(a, \omega)) = 1,$$

but

$$\mu^{\mathcal{M}}(f)(a, \omega)(\Delta\omega(\omega))(f(a, \omega)) = 0$$

since the latter collection only contains $f(a, \Delta\omega(\omega))$. This proves "⇒".

Conversely, let ω be the global state such affects the collection $\mu^{\mathcal{M}} \circ f$ for the object a. Since $\mu^{\mathcal{M}}$ is stateless, $\Delta\omega$ must have affected the collection implementing f. However, this collection contains only at most one element and therefore, we deduce that $f(a, \omega) \neq f(a, \Delta\omega(\omega))$ which concludes "⇐". □

Remark 18. As a consequence of Proposition 19, in the remainder we will restrict our consideration to multi-valued morphisms. Nevertheless, it may be more efficient to keep these two artifacts separated in an implementation, since single-valued morphisms are a very frequent and thus important special case for which dedicated optimization are very useful.

In implementations, it is usually easy to decide that a state change affects a given property. We want to use this knowledge to decide when a more complex function needs to be reevaluated. We define such an indicator for a reevaluation as a trigger according to the following definition.

Definition 34. Let $f : A \to \mathcal{M}(B)$ be a morphism. We say that f is covered by morphisms $f_1 : A_1 \to B_1, \ldots, f_n : A_n \to B_n$ with selectors $s_1 : A \to A_1, \ldots, s_n : A \to A_n$ if we have for all $a \in A$ and global state changes $\Delta\omega \in \Delta\Omega$ that

$$f \text{ is collection-affected by } \Delta\omega \text{ for } a \text{ through state } \omega$$
$$\Rightarrow$$
$$\text{Any of the } f_i \text{ is collection-affected by } \Delta\omega \text{ through state } \omega \text{ for } s_i(a, \omega).$$

In case $f : A \to B$, we say that f is covered by the f_i through selectors s_i if the embedding of f into $Mor(A, \mathcal{M}(B)$ is covered by these morphisms. The collection of morphisms $f_1, \ldots, f_n$ and selectors $s_1, \ldots, s_n$ is called a trigger coverage. We denote the space of these trigger coverages as $\boldsymbol{T}$.

The problem with this definition is that we need to keep track of the selectors s_i. Therefore, a reasonable goal in an implementation is to find a coverage where the s_i are simple, for example stateless. This includes identities and projections, in case A is a tuple type. As a consequence, the indicator objects selected by s_i stay the same even when the global state changes and can be simply computed when the object a is created.

If this is not the case, the principle idea behind this chapter is to approximate when a global state change $\Delta\omega$ changes any of the $s_i(a, \omega)$ for a given state ω. We present multiple strategies to find suitable coverages and present an approach to automatically construct the Pareto-optimal strategy for a given morphism.

However, before these approximation strategies are discussed, the next couple of propositions explain how coverages can be extracted for composite model analyses. The propositions 20-23 are easy, yet very technical consequences from the definitions above, so that their proofs are omitted to save space.

Proposition 20. Any morphism $f : A \to B$ is covered by itself through selector id_A.

Proposition 21. Let $f : A \to B$ and $g : B \to C$ be morphisms such that $f_1, \ldots, f_n$ cover f with selectors $s_1, \ldots, s_n$ and g is stateless. Then $f_1, \ldots, f_n$ also cover $g \circ f$ with selectors $s_1, \ldots, s_n$.

Proposition 22. Let $f : A \to B$ and $g : A \to C$ be morphisms such that $f_1, \ldots, f_n$ cover f with selectors $s_1^f, \ldots, s_n^f$ and $g_1, \ldots, g_m$ cover g with selectors $s_1^g, \ldots, s_m^g$. Then we have that (f, g) is covered by $f_1, \ldots, f_n, g_1, \ldots, g_m$ with selectors $s_1^f, \ldots, s_n^f, s_1^g, \ldots, s_m^g$.

Proposition 23. Let $f : A \to B$ and $g : B \to C$ be morphisms such that $f_1, \ldots, f_n$ cover f with selectors $s_1^f, \ldots, s_n^f$ and $g_1, \ldots, g_m$ cover g with selectors $s_1^g, \ldots, s_m^g$. Then we have that $g \circ f$ is covered by $f_1, \ldots, f_n, g_1, \ldots, g_m$ with selectors $s_1^f, \ldots, s_n^f, s_1^g \circ f, \ldots, s_m^g \circ f$.

Remark 19. One may think that the selectors in Proposition 23 are problematic because they contain f which may be complicated. However, this is not problematic because the cover of $g \circ f$ generated by Proposition 23 also include a cover of f that does not require f.

The basic idea behind trigger coverages is to use them for an implementation of an incrementalization functor $\mathcal{I}$. The idea for this functor implementation is that it maps a morphism f to a morphism from $\mathcal{I}(A)$ to $\mathcal{I}(A) \times B \times \boldsymbol{T}$ that consists of an incremental source value, the morphism and a trigger coverage to provide a conservative approximation when the morphism should be recomputed. This morphism is a candidate for $\mathcal{I}(f)$, expressed in the following theorem.

Theorem 2. Let $f : A \to B$ and $(s_1, \ldots, s_n; f_1, \ldots, f_n)$ be trigger coverage of f. Let

$$\begin{aligned} \tilde{f} :& \mathcal{I}(A) \to \mathcal{I}(A) \times B \\ & (a, \omega) \mapsto (a, f(value_A(a, \omega)), \omega) \end{aligned}$$

a morphism that pairs an incremental value A with the result of f under the current state ω. Further, let

$$\begin{aligned} val :& \mathcal{I}(A) \times B \to B \\ & (a, b, \omega) \mapsto (b, \omega) \\ app :& (\mathcal{I}(A) \times B) \times \Delta\Omega \to \mathcal{I}(A) \times B \\ & (a, b, \omega, \Delta\omega) \mapsto \begin{cases} (a', f(value(a, \omega')), \omega') & f \text{ is collection-affected} \\ (a', b, \omega') & \text{else} \end{cases} \end{aligned}$$

where $(a', \omega') = apply_A(a, \omega, \Delta\omega)$ is the state change applied to a. In the definition of app, collection-affected is with regard to $\Delta\omega$ for a in state ω. Then the following two diagrams commute:

$$\begin{array}{ccc} I(A) & \xrightarrow{\tilde{f}} & \mathcal{I}(A) \times B \\ \Big\downarrow{\scriptstyle value_A} & & \Big\downarrow{\scriptstyle val} \\ A & \xrightarrow[f]{} & B \end{array}$$

and

$$\begin{array}{ccc} \mathcal{I}(A) \times \Delta\Omega & \xrightarrow{(\tilde{f}, Id_\Omega)} & (\mathcal{I}(A) \times B) \times \Delta\Omega \\ \Big\downarrow{\scriptstyle apply_A} & & \Big\downarrow{\scriptstyle app} \\ \mathcal{I}(A) & \xrightarrow[\tilde{f}]{} & \mathcal{I}(A) \times B \end{array}$$

As a consequence, $\tilde{f}$ is a valid implementation choice for $\mathcal{I}(f)$ if elements of $\mathcal{I}(A) \times B$ are elements of $\mathcal{I}(B)$ or isomorphic to such elements. In that case, $value_B$ can be implemented as val and $apply_B$ as app.

Proof. The commutativity of the first diagram is clear because the val function simply is a projection to the second argument which is exactly $f(value(a, \omega))$.

To see the commutativity of the second diagram, let $(a, \omega, \Delta\omega) \in (\mathcal{I}(A) \times \Omega) \times \Delta\Omega$. We have that

$$\begin{aligned} \tilde{f}(apply_A(a, \omega, \Delta\omega)) &= \tilde{f}(a', \omega') \\ &= (a', f(value_A(a', \omega')), \omega'). \end{aligned}$$

Meanwhile,

$$app((\tilde{f}, Id_\Omega)(a,\omega)) = app(a, f(value_A(a,\omega)), \omega, \Delta\omega)$$
$$= \begin{cases} (a', f(value_A(a,\omega')), \omega') & f \text{ is collection-affected} \\ (a', f(value_A(a,\omega)), \omega') & \text{else} \end{cases}.$$

In case that f is collection-affected with regard to $\Delta\omega$ for a in state ω, the equality is clear. In □

Remark 20. The *app* morphism in the last theorem essentially applies the state change to the underlying incremental value for the passed argument and then checks whether the state changes could make it necessary to update the cached value for f.

5.2. Generalization of Model Changes

To the best of our knowledge, most nowadays object-oriented programming languages do not directly support the compositionality from object-oriented design, i.e. when working with an object model, one has no information about composition as there is no distinction between associations and compositions. This is different when working with models. Since their metamodel is available, one can easily retrieve the composition hierarchy of a given model element. Further, a model element usually knows its parent in the containment hierarchy as this information is necessary for various purposes, including serialization. The goal of our approach is to use this information to coarsen the granularity of the dynamic dependency graph created for a model analysis.

Current approaches to implicit incremental computation operate on elementary model changes [166, 43]: Each feature of a model element that is used somewhere in the analysis is represented as a node in the dynamic dependency graph as well as all the intermediate results that are based on them. This node changes when the feature of this particular model element changes.

We can see batch programs as a scenario where there is a degenerated graph consisting only of a single node that holds the analysis result and changes

whenever any elementary change is made in any model element contained in or referenced by the model. This batch semantics has the advantage that the dynamic dependency is much smaller but on the other hand a lot of elementary model changes must be considered when trying to keep an analysis result updated. Our approach aims to find compromises between these extremes. Thus, instead of a single batch semantics or an alternative instruction level incremental semantics, we want to find the remaining design alternatives in between.

Therefore, we propose a notification that informs clients when an elementary model change happened in the containment hierarchy rooted at the current model element. This notification is straight-forward to implement if model elements know their parents in the containment hierarchy. Starting with the model element where the elementary model change originates from, every element has to issue this notification and then ask its parent to so as well, if any.

In our implementation, we call this notification `BubbledChange`, inspired by the equally named bubbled change events known from user interface technologies such as the Windows Presentation Foundation (WPF). The event data of this `BubbledChange` event still carries information about the type of elementary model change and the details, i.e. the name of the property that has changed. This allows the incremental change propagation to ignore elementary model changes that statically cannot have an influence on a given function.

Furthermore, as the propagation of the event can be expensive in comparison to simple elementary changes, we manage a flag in each model element whether any ancestor has a client subscribed to the `BubbledChange` event and only propagate elementary changes if necessary. This minimizes the overhead in case the `BubbledChange` is not used.

The benefit of having such a notification mechanism is that for a morphism f, we may allow also compositions (of projections) as selectors in the situation of Definition 34 without any computational overhead. As a reason, if any change affects a trigger morphism or its selector, this change is propagated automatically to the projection of the input value from where it can be consumed without tracking overhead. This is because the value of a projection in this case simply is the according parameter (from the set of parameters which form a tuple).

To make an example, we reconsider the lambda expression in Line 2 of Listing 1.1 on page 23, again depicted in Listing 5.1.

```
1 route => route.Entry != null && route.Entry.Signal == Signal.GO
```

Listing 5.1: The example predicate to check whether a routes entry semaphore shows GO

By default, NMF Expressions generates the dynamic dependency graph depicted in Figure 4.1 on page 93 where each instruction is turned into a node in the dynamic dependency graph. However, graph traversal may become a bottleneck if the DDG becomes too large and consume too much memory. The basic idea of this chapter is to contract the dynamic dependency graph of Figure 4.1 to use the entire predicate in Listing 5.1 as a single node. The correctness of this incrementalization is implied by Theorem 2.

In the example, the predicate in Listing 5.1 only changes when either the `Entry` of the given route changes or the `Signal` of that entry semaphore. Formally, the predicate is covered by the trigger morphism *Signal* with the selector $Entry \circ route$ and another trigger for *Entry* with the selector *route* where *route* is the projection of the input arguments to the *route* parameter. Because the predicate in Listing 5.1 only has a single input parameter, this happens to be equivalent to the identity[41]. In the example, the entry semaphore of a route may be contained in the route element and thus changes to its signal will be propagated to the route as well. This makes it possible to simply collect generalized changes at the *route* argument and reevaluate the predicate only if the collected change has happened either at the *Entry* or *Signal* property.

If the entry was contained in a route, this leads to the dynamic dependency graph shown in Figure 5.1 which uses significantly less nodes and thus less memory than the graph of Figure 4.1[42].

[41] The algebraic point of view that a function with multiple input parameters simply is a function with a tuple as single input is usually not used in programming. Therefore, a projection is just a reference to a parameter, same as the identity in case of a single input.

[42] In the metamodel used in the Train Benchmark, the entry semaphore of a route is *not* contained in a route (cf. Figure 1.5). We discuss strategies to handle such a scenarion in Section 5.4.

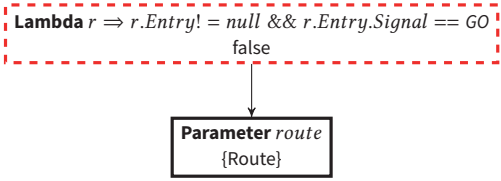


Figure 5.1.: The contracted DDG for `route.Entry != null && route.Entry.Signal == Signal.GO` and effects of changing the signal to *FAILURE* (affected nodes in red and dashed)

Since only model elements along the composition hierarchy will receive such notifications, the propagation overhead only depends on the composition height of the model which often is only logarithmic.

5.3. Obtaining Approximate Trigger Coverages

In this section, we briefly explain how trigger coverages as defined in Definition 34 can be constructed efficiently in order to create contracted dynamic dependency graphs as shown in Figure 5.1. For that, also the structure of the selectors is analyzed, in particular for the involved property accesses.

The algorithm is based on the following three main ideas:

- We represent triggers using lists in which we store the elementary morphisms that have been used along this path as well as projections. The head of this list forms the trigger morphism while the tail represents the property accesses along the selector.
- We maintain a linked list of property accesses that cover primitive expressions.
- We combine covering triggers to obtain covering triggers for the entire analysis in a divide-and-conquer fashion.

With these ideas in mind, the trigger coverage can be obtained inductively from the expressions, e.g., using a visitor-pattern. The result of the visitor is a linked list of lists that represent the triggers that we refer to in the remainder as `triggers`. In the remainder of this section, the algorithm is described for each node type separately:

Parameters: A parameter access is a projection from the input tuple type to the given parameter or an identity in case it is the only parameter. Therefore, we add a new list to `triggers` that only contains the parameter projection.

Constants: Constants never change and therefore, the trigger list for them is empty.

Unary expressions: The considered unary expressions are stateless and therefore do not have a consequence on the triggers.

Binary expressions: The binary expressions supported by most languages such as arithmetics, logical operators or bitwise operators, are stateless. Therefore, the triggers do not change as a consequence of Proposition 21. However, a binary expression implicitly also implies a product of its arguments. Therefore, we compute the trigger coverage for each argument separately and combine them, according to Proposition 22. This can be done efficiently in $O(1)$ using linked lists.

Conditional expressions: The conditional operator is also stateless but implicitly contains a ternary product. Therefore, we compute the triggers for the test expression, true expression and false expression again separately and combine them, applying Proposition 22 twice.

Property Access: When a property access is made, we assume that the property access may target an element of any current trigger. We thus simply add the property access to each current trigger list. On the other hand, if this property access makes it impossible to obtain notifications for the selector, e.g. because the selector contains a cross-reference, we still have the exact selector available because it is the target of the currently visited property access.

Method Calls: For method calls, we require annotations, similar to the annotations needed for the incrementalization. These annotations steer which data accesses are required by a given function. For higher-order functions, an implementation is provided that uses the triggers of provided lambda expressions to obtain the triggers of the higher-order function.

5.4. Incrementalization Strategies

In this section, we explain how generalized change notifications presented in Section 5.2 can be used to contract the dependency graph. We do this contraction at the level of methods. In particular, this includes lambda expressions such as the filter conditions and selectors in Lines 2-4 of Listing 1.1.

However, the generalized model changes only suffice to properly catch triggers when the selectors of coverages are projections or compositions. Therefore, we need to have strategies when the selectors include cross-references. Indeed, many predicates rely not only on the identity of cross-referenced elements but also its properties, such as in the filter predicate depicted in Line 4 of Listing 1.1, where the `CurrentPosition` of a cross-referenced switch is accessed. Here, changes to this current position will not be propagated to the `SwitchPosition` element because the switch is not contained in it. In particular, we cannot be sure that the sub-models spanned by the containment hierarchies started by the parameters of the predicate contain all model elements necessary to compute the predicate. Therefore, other strategies are necessary to find out when this predicate needs to be reevaluated.

We propose several strategies that are presented in the subsequent subsections.

5.4.1. Instruction-Level

In any case, we can fall back to instruction-level incrementality (cf. Section 4.3) and ignore the information about containment references entirely. While this approach has an optimal computational complexity, memory consumption and time used to traverse the dependency graph may outweigh the advantages by incremental computation.

5.4.2. Argument Promotion

One approach to resolve this situation is to extract the access to the cross-referenced model element into a new parameter of the predicate. For an example, consider the (true) case that the entry semaphore of a route is

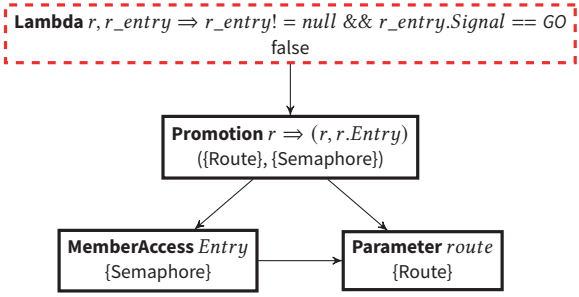


Figure 5.2.: The contracted DDG for `route.Entry != null && route.Entry.Signal == Signal.GO` if the entry semaphore is not contained in a route and effects of changing the signal to *FAILURE* (affected nodes in red and dashed)

not contained in the route. In that case, the dependency graph in Figure 5.1 is invalid because elementary changes of a routes entry signal are not propagated to the route. Therefore, the expression `route.Entry` is extracted as an additional formal parameter. As an immediate consequence, all elementary model changes that may require a reevaluation of the predicate are propagated to the formal parameters of the predicate.

In the case of the check for route entries with signal *GO*, this means that we extract a new predicate

```
(route, route_entry) => route_entry != null &&
                        route_entry.Signal == GO.
```

For this predicate, we have the situation that every change that affects the predicate is an elementary change in any of the parameters. However, this leaves the problem that we must convert any invocation of the original predicate into an invocation of the new predicate. Here, we make use of that `route_entry = route.Entry`.

If we do this for all cross-references, we arrive again at a method that does no longer contain property accesses to cross-referenced model elements. Since we assume a side-effect-free language such as fully functional λ-expressions, we can also combine multiple accesses to the same property into a single parameter.

We have to encapsulate the resulting predicate in a new predicate which derives the parameters of the new predicate from the original parameters (incrementally). Since this parameter derivation by construction requires several accesses to cross-references, it is unlikely that benefits can be drawn from the containment hierarchy and thus, this derivation is done using instruction-level incrementality. In the example of an entry semaphore not being contained in a route, this means we arrive at the DDG depicted in Figure 5.2. This graph contains four nodes which still is an improvement over the DDG from Figure 4.1.

Since this strategy reduces the dependency graph of a potentially complex predicate to the dependency graph of subexpressions that break the containment hierarchy, it should be very useful in case of predicates with a complex Abstract Syntax Tree (AST) that consists mainly of computationally inexpensive operations. The premier area of application are binary expressions such as arithmetic or logical operators. Here, this strategy should give a significant improvement.

5.4.3. Reaction to Repository Changes

Any changes that affect the cross-referenced element will ultimately end up in the repository where the model elements are registered in. Therefore, we can listen to this event but still filter on the property names. However, this means to recompute the method for a large set of elementary model changes.

The big advantage of this strategy is that it adds a very low overhead in terms of memory usage and initialization. Its disadvantage is that it is limited to a static dependency analysis. This is useful either if predicates for a given input hardly change their value or the predicate is called with a very limited set of input values. The former may be the case for example if the changes that would affect the predicate result to change do not occur in practice. Whether this is the case or not cannot be judged without context: it depends heavily on the usage scenario under which the model analysis is used.

In the scenario of checking whether a given switch position is satisfied, this strategy means that this check is reevaluated whenever the position of *any* switch *in the repository* is changed, regardless of whether this switch is the one referred to by the switch position or not. However, change types where

we can be sure they do not affect the predicate value such as a changed semaphore will not cause the predicate to be reevaluated. Because we use the results of static analysis, we can discard such changes as soon as we see that the changed properties cannot affect the current predicate.

5.4.4. Tree Extension

An approach that in some way combines the latter two is tree extension where the goal is to navigate the predicate arguments up in the containment hierarchy until we arrive at model arguments that cover all the arguments.

In many cases, cross-references have a scope more narrow than the entire model. Consider the railway network contains a notion of districts and let routes as well as switches be contained in districts. In this case, the switch reference of a switch position would certainly not leave the district the route is contained in. If we augment the cross-reference `Switch` of the `SwitchPosition` model element with this information, we navigate from the switch position to the district it is contained in and listen to elementary change notifications of this district. This has the advantage that elementary model changes from other districts do not cause a reevaluation of the predicate.

To implement this annotation, we propose a slight extension of the meta-metamodel to annotate cross-references with an anchor class. The semantics of a reference $r : A \to B$ to have an ancestor type C is that for any instance a of A, the least common ancestor of a that is an instance of C is also an ancestor of $r(a)$. In particular, if $A = C$, then r remains inside the containment hierarchy of a.

5.5. Finding Pareto-Optimal Configurations

Comparing the dependency graphs in Figure 4.1 and Figure 5.1, the latter consists of significantly less nodes (two instead of ten). However, the nodes in Figure 5.1 are computationally more complex and have to be reevaluated on more change events. Through static analysis, one can restrict the changes that should trigger reevaluation to elementary changes of `Signal` or `Entry` members. However, this static analysis loses the context and we would

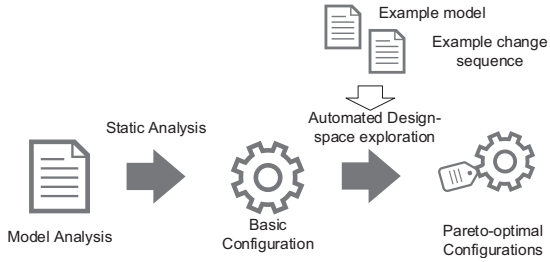


Figure 5.3.: Finding pareto-optimal incrementalization configurations using genetic search algorithms

therefore have to reevaluate the predicate if *any* of the semaphores contained in the analyzed subtree changes its signal.

Thus, even in the absence of cross-references, it is not obvious that it is beneficial to contract the DDG. Hence, the application of this contraction is a strategy which may or may not be beneficial for the performance of the analysis. We would expect a contraction to always save memory, but listen to too many change notifications. Whether or not this results in better response times to changes depends on the scenario.

As a result from the strategies to contract the dependency graph, we have a (potentially large) space of configurations how the DDG for a given model analysis can be implemented. Each configuration has a different impact on how much effort must be undertaken to update the analysis for a certain change and on the memory consumption of the model analysis. In this section, we present our approach to find the (Pareto-)optimal configuration with regard to memory consumption and response time to changes through a search-based approach. An overview of the entire approach is depicted in Figure 5.3.

The configurations are essentially assignments of incrementalization strategies to parts of the analysis (methods or predicates). Therefore, in a first step, we have to explore the degrees of freedom that we have for a given analysis, i.e. the methods that the analysis consists of. This can be done easily through static analysis that finds an initial configuration, for example configuring all methods for instruction-level incrementality.

A model analysis already contains executable semantics. Therefore, to decide between given configurations which one is the fastest, we simply run the model analysis on a predefined model with predefined set of changes and use the measurement result. Changes to the analyzed model can be represented as models again by generating a metamodel of elementary model changes [34]. We play these changes on the analyzed model and record the response times from playing an elementary model change to the updated analysis results.

The optimal configuration of the model analysis regarding the granularity of the DDG depends on several aspects such as the size and structure of the model, the frequency in which what parts of the models change and how, the memory availability on the target machine that should run the model analysis and its processing speed. Therefore, we demand from the analysis developer an example model and an example change sequence as well as access to the target machine that should run the analysis in production.

However, such a full design-space exploration requires s^n measurements for s different strategies (in our case $s = 4$) and n different predicates in the analysis. Therefore, we try to prune the design-space. Heuristically, we can always use instruction-level incrementality if a predicate or method only consists of a single property access. Further, we heuristically exclude the tree extension strategy if no anchors are defined. Still, a full design space exploration is only feasible for small analysis with few predicates.

If the number of predicates gets too high, we propose to use genetic algorithms to automatically optimize the granularity of the dependency graph according to these response times and the memory consumption of the model analysis. As objective functions, we use memory consumption and the time to run the given example change sequence on the given initial model when executed on the target machine.[43] We expect these artifacts to be provided by the developer. The outcome of the search tool is then a set of Pareto-optimal configurations to incrementalize the model analysis for the given scenario which the developer can use in production environments.

[43] Compared to other automatically optimizing algorithms, a genetic search algorithm needs less assumptions but requires more metric evaluations. Since in our case, a metric is an expensive benchmark execution, other search algorithms such as Simplex optimization algorithms may result in faster optimization but we have not tried this, yet.

Optimizing the granularity of incremental systems can yield important insights on how well particular applications fit for incrementality. In some scenarios, it might be possible that the best granularity of for the dependency graph may be the degenerate DDG that simply reevaluates the analysis after every change. In this case, we gain an insight that this type of model analyses would not be suitable to be incrementalized at all, for example due to butterfly-effects. Thus, the optimization gives us tools at hand to reason about the potential for incrementality of a given model analysis under a particular situation in a much better way.

5.6. Incerator

We have implemented our approach into a new tool called Incerator, part of NMF. In particular, Incerator automates the workflow depicted in Figure 5.3 based on an adapter implementation that runs a given analysis with an example model.

This adapter implementation only has to implement a single method `Run` that runs the model analysis and change sequence for a given model repository. The adapter implementation is passed to Incerator as the assembly-qualified name of the class. Incerator loads this type and the container assembly and instantiates the type. Thus, the type must exist and must have a parameterless constructor.

The tool then records the variation points and explores the design space either using full design space exploration or genetic search. As benchmark, the tool by default records the runtime and memory consumption of dedicated per-configuration analysis processes. Alternatively, a custom benchmark can be supplied.

The recording of variation points works by running the analysis in a special incrementalization system that statically chooses instruction-level incrementality for each analysis predicate, but records all variation points in a static variable. Incerator reads out this variable to obtain the design-space for the incrementalization of the model analysis that is run in the `Run` method of the analysis adapter.

In a next step, Incerator traverses the design-space. At the moment, the user has the choice to either explore the full design space or perform a genetic search.

The genetic search algorithm uses 5% elites of the populations, a double-point crossover operator with probability 85% and mutation operator with probability 8%. The rationale to use a double-point crossover instead of a single-point crossover is that we think that it is more reasonable to assume that only a chunk of configuration entries should be exchanged for a new generation. However, we have not performed an adequate analysis which configurations for the genetic search algorithm produce the best results. The size of the population as well as the number of generations can be specified by the user as they have a massive impact on the duration of an optimization.

By default, Incerator measures the time to run a child Incerator process that starts up and runs the specified analysis in a passed in configuration. However, this means that the time measurement is inflated by the time to start the process, load the model, load the changes, load the configuration and initialize the analysis. This overhead is applied to each configuration in the same way and therefore the results should be valid with regard to the pareto-optimality. Furthermore, Incerator measures the peak memory consumption of the spawned process.

To minimize the influence of overhead for the time measurements, Incerator further supports to specify a custom benchmark. Like the analysis adapter class, this benchmark can be provided in the form a string noting the assembly-qualified name of the benchmark implementation class. With this class, the user can influence exactly how the measurement of time and memory consumption is performed.

The main Incerator process, i.e. the process that was started by the user from the command line, then collects the results and optionally collects the set of Pareto-optimal candidates with respect to time and memory consumption. The results are returned in the form of a CSV file. Each entry of this table consists of the path to the corresponding configuration and all measured values for the benchmark metrics[44].

[44] Although only the peak memory consumption and the response time are considered in the Pareto-filter, the default benchmark also collects other performance metrics such as the working set. Custom benchmarks are free to define the benchmark metrics as long as they contain time and memory consumption.

5.7. Summary

In this chapter, we presented an approach to contract the DDG and adjust it to the actual change sequences. The approach automatically tunes the incremental execution of a given analysis by selecting among different incrementalization strategies on the level of methods and predicates[45]. This tuning depends on the given example input, an example change sequence and the given target machine. It utilizes the composition hierarchy of the input model. The approach is implemented in Incerator, which is a part of NMF.

The partition of models using their containment hierarchy seems a natural choice, but it is in fact only one option. Therefore, the approach presented in this chapter can be seen as a proposal for RQ II.1, other approaches may also be viable. Likewise, the partition of model analyses to methods and predicates has the characteristics of a proposal for RQ II.2. Therefore, the thesis leaves many open questions left as future work in this area.

For the proposed combination of model partition strategy and analysis partition strategy, we propose four different incrementalization strategies. First, this shows that RQ II.3 has not unique answer, not even if the metamodel and the analysis is fixed. Rather, we are confronted with a (search) space of possible answers. Our approach tries to select the most appropriate solution with regard to non-functional properties by an automated design space exploration, supported by our tool Incerator. This tool can be seen as an answer to RQ II.4.

As the search space is exponential with the number of predicates, it quickly becomes infeasible to perform a full design space exploration. Besides heuristic improvements, we propose a genetic search algorithm to prune the search space more efficiently, but there are many open questions in this line of research.

The actual suitability of our approach, in particular the improvements that can be achieved using Incerator for a practical model analysis, is discussed in Section 9.3.5.

[45] The formalization included in this chapter is actually agnostic of the choice how model analyses are partitioned, but our implementation is not.

Part III.

Implicit Incremental Model Transformations

6. Incremental Model Transformations

In this chapter, we introduce an approach to apply the incrementalization system proposed in Chapter 4 for uni-directional or bidirectional, incremental model transformations. We first introduce a running example for this chapter, before the synchronization formalism using synchronization blocks (C II) is introduced and an implementation in an internal DSL is presented.

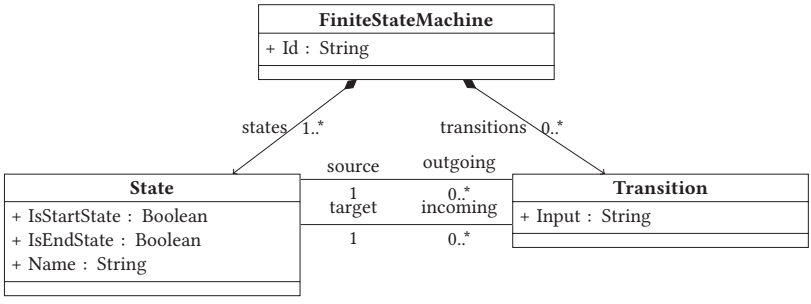
The contents of this chapter have been accepted for publication at the Springer *Software and Systems Modelling* journal [98].

The remainder of this chapter is structured as follows: Section 6.1 introduces the running example for this chapter, a synchronization of finite state machines and Petri nets. Section 6.2 introduces the formal concept of synchronization blocks and proves some important properties. Section 6.3 explains how synchronization blocks can be implemented in an internal DSL. Section 6.4 shows how the language is applied to the running example and how this language related to Triple Graph Grammars. Lastly, Section 6.5 summarizes the insights and achievements of this chapter.

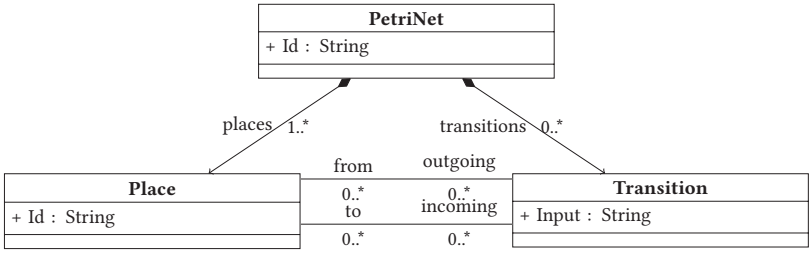
6.1. Finite State Machines to Petri Nets

Throughout the chapter, we use the example of the transformation of Finite State Machines to Petri Nets, two well-known formalisms in theoretical computer science. Both of them are well suited to describe behaviors but each of them has its advantages. Therefore, both of them are widely used. Finite state machines can be easily transformed to Petri nets.

However, for model synchronization the example of Finite State Machines and Petri Nets is a rather synthetic one as usually only one of these formalisms



(a) The metamodel for finite state machines



(b) The metamodel for Petri Nets

Figure 6.1.: The metamodels of finite state machines and Petri nets

is used. Nevertheless, we use it as our running example because the involved metamodels are rather simple and structurally similar but yet different.

The metamodel that we use for finite state machines is depicted in Figure 6.1a. Finite state machines consist of states and transitions where transitions hold a reference to the incoming and outgoing states and states hold a reference to the incoming and outgoing transitions. States can be start or end states.

The metamodel of Petri Nets is depicted in Figure 6.1b. Petri Nets consist of places and transitions. Unlike state machines where states are modeled explicitly, the state of a Petri Net is the allocation of tokens in the network.

The transformation from finite state machines to Petri Nets transforms each state to a place. Transitions in the finite state machine are transformed to Petri Net transitions with the source and target places set accordingly. Final states are transformed to a place with an outgoing transition that has no target place and therefore ‘swallows’ tokens.

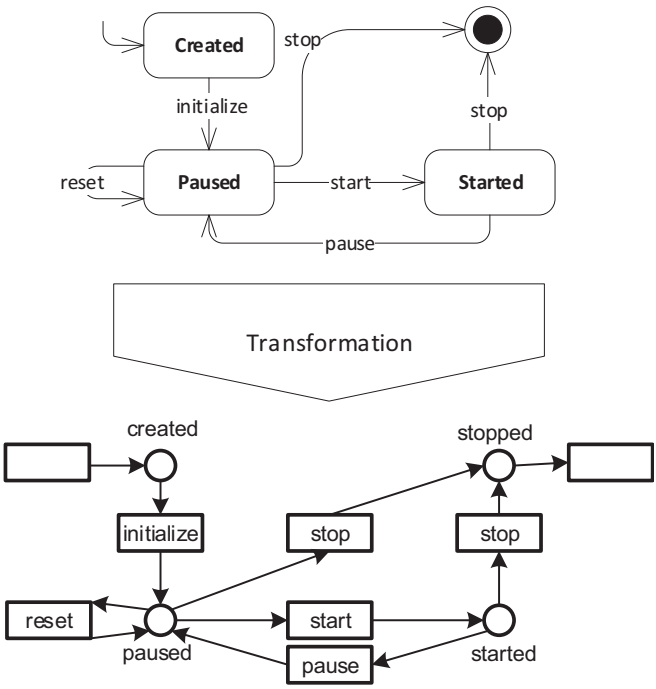


Figure 6.2.: Illustration of the considered example transformation from finite state machines to Petri nets exemplified for a simulation lifecycle.

An example of this transformation is illustrated in Figure 6.2 where the state machine to manage the lifecycle of a simulation is depicted. The advantage of a Petri net is here that using tokens, Petri nets allow to represent the state of multiple simulations in the same diagram while in state machines, only one state can be active at a time.

The backward transformation from Petri Nets to finite state machines is not always well defined since Petri Net transitions may have multiple source or target places. However, if the Petri Net is an image of a finite state machine under the above transformation, then the backward transformation is useful to have.

6.2. Model Synchronization with Synchronization Blocks

In this section, we present our model synchronization approach. First, we introduce the underlying synchronization theory our implementation is built upon. We then explain synchronization blocks as the synchronization primitives of our approach and how these primitives are composed to model synchronizations.

6.2.1. Combining Bidirectionality and Change Propagation

To combine incrementality and bidirectionality, one must find a suitable formalization able to describe both of them. On the one hand, we have incrementality which can be described as functors (cf. Chapter 4). On the other hand, we have bidirectionality, where Foster et al. have proposed the lens approach [64] for bidirectional computation. In this section, we present our approach for such a common formalization through incremental lenses.

To do that, we use that both approaches can be described in terms of category theory. However, there is an important difference of what entities are modeled in the categories: While the objects in Croles categories reconstructed from algebraic type theory are types, lenses often consider entire models as objects [57].

There have been many different versions of lenses with close correlations [119]. Diskin, Xiong and Czarnecki argue that the original lenses have problems as they do not know the change sequence and propose delta-lenses as a solution [58]. However, this problem only arises when the difference between two states is not distinct as shown by Johnson and Rosebrugh [119]. As Diskin uses categories where objects represent entire models [57][46], deltas require model differencing which in general has no unique solution.

In our case, the objects of the category consist of identities of model elements or simple values. Thus, the differencing is easy and unique: A change

[46] also referred to as model-at-a-time, as opposed to object-at-a-time

sequence is uniquely described by the previous model element identity and the new identity. Therefore, state-based lenses suffice.

Therefore, we can simply use in-model lenses such as introduced in Definition 23. However, we would also like to transfer the composition properties of the original lens definition by Foster et al. (cf. Definitions 6 and 8) to in-model lenses.

Proposition 24. There is a composition operator $\circ$ that maps a lens $f : A \hookrightarrow B$ and a lens $g : B \hookrightarrow C$ to a combined lens $(g \circ f) : A \hookrightarrow C$ if $g \nearrow$ is stateless by the following definition:

$$(g \circ f) \nearrow : (a, \omega) \mapsto g \nearrow (f \nearrow (a, \omega))$$
$$(g \circ f) \searrow : (a, c, \omega) \mapsto f \searrow (a, g \searrow (\pi_B(f \nearrow (a, \omega)), c, \omega)).$$

The latter is defined on its canonical domain.

Proof. $(g \circ f) \nearrow$ is side-effect free as concatenation of side-effect free morphisms. Let $a \in A, c \in C$ and $\omega \in \Omega$. We first proof GetPut:

$$\begin{aligned}
&(g \circ f) \searrow ((g \circ f) \nearrow (a, \omega)) \\
&\quad = f \searrow (a, g \searrow (\pi_B(f \nearrow (a, \omega)), (g \circ f) \nearrow (a, \omega))) \\
&\quad = f \searrow (a, g \searrow (\pi_B(f \nearrow (a, \omega)), g \nearrow (f \nearrow (a, \omega)))) \\
&\quad = f \searrow (a, f \nearrow (a, \omega)) = (a, \omega).
\end{aligned}$$

Here, we first applied GetPut for g and then for f.

To see PutGet, we note that

$$\begin{aligned}
&(g \circ f) \nearrow ((g \circ f) \searrow (a, c, \omega)) \\
&\quad = g \nearrow (f \nearrow ((g \circ f) \searrow (a, c, \omega))) \\
&\quad = g \nearrow (f \nearrow (f \searrow (a, g \searrow (\pi_B(f \nearrow (a, \omega)), c, \omega)))) \\
&\quad = g \nearrow (f \nearrow (f \searrow (a, g \searrow (\pi_B(f \nearrow (a, \omega)), c, \omega)))) \\
&\quad = g \nearrow (\pi_C(g \searrow (\pi_B(f \nearrow (a, \omega)), \omega)), \tilde{\omega}) \\
&\quad = (c, \tilde{\omega}) \quad \text{for some } \tilde{\omega} \in \Omega.
\end{aligned}$$

Here, we first applied PutGet for f. However, $f \searrow$ may change the state from whatever $g \searrow$ returned to some state $\tilde{\omega}$. Because we do not know

anything about $\tilde{\omega}$, we have to demand that $g \nearrow$ returns the same result regardless of the global state. As we have that, we know that $g \nearrow$ returns the same result as in the case of the PutGet of g and since we know that $g \nearrow$ is side-effect free, we even know that the final state is $\tilde{\omega}$. □

Remark 21. The composition operator $\circ$ is closely related to the concatenation operator ;, with the exception of the parameter order. In category, it is common to read $g \circ f$ as 'g after f' whereas the original lens concatenation $(f; g)$ means 'f, then g'. Intuitively, this is the same.

Example 23. An example of lenses where the GET morphism is stateless are arithmetic operations because the information what has changed is already encoded in the reference to the number. Consider for example the lens $+c : \mathbb{Z} \to \mathbb{Z}$ given by $+c \nearrow$: $(i, \omega) \mapsto (i{+}c, \omega)$ and $+c \searrow$: $(i, j, \omega) \mapsto (j{-}c, \omega)$ for some $c \in \mathbb{Z}$. Informally, the lens simply adds a constant number.

Example 24. An example of an operation beyond arithmetics is `FirstOrDefault` that returns the first item of a collection or the default value of a type (`null` for a reference type and zero for numeric types) if the collection is empty. If we were to assign $x.FirstOrDefault() = y$, i.e., evaluate $FirstOrDefault \searrow (x, y, \omega)$, we can distinguish the following cases:

1. The collection x contains y and y is the first element. In this case, we do not have to change x since the assignment is already satisfied.
2. The collection x contains y but not as the first element (if the collection is ordered). In this case, we have multiple options. We could either move y to be the first element (matching the semantics of getting the literally first element) or leave the collection unchanged (with the semantics of getting any element, e.g. in an unordered collection). This is because a single functional implementation can implement multiple semantics that need different reversability behaviors.
3. The collection x does not contain y. In this case, we add y to the collection x. We can either add it as first element if x is an ordered collection or add it to x at all if x is unordered.
4. The element y is the element type default value. In this case we again have multiple options. In our implementation we clear the collection x.

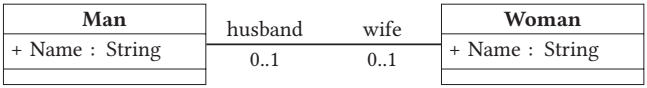


Figure 6.3.: A simple metamodel of men and women

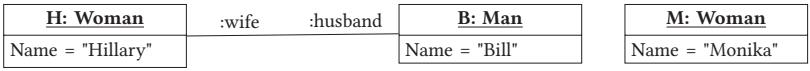


Figure 6.4.: An example instance of men and woman at state ω_0

The main learning point from this example is that the same operational implementation of an operator can match multiple lens semantics. In the example of `FirstOrDefault`, we have two versions (with different names) realizing the two options in case 2. On the other hand, this limits the possibility for implicitly inferring a reversibility semantics from existing code since there we do not know how a particular operator has been used. Thus, we decorate each operator with its reversability behavior explicitly.

Example 25. An example that breaks PutGet for composed lenses is the following: Consider a very simple metamodel of old-fashioned[47] relationships depicted in Figure 6.3. It consists only of two classes `Man` and `Woman` that have a bidirectional reference to each other. A man may or may not have a wife and a woman may or may not have a husband.

In most metamodeling languages such as Ecore or NMeta, such a bidirectional reference is modeled as two separate references with a set opposite. This means, as soon as the developer sets this reference, implicitly also the opposite reference is set. We refer to these references as lenses $wife$ and $husband$. We will consider their concatenation $(wife \circ husband)$.

Now consider the example instance depicted in Figure 6.4.

The example instance consists of three model elements H, B and M. In state ω_0, H is the wife of B and conversely, B is the husband of H. M has no husband. We want to see whether PutGet holds for the tuple (H, M, ω_0) and the concatenation $(wife \circ husband)$.

[47] In the sense that homosexual relationships are ignored

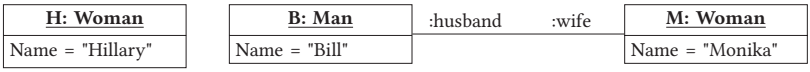


Figure 6.5.: An example of men and women at state ω_1

For this, we first have to evaluate $(wife \circ husband) \searrow (H, M, \omega_0)$. Because $wife \searrow$ will keep the identity of the model element it is based on, it will return B but change to a new state:

$$\begin{aligned}
&(wife \circ husband) \searrow (H, M, \omega_0) \\
&\quad = husband \searrow (H, wife \searrow (\underbrace{husband \nearrow (H, \omega_0)}_{B}, M, \omega_0)) \\
&\quad = husband \searrow (H, B, \omega_1).
\end{aligned}$$

This new global state is depicted in Figure 6.5. As a side-effect of $wife \searrow$, also the reference *husband* has changed both for H and for M: Because $wife$ has a maximum cardinality of 1, H is no longer a $wife$ of B which in turn resets the *husband* reference. On the other hand, M now is a wife of B and therefore the *husband* reference is set appropriately.

If we go on and evaluate $husband \searrow (H, B, \omega_1)$, this again sets the $wife$ reference of B because it is an opposite of the *husband* reference. Because $wife$ still has a maximum cardinality of 1, M is no longer a $wife$ of B and we finally arrive back in state ω_0.

The problem is now that in state ω_0, we have that $(wife \circ husband) \nearrow (H, \omega_0) = (H, \omega_0)$ whereas PutGet would demand this to be M. Even worse, because $wife$ and *husband* are opposite references, there must not be a state $\omega \in \Omega$ such that $(wife \circ husband) \nearrow (H, \omega) = (M, \omega)$. In particular, evaluating $(wife \circ husband) \nearrow$ for H in state ω_1 even throws an exception because $husband(H, \omega_1)$ returns a null-reference.

Remark 22. We do not yet have a clear criterion to automatically decide whether a given Get morphism is stateless or not. Furthermore, the fact that a given Get morphism is not stateless does not immediately imply that the resulting pair of morphisms breaks PutGet. Therefore, our implementation currently assumes that the developer is aware.

6.2.2. Synchronization Blocks

Before we describe synchronization blocks, we need a further definition.

Definition 35. A lens $l : A \hookrightarrow B$ is called persistent if for all $a \in A, b \in B$ and $\omega \in \Omega$, we have that $\pi_A(l \searrow (a, b, \omega)) = a$. This means that the PUT operation only changes the state but not the identity.

Example 26. A property access is a persistent lens as we have shown in previous examples.

Proposition 25. Let $f : A \hookrightarrow B$ a persistent lens and $g : B \hookrightarrow C$ a lens such that $(g \circ f)$ fulfills the PutGet law and therefore is a lens. Then, $(g \circ f)$ is persistent.

Proof. The proof follows straight from the definition of $(g \circ f) \searrow$. □

The very basic idea behind our approach is to describe the correspondence between elements of heterogeneous models through *isomorphisms* that are incrementally build up during a synchronization through *synchronization blocks* as in the following definition:

Definition 36 (Synchronization Block)**.** A (single-valued) synchronization block $\boldsymbol{S}$ is an octuple $(A,B,C, D,\Phi_{A-C},\Phi_{B-D},f,g)$ of four types with two isomorphisms and two persistent lenses. Here, A, B, C and D are types for which a correspondence isomorphism Φ_{A-C} is defined between A and C and likewise Φ_{B-D} between B and D. The types A and B originate from a mutable type system C_L meanwhile C and D originate from a type system C_R. We further have persistent lenses $f : A \hookrightarrow B$ and $g : C \hookrightarrow D$ in their respective type systems C_L and C_R to navigate through the models.

A schematic overview of a synchronization block is depicted in Figure 6.6. We call the isomorphism Φ_{A-C} the base isomorphism of $\boldsymbol{S}$, denoted as $\boldsymbol{S}_s$ and say that Φ_{A-C} depends on Φ_{B-D} through $\boldsymbol{S}$. Likewise, the morphism Φ_{B-D} is called the target isomorphism and is denoted as $\boldsymbol{S}_t$.

A multi-valued synchronization block is a synchronization block where the lenses f and g are typed with collections of B and D, for example $f : A \hookrightarrow B*$ and $g : C \hookrightarrow D*$.

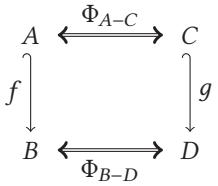


Figure 6.6.: Schematic overview of synchronization blocks

Remark 23. The semantics of such a synchronization block is to declaratively specify validation constraints that must hold for any elements $a \in A$ and $c \in C$ when they have a correspondence $(a, c) \in \Phi_{A-C}$. The lenses allow us to enforce these constraints in one direction or the other.

Remark 24. Kleene closures in this formalization are immutable, ordered collections. In an implementation, one would also like to allow mutable collections that may not be ordered. In Section 3.4, we discussed the different collection monads for mutable collections. For a synchronization block to support other types of collections, one simply has to exchange the usage of Kleene closures with the respective collection monad. However, the formalism for these other collection types is much more complicated, though the insights from them are limited. Therefore, in the remainder of this chapter, the full theory of single-valued synchronization blocks and multi-valued synchronization blocks using ordered, immutable collections is presented but the equivalent definitions and propositions for other types of collections is omitted to save space.

Remark 25. The type of the collection, i.e. the used collection monad, must be the same for both left and right side. This is necessary because the transformations between the different collection monads are only natural for some (few) cases (cf. Section 3.4). If this is not the case, for example because the left side uses $\mathcal{M}$ and the right side uses $\mathcal{K}$, then the synchronization is only performed for the weaker (in terms of the existence of natural transformations) monad. In the example, this means that the synchronization engine ignores the order of elements (because such an order does not exist in unordered collections).

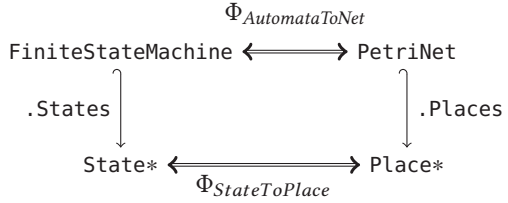


Figure 6.7.: Synchronization of the states of a finite state machine with the places of a Petri net

Example 27. As a first example, we want to synchronize the states of a finite state machine with the places of a Petri net. This can be realized through the synchronization block depicted in Figure 6.7.

The synchronization block in Figure 6.7 states that for each state of a state machine, there should be a place in the Petri net (and vice versa).

Example 28. An important special case is when $B = D$ and we can simply use the identity as Φ_{B-D}. This case is particularly relevant for the synchronization of attributes as their data types are typically used in many independent models. However, this can also be interesting when models have an overlap in model classes.

In the following definitions and propositions, we show how a synchronization block is used.

Definition 37 (Consistency with respect to single-valued synchronization blocks)**.** Let $\boldsymbol{S} = (A, B, C, D, \Phi_{A-C}, \Phi_{B-D}, f, g)$ be a single-valued synchronization block. We denote the state spaces of the type systems C_L and C_R with Ω_L and Ω_R, respectively. Let further $\omega_L \in \Omega_L$ and $\omega_R \in \Omega_R$. We say that the state pair (ω_L, ω_R) is consistent for a tuple $(a, c) \in \Phi_{A-C}$ regarding $\boldsymbol{S}$ if

$$(f \nearrow (a, \omega_L), g \nearrow (c, \omega_R)) \in \Phi_{B-D}.$$

We say that the state tuple (ω_L, ω_R) is consistent regarding $\boldsymbol{S}$ if it is consistent for all tuples $(a, c) \in \Phi_{A-C}$.

Definition 38 (Consistency with respect to multi-valued synchronization blocks)**.** In case S is a multi-valued synchronization block in the last definition, we say that the states (ω_L, ω_R) are consistent for the tuple (a, c) with respect to S if the following conditions hold:

- $f \nearrow (a, \omega_L)$ and $g \nearrow (c, \omega_R)$ have the same length and
- for each index i for $f \nearrow (a, \omega_L)$, we have that $(f \nearrow (a, \omega_L)_i, g \nearrow (c, \omega_R)_i) \in \Phi_{B-D}$.

Example 29. With respect to the synchronization block from Figure 6.7, two states ω_L and ω_R are consistent if for each pair (f, p) of a state machine and a petri net, there is an isomorphism between the states of f and the places in p, namely the isomorphism $\Phi_{State2Place}$ restricted to the states of f.

Definition 37 clearly can be used to check whether two models that should be treated equally (meaning that they are treated as isomorphic) but the more interesting use case of synchronization blocks is to repair inconsistencies. This is captured in the following propositions.

Definition 39. Let $S = (A, B, C, D, \Phi_{A-C}, \Phi_{B-D}, f, g)$ be a single-valued synchronization block. The right repair operator $\mathcal{R}_R : A \times C \times \Omega_L \times \Omega_R \to \Omega_R$ for S is defined as

$$\mathcal{R}_R(a, c, \omega_L, \omega_R) := \pi_{\Omega_R}(g \searrow (c, \Phi_{B-D}(\pi_B(f \nearrow (a, \omega_L))), \omega_R)).$$

In case S is a multi-valued synchronization block, we exchange $\Phi_{B-D}(\pi_B(f \nearrow (a, \omega_L)))$ with $\bar{\Phi}_{B-D}(\pi_B * (f \nearrow (a, \omega_L)))$ where

$$\bar{\Phi}_{B-D} : B* \to D*, (b_1; \ldots; b_n) \mapsto (\Phi_{B-D}(b_1); \ldots; \Phi_{B-D}(b_n)).$$

This means, we convert the items in the collection separately through the isomorphism.

Definition 40. Let $S = (A, B, C, D, \Phi_{A-C}, \Phi_{B-D}, f, g)$ be a single-valued synchronization block. The left repair operator $\mathcal{R}_L : A \times C \times \Omega_L \times \Omega_R \to \Omega_L$ is defined as

$$\mathcal{R}_L(a, c, \omega_L, \omega_R) := \pi_{\Omega_L}(f \searrow (a, \Phi^{-1}_{B-D}(\pi_D(g \nearrow (c, \omega_R))), \omega_L)).$$

In case S is a multi-valued synchronization block, we exchange $\Phi^{-}_{B-D}1(\pi_D(g \nearrow (c, \omega_R)))$ with $\Phi^{-}_{B-D}1(\pi_D * (g \nearrow (c, \omega_R)))$ with the closure of the isomorphism defined as above.

Proposition 26. Let $S = (A, B, C, D, \Phi_{A-C}, \Phi_{B-D}, f, g)$ be a synchronization block and $\omega_L \in \Omega_L, \omega_R \in \Omega_R$ be states such that the tuple (ω_L, ω_R) is not consistent for a tuple $(a, c) \in \Phi_{A-C}$ with respect to S. Then, the operator $\mathcal{R}_R$ can repair this inconsistency. This means, the tuple $(\omega_L, \mathcal{R}_R(a, c, \omega_L, \omega_R))$ is consistent for (a, c) with respect to S.

Proof. Assume that S is single-valued. We need to check that

$$\pi_D(g \nearrow (c, \mathcal{R}_R(a, c, \omega_L, \omega_R))) = \Phi_{B-D}(\pi_B(f \nearrow (a, \omega_L))).$$

To see this, we have that

$$\begin{aligned} & g \nearrow (c, \mathcal{R}_R(a, c, \omega_L, \omega_R)) \\ & \quad = g \nearrow (c, \pi_{\Omega_R}(g \searrow (c, \Phi_{B-D}(\pi_B(f \nearrow (a, \omega_L))), \omega_R))) \\ & \quad = g \nearrow (g \searrow (c, \Phi_{B-D}(\pi_B(f \nearrow (a, \omega_L))), \omega_R)) \\ & \quad = (\Phi_{B-D}(\pi_B(f \nearrow (a, \omega_L))), \tilde{\omega}) \quad \text{for some } \tilde{\omega} \in \Omega_R. \end{aligned}$$

The $\tilde{\omega}$ is precisely the result from the repair operator $\mathcal{R}_R$ and hence the result of the Put operation of g.

Here, we used that g is persistent and we therefore know that $\pi_C(g \searrow (c, \ldots)) = c$. The rest follows from the PutGet for g. The projection of the resulting tuple is exactly what is requested. The proof for the multi-valued case is exactly equivalent. □

Remark 26. It may be possible that there is not yet a corresponding element for $\pi_B(f \nearrow (a, \omega_L))$ while resolving Φ_{B-D}. In that case, the engine may decide whether or not to extend the isomorphism Φ_{B-D} dynamically by creating an entry for the tuple $(\pi_B(f \nearrow (a, \omega_L)), \pi_D(g \nearrow (c, \omega_R)))$. Whether or not this is intended is usually application-specific. In our implementation, we deny creating such a trace entry if either of the elements is a `null`-reference. Otherwise, a new model element is created. However, this behavior can be easily overridden. In case of collections, the reconstruction of Φ_{B-D} is done element-wise.

Example 30. To give an example to the last remark, consider again the synchronization block from Figure 6.7 that synchronizes the states of a state machine with the places of a Petri net. Consider that we start to synchronize two consistent models but the isomorphism $\Phi_{State2Place}$ is not yet populated, for example because the synchronization is run in an offline scenario. In that case, the engine has two options: It could either create entirely new places and discard the existing ones or it could try to reuse the existing places. Because creating new model elements may lead to information loss, our implementation always tries to reuse existing model elements. To match the states to the existing places, we request the developer to specify when a new trace entry should be created. Therefore, we may specify that creating such a new correspondence tuple is permitted if the names of the state and place match (cf. Section 6.3.2).

Proposition 27. Let $\boldsymbol{S} = (A, B, C, D, \Phi_{A-C}, \Phi_{B-D}, f, g)$ be a synchronization block and $\omega_L \in \Omega_L, \omega_R \in \Omega_R$ be states such that the tuple (ω_L, ω_R) is not consistent for a tuple $(a, c) \in \Phi_{A-C}$ with respect to $\boldsymbol{S}$. Then, the operator $\mathcal{R}_L$ can repair this inconsistency. This means, the tuple $(\mathcal{R}_L(a, c, \omega_L, \omega_R), \omega_R)$ is consistent for (a, c) with respect to $\boldsymbol{S}$.

Proof. The proof is exactly symmetric to the proof of Proposition 26 as synchronization blocks are entirely symmetric. □

Proposition 28. The right repair operator $\mathcal{R}_R$ is hippocratic in the sense that if the states (ω_L, ω_R) are consistent for the tuple $(a, c) \in \Phi_{A-C}$ (with respect to $\boldsymbol{S}$), then $\mathcal{R}_R(a, c, \omega_L, \omega_R) = \omega_R$.

Proof. Again, we proof this proposition only for single-valued synchronization blocks as the proof for multi-valued synchronization blocks is equivalent. We have that

$$\Phi_{B-D}(\pi_B(f \nearrow (a, \omega_L))) = \pi_D(g \nearrow (c, \omega_R)).$$

Therefore,

$$\begin{aligned} g \searrow &(c, \Phi_{B-D}(\pi_B(f \nearrow (a, \omega_L))), \omega_R) \\ &= g \searrow (c, \pi_D(g \nearrow (c, \omega_R)), \omega_R) \\ &= g \searrow (c, g \nearrow (c, \omega_R)) = (c, \omega_R). \end{aligned}$$

This is because $g \nearrow$ is side-effect free and therefore always returns the same state it was executed with – in this case ω_R. The last line is a consequence of GetPut. The projection of the resulting tuple is ω_R as requested. □

Proposition 29. The left repair operator $\mathcal{R}_L$ is also hippocratic.

Proof. The proof is once again exactly symmetric to the proof for $\mathcal{R}_R$. □

Remark 27. If one of the input model changes, reflected by a state change, then all synchronization blocks must be revisited to check whether the states of the input models are still consistent with respect to this synchronization block. Depending on the size of the base isomorphism but also the complexity of the involved lenses, this can be very time-consuming (which is not reflected in the formalization). Therefore, we use the incrementalization to accelerate this process.

In particular, the synchronization engine may keep an incrementalization

$$i = \mathcal{I}(f \nearrow)(\eta_A(a)).$$

In this case, calling $f \nearrow (a, \omega)$ is replaced by $value(i, \omega)$ which yields the same result due to Theorem 1. In case the model state $\omega \in \Omega_L$ is updated, the system may use the *apply* transformation to apply the model change sequence $\Delta\omega \in \Delta\Omega_L$ that lead to the new state directly to the incremental value of the lens.

We observed that it sometimes comes in very practical to be able to also have synchronization blocks that only allow to repair inconsistencies in one direction. This may be because one of the models contains not invertible analysis results from the other model, the transformation should be only unidirectional or such a one-way synchronization block accounts for a flaw in some other synchronization block where the lens does not respect the PutGet law in some cases[48]. After all, the design-aim of one-way synchronization blocks is to give the developer a choice what information he would like to have synchronized and which information should not be synchronized. In

[48] This may be acceptable because the – let us call such a thing for the moment a semi-lens – can be specified much more generic in this way. An example of such a construct is given in Section 6.4

(a) from left to right **(b)** from right to left

Figure 6.8.: Schematic overview of unidirectional synchronization blocks

the latter case, the synchronization engine may still be used to detect any inconsistencies. This is the subject of the next two definitions:

Definition 41 (One-way synchronization block)**.** A one-way synchronization block $\boldsymbol{S}$ is an octuple $(A,B,C,D,\Phi_{A-C},\Phi_{B-D},f,g)$ like a regular synchronization block with either of the following exceptions:

- f is not a lens, but a regular morphism $f : A \to B$ (single-valued) or $f : A \to B*$ (multi-valued). In this case, we call the one-way synchronization block a Left-to-Right synchronization block.
- g is not a lens, but a regular morphism $g : C \to D$ (single-valued) or $g : C \to D*$ (multi-valued). In this case, we call the one-way synchronization block a Right-to-Left synchronization block.

The consistency for one-way synchronization blocks is the same as for regular synchronization blocks except that the missing lens' GET has to be replaced by the respective regular morphism.

A diagrammatic overview of unidirectional synchronization blocks is depicted in Figure 6.8.

Remark 28. The advantage of a one-way synchronization block is that the choice of the function is more liberal and the transformation developer may also chose non-invertible functions.

Proposition 30. The consistency repair operator $\mathcal{R}_R$ is also applicable for Left-to-Right synchronization blocks and in that case is also hippocratic.

Likewise, the consistency repair operator $\mathcal{R}_L$ is also applicable for Right-to-Left synchronization blocks and in this case, is also hippocratic.

Proof. The proof is equivalent to the case of regular synchronization blocks where we again exchange the GET operation of the missing lens by the regular morphism. □

6.2.3. Composition of Synchronization Blocks

A synchronization block is made to define how an isomorphism should be populated based on the knowledge of another one. In the implementation, these isomorphisms are usually synchronization rules so that the synchronization block in Figure 6.7 specifies how to build up the isomorphism *StateToPlace* from the isomorphism *AutomataToNet*. This stacking process is the subject of the next definition:

Definition 42 (Model Synchronization)**.** A model synchronization is a set of synchronization blocks S with an entry isomorphism s such that for each $\boldsymbol{S} \in S$, we have that either $\boldsymbol{S}_s = s$ or there is another synchronization block $\tilde{\boldsymbol{S}} \in S$ such that $\boldsymbol{S}_s = \tilde{\boldsymbol{S}}_t$.

Remark 29. The synchronization blocks of a model synchronization can be regarded as a graph where the nodes are the isomorphisms and the edges are the synchronization blocks. A synchronization block $\boldsymbol{S}$ then points from $\boldsymbol{S}_s$ to $\boldsymbol{S}_t$. This graph may be an arbitrary directed graph. It is not required to be free of circles. Rather, circles are required to handle composite structures such as expressions.

The start isomorphism s of a model synchronization determines the signature of the model synchronization, i.e. what model elements it can synchronize. Usually, this isomorphism is defined between the root model class of the Left Hand Side (LHS) and the root model class of the Right Hand Side (RHS).

Definition 43. Because the synchronization blocks having a given isomorphism Φ as their base isomorphism describe validity constraints for tuples in this isomorphism, we associate them with Φ and refer to them as the synchronization blocks of Φ.

In particular, within a model synchronization (S, s), we have that

$$blocks_S(\Phi) := \{\boldsymbol{S} | \boldsymbol{S}_s = \Phi\}.$$

Remark 30. In practice, the isomorphisms are created and dedicated for a given model synchronization scenario. Therefore, we omit the subscript in the last definition.

Definition 44. Let (S, s) be a model synchronization, $\boldsymbol{S} \in S$ a single-valued synchronization block with source isomorphism $\boldsymbol{S}_s : A \to C$ and target isomorphism $\boldsymbol{S}_t : B \to D$. We call a tuple of states $(\omega_L, \omega_R) \in \Omega_L \times \Omega_R$ fully consistent for a tuple $(a, c) \in \boldsymbol{S}_s$ with respect to $\boldsymbol{S}$ in S, if

- the states are consistent for the tuple (a, c) with respect to $\boldsymbol{S}$ and
- the states are fully consistent for the tuple $(f \nearrow (a, \omega_L), g \nearrow (c, \omega_R))$ with respect to all synchronization blocks of $\boldsymbol{S}_t$ in S.

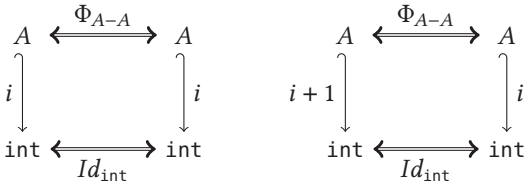
If $\boldsymbol{S}$ is a multi-valued synchronization block, then the states have to be fully consistent for all tuples spanned by $f \nearrow (a, \omega_L)$ and $g \nearrow (c, \omega_R)$ with respect to all synchronization blocks of $\boldsymbol{S}$ in S.

Definition 45. Let (S, s) be a model synchronization. We call a tuple of states $(\omega_L, \omega_R) \in \Omega_L \times \Omega_R$ consistent for a tuple $(a, c) \in s$ with respect to (S, s) if the states are fully consistent for the input tuple with respect to all synchronization blocks in s within S.

Remark 31. Like for synchronization blocks, one would like to obtain generic consistency repair operators $\tilde{\mathcal{R}}_R$ and $\tilde{\mathcal{R}}_L$ that are able to repair any possible inconsistency. Such a repair operator could be obtained by repeatedly executing $\mathcal{R}_R$ or respectively $\mathcal{R}_L$ for all inconsistencies that arise. However, we have no guarantee that repairing one consistency does not open a new one.

Remark 32. To repair an inconsistency, the synchronization blocks only change the respective model through Put operations. This means that in case of heterogeneous models, any information not contained in the other model simply is ignored, meaning that it is not propagated to the other model but left intact.

Example 31. Consider a metamodel with just a single class A that has an integer-valued property called i which we extend to a lens (cf. Example 23). We look at synchronizing two copies of this metamodel with the following two synchronization blocks:



It is clear that no state tuple can be consistent for any pair of instances of A because the i property of both copies must be the same and different at the same time which is not possible. If we go ahead and start repairing inconsistencies, then each repair will create a new inconsistency, which is why the synchronization does not terminate.

Proposition 31. If the model synchronization terminates to apply $\mathcal{R}_R$ and $\mathcal{R}_L$, it returns a new consistent state.

Proof. This proposition is an immediate consequence from the fact that the non-existence of further inconsistencies is used as the termination criterion. □

Proposition 32. The model synchronization is hippocratic, i.e. if applied to consistent changes, the model synchronization does not change the states.

Proof. This proposition is an immediate consequence from the fact that the repair operator(s) for the individual synchronization blocks is hippocratic. □

Remark 33. As the last example shows, repairing the inconsistencies between two states may not be terminating. To find a suitable theory to proof termination is subject of future work. To us, it is unclear whether such a theory may even exist, in particular, whether or not the construct of synchronizations as presented in this paper is Turing-complete.

Nevertheless, we have a formal tool that is able to repair inconsistencies one by one. If only finitely many new inconsistencies arise from fixing existing ones and the synchronization only consists of regular (two-way) synchronization blocks, then any inconsistency can be repaired automatically.

Problems as in Example 31 can be avoided if the properties used in synchronization blocks are mutually exclusive. In that case, it is unlikely that the repairing an inconsistency just produces another inconsistencies.

Furthermore, in an implementation, one can easily detect situations as precisely in Example 31 by detecting whether the executing the PUT operation is reentrant, i.e. whether changing a value in the leads to a consequence to change that same value again.

Remark 34. Propositions 31 and 32 are independent of the order in which inconsistencies are resolved. In practice, this order may be important. This is because very often, isomorphisms are defined based on other isomorphisms, such as for example Listing 6.6 in Section 6.4. Our implementation uses the literal order in the transformation specification. The fact that the correctness of the synchronization process is independent of this order also means that the transformation developer can play with it to make sure that all elements are available when the synchronization is executed.

6.3. Implementation in an Internal DSL

In this section, we present our approach for implementing model synchronization through synchronization blocks in an internal DSL. For this, we first describe how the primitives that are used in synchronization blocks, lenses, morphisms and isomorphisms are represented in the language before we describe the synchronization modes and how the model synchronization is executed.

6.3.1. Lenses and Morphisms

To support multiple transformation modes, we need to operate on incrementalizable lenses. However, most general-purpose programming languages only provide predefined operators or simple method calls, combined in compiled code – an artifact which is usually very hard to analyze. To solve this problem, we operate on a model of the code. This could be created for example by a fluent syntax [66].

In our implementation, we use a feature of our host language C# to retrieve this model of the code even simpler. C# allows us to obtain the abstract syntax tree (called expression tree) from an expression instead of compiled code (cf. Section 2.3). This means that the transformation developer writes regular C# code but the compiler does not compile this code down to intermediate language. Instead, the compiler generates code to create a model of the abstract syntax tree. Fowler calls this construction principle of an internal DSL *Parse Tree Manipulation* [67, p. 455].

The rationale behind this decision is that 1. the language adoption problem is mitigated because the transformation developer writes regular C# code, 2. the understandability is improved because the code does not contain syntactic boilerplate and 3. the tool support is maintained: The compiler still checks the correct types and the editor offers support such as code completion or navigation.

This ability to step into the compilation process is the one and only syntax feature that we use from C# that makes our language in this form impossible to implement in many other languages (apart from Visual Basic). However, we believe that other languages such as Java or in particular Xtend will soon adapt this feature as well, making our approach applicable to other languages.

For operators built into the host language, we implement default Put operators if they can be considered as lenses. For example, consider the expression $x + c$ for some incremental values x and c. Through the incrementalization system, we know that whenever x changes its value, also the value of the sum may change. For the lens, the expression resembles the Get function. The lens allows us to assign a value, say 42 to the sum given that the reference c is constant (cf. Example 23). This is applied by setting $x = 42 - c$, the Put function of the respective lens. The lens is represented by its Get function which we expect to be decorated with a Put function reference. In our implementation, this reference is realized through an annotation.

Other operators such as value equality cannot be reverted in general. It is unclear how to set an expression $x == c$ to `false`, in particular, what value to assign to x. This can be solved by additional parameters that are only taken into account when reversing the operation such as a method `EqualsOrDefault` providing the missing information with a third parameter.

To reconstruct the lens from a method call, we decorate each allowed method with an information how this method can be inverted – the developer may annotate the respective Put operation. Since this decoration is publicly accessible, this even allows developers of our approach to extend the API that can be used to specify a model synchronization.

For an example, consider the *FirstOrDefault* function that has been considered previously in Section 6.2.1. Ideally, one would like to define this function in a generic way such that for each type A, $FirstOrDefault_A : A* \rightarrow A$. To make this a lens, the developer has to specify a Put method $FirstOrDefault_A \searrow$: $A * \times A \rightarrow A*$. Alternatively, we also allow the developer to specify an operation $FirstOrDefault_A \searrow'$: $A * \times A \rightarrow \bot$ to specify that the lens is persistent.

This method is specified using a type (or type template) and a method name. The problem here is that the method may be generic but method annotations must not be generic. However, if the Get method is generic, the Put method must also be generic with the same type arguments. Therefore, the implementation collects generic type arguments in the Get method and applies them automatically also to the selected method. The system also checks the selected method that shall be used as Put operation for type conformance.

In the .NET platform, for a given type A, the Kleene closure is represented by the array type $A[]$, as long as indices of the array are not changed directly. The Put annotation is called `LensPut` in NMF. This annotation has to be put on a given method to specify its corresponding Put operation. In the example, the Put operation of `FirstOrDefault` is `PutFirst`. Therefore, an implementation for a *FirstOrDefault* lens for arrays is the one depicted in Listing 6.1.

When the framework is asked to build a lens of a given expression, it uses the abstract syntax tree of that expression and tries to apply Proposition 24 repeatedly to it. Our implementation currently does not enforce yet that g from this proposition is indeed stateless, but this is currently left to the transformation developer. However, it is usually not a problem because most synchronization blocks that we have come across so far either are very simple, most of them only consist of a single property access.

The ability to extend the language with custom lenses gives the transformation developer the possibility to extend the capabilities of the synchronization language whenever necessary.

```
static class Helpers
{
  [LensPut(typeof(Helpers), "PutFirst")]
  public static T FirstOrDefault<T>(this T[] array)
  {
    return array != null && array.Length > 0 ? array[0] : default(T);
  }

  public static T[] PutFirst<T>(this T[] array, T element)
  {
    if (array != null && array.Length > 0) {
      array[0] = element;
      return array;
    } else if (EqualityComparer<T>.Default.Equals(element, default(T))) {
      return array;
    } else { return new T[] { element }; }
  }
}
```

Listing 6.1: Implementation of a FirstOrDefault lens for arrays

If the lens is to be used also in an incremental setting, we also require to specify a function $\mathcal{I}(\mathit{FirstOrDefault}) : \mathcal{I}(A*) \rightarrow \mathcal{I}(A)$, though the implementation is actually able to lift a function $\mathit{FirstOrDefault}' : A* \rightarrow \mathcal{I}(A)$ by reevaluating the latter when the input changes. Similar to the PUT operation, we annotate this manual incrementalization using an annotation `ObservableProxy`. If this function is not provided, then the system automatically assumes that the function only changes when the input reference changes.

In the case of the *FirstOrDefault* function, such an annotation is not required because the length of an array is fixed. If the collection type was mutable, then such a proxy method must be available. As an example, Listing 6.2 contains the same lens for generic lists. For the incremental evaluation, we simply reuse the incrementalization system of NMF through the class `ObservingFunc`.

6.3.2. Isomorphisms

To represent isomorphisms, we distinguish between two cases: Identities and isomorphisms between model classes. The case of an identity isomorphism

```
static class Helpers
{
  private static ObservingFunc<IList<T>, T> firstOrDefaultFunc =
    ObservingFunc<IList<T>,T>.FromExpression(list =>
      list != null && list.Count > 0 ? list[0] : default(T));

  [LensPut(typeof(Helpers), "PutFirst")]
  [ObservableProxy(typeof(Helpers), "FirstOrDefault")]
  T FirstOrDefault<T>(this IList<T> list) {
    return firstOrDefaultFunc.Evaluate(list);
  }
  INotifyValue<T> FirstOrDefault<T>(this INotifyValue<IList<T>> list) {
    return firstOrDefaultFunc.Observe(list);
  }
  T[] PutFirst<T>(this IList<T> list, T element) {
    ...
  }
}
```

Listing 6.2: Implementation of a FirstOrDefault lens for lists

is easy since the model synchronization engine does not have to do anything as the identity of an object is easy to compute.

In the latter case, we realize the isomorphism using two unidirectional NTL transformation rules $\Phi^{\leftarrow}$ and $\Phi^{\rightarrow}$, one for transforming the models in each direction. Thus, the relationship $(a, c) \in \Phi$ is manifested in two trace entries $(a, c) \in \Phi^{\leftarrow}$ and $(c, a) \in \Phi^{\rightarrow}$. This of course implies a 1:1 relationship between synchronized elements, but the transformation developer is free to define arbitrarily many other isomorphisms for the same element(s)[49].

The implementation of synchronization blocks as NTL transformation rules also has another advantage: As we expose the underlying transformation rules, dependencies may be added to them which may result in executing a model transformation each time a new correspondence is set. This behavior is used for example in the TTC 2015 Java Refactoring case (cf. Section 9.7).

In case custom lenses do not suffice for a given task, we allow opaque synchronization blocks where we give the transformation developer full control.

[49] We do not support dynamic creation of isomorphisms, the isomorphisms must be given at compile time.

In this case, the transformation developer may hook in arbitrary C# code but has to manage all the operation modes and change propagation modes by himself.

Based on the experience we collected with the syntax of the non-incremental, unidirectional transformation language NTL [100, 101], we decided to apply a similar strategy and represent synchronization rules as generic classes. However, the body of such a synchronization rule consists of synchronization blocks rather than compiled code. We therefore use a dedicated method to create these synchronization blocks through method calls.

Synchronization rules may also serve as containers for helper methods, should they be necessary for the definition of a synchronization block. The concrete syntax is presented for our motivational example in Section 6.4.

As a further consequence, we also inherit the modularization techniques as discussed in earlier work [106, 101]. We can therefore offer the transformation developer advanced modularization techniques, such as version conflict detection, integrity checking, creating model synchronization libraries and model synchronization rule templates.

6.3.3. Synchronization Modes

As an advantage of the declarative specification of synchronization blocks, they are not tied to specific operation modes and therefore can be reused in many scenarios. We have seen that we always have a choice whether to fix inconsistencies at the left or right side of the synchronization blocks. Furthermore, in some scenarios, it may be desirable to allow certain inconsistencies. We refer to the strategies of selecting the appropriate repair operator as synchronization modes and discuss them in the following.

We support six different synchronization modes that can be combined with three different change propagation modes where we adapted the terminology from Triple Graph Grammars and refer to type systems C_L as the LHS and similarly C_R as the RHS. The synchronization modes are as follows:

LeftToRight: the transformation ensures that all model elements on the LHS have some corresponding model elements on the RHS. However, the RHS may contain model elements that have no correspondence on

the LHS. This means, we apply $\mathcal{R}_R$ as long as we find inconsistencies but do not propagate `null`-values.

LeftToRightForced: the transformation ensures that all model elements on the LHS have some corresponding model elements in the RHS. All elements in the RHS that have no corresponding elements in the LHS are deleted. In this mode, we apply $\mathcal{R}_R$ as long as possible and also propagate `null`-values.

LeftWins: the transformation ensures that all model elements on the LHS have some corresponding model elements in the RHS and vice versa. Synchronization conflicts are resolved by taking the version at the LHS. This means, we apply $\mathcal{R}_R$ as long as possible but do not propagate `null`-values. Instead, these inconsistencies are resolved using $\mathcal{R}_L$.

RightToLeft, RightToLeftForced, RightWins: same as the above but with interchanged roles of RHS and LHS

The change propagation modes are the following:

None: no change propagation is performed. In this case, also no dynamic dependency graphs for any expressions are created as they are not necessary.

OneWay: change propagation is only performed in the main synchronization direction, i.e. LHS to RHS for the first three synchronization modes and RHS to LHS otherwise.

TwoWay: change propagation is performed in both directions, i.e. any changes on either side will result in appropriate changes in the other side.

Usage examples in the TTC 2015 Java Refactpring case [94] (cf. Section 9.7) have shown that there are some cases where also the remaining operation mode to propagate updates from the target side of the synchronization back to the origin model are useful in some application scenarios but we decided not to support this operation mode in our implementation as we believe that these are rare corner-cases. Conceptually, there is no limitation.

Furthermore, there may be even more operation modes such as a check-only mode that only tests whether the selected constraints hold, but would not enforce these constraints.

The applicable synchronization direction and change propagation mode is specific to a synchronization run and is provided together with the input arguments, i.e. LHS and RHS initial models. At this initialization, we generate code to minimize the performance impact when no change propagation should be performed, i.e. the synchronization should run with a performance comparable to a transformation without change propagation as e.g. pure NMF Transformations.

6.3.4. Execution

In this section, we explain the bidirectionality a synchronization engine implementation can support based on the concept of synchronization blocks. Throughout this section, we will use the synchronization block from Figure 6.7 from page 143 as example.

For example, assume that the synchronization block from Figure 6.7 was the only synchronization block in our model synchronization and the synchronization engine was asked to execute this synchronization for a given finite state machine and Petri net model. The engine finds the synchronization rule to start with based on the synchronization rule types, in this case $\Phi_{AutomataToNet}$ and executes this rule (and therefore all of its synchronization blocks) for the given direction.

When executing the synchronization block from Figure 6.7, the synchronization engine uses the Get operation to obtain the states of the finite state machines and the places of the Petri net. Then, it tries to find a corresponding place for each state of the finite state machine, thereby executing the synchronization rule $\Phi_{StateToPlace}$. Synchronization rules are required to determine whether elements of LHS and RHS should correspond. For the synchronization rule $\Phi_{StateToPlace}$, a reasonable definition is that a state and a place should correspond when their names match. Once the correspondence has been established, it is saved in the trace. Because a subsequent query for the corresponding element of a given LHS element (or RHS element, respectively) results in a trace access, this guarantees that the correspondence relation stays bijective.

The result of this matching are three sets: a set $M_{\leftrightarrow}$ of tuples of states and places that correspond according to the $\Phi_{StateToPlace}$ isomorphism, a set $M_{\rightarrow\emptyset}$ of states with no corresponding place and similarly a set $M_{\emptyset\leftarrow}$ of places with

no corresponding state. In all direction modes, the set $M_{\leftrightarrow}$ is traversed and the synchronization engine makes sure that each synchronization block for the dependent synchronization rule is executed for each pair.

The fact that trace entries in NTL are keyed not only by their source elements but also by their transformation rules means that a model element on the LHS may easily mapped to multiple elements of the RHS (or vice versa) by just specifying multiple synchronization rules for that element.

For the contents of the two sets $M_{\rightarrow\emptyset}$ and $M_{\emptyset\leftarrow}$, the next step depends on the synchronization direction:

LeftToRight : In this direction, the set $M_{\emptyset\leftarrow}$ is ignored. The engine only traverses the set $M_{\rightarrow\emptyset}$ and creates a new element in the RHS, in the example a place, and establishes a correspondence, including to execute the synchronization blocks of the dependent synchronization rule. The newly created element is then added to the Petri net using the PUT operation of the RHS lens *.Places*. Since this is a collection-valued lens, this results in adding the element to the collection.

LeftToRightForced : In this direction, the RHS must look exactly like the LHS, up to isomorphism. Therefore, additionally to the processing of $M_{\rightarrow\emptyset}$ as in the *LeftToRight* direction, the engine removes elements in $M_{\emptyset\leftarrow}$, again using the collection interface of *.Places*.

LeftWins : In this direction, the set $M_{\emptyset\leftarrow}$ is similarly to the set $M_{\rightarrow\emptyset}$ as the synchronization engine creates new states, establishes a correspondence and adds the newly created states to the state machine using the PUT operation of the *.States* lens which again means to add the state to the collection. The direction *LeftWins* is therefore almost symmetric as the sets $M_{\emptyset\leftarrow}$ and $M_{\rightarrow\emptyset}$ are treated equally. The only difference is in case of conflicts, e.g. when the lenses are single-valued. This is very often the case for attributes such as an elements name. Here, setting a new name also means to delete the old one and therefore, the synchronization engine must only apply at most one of the lenses. Here, the direction *LeftWins* specifies that in such a scenario, the engine should always apply the LHS to the RHS.

The directions *RightToLeft*, *RightToLeftForced* and *RightWins* are equivalent except for exchanged roles of LHS and RHS.

To process the sets, the synchronization engine makes sure that there is a corresponding element in the trace, if necessary by executing the forward or backward rule of the respective synchronization rule. Executing this NTL rule, the synchronization engine essentially executes all synchronization blocks of that rule. The precise way of how the synchronization rules are executed is determined by the change propagation mode.

The easiest case for change propagation is of course disabled change propagation (*None*) since in this case, the synchronization does not have to perform any lifting. This means, the transformation can simply use the lense f to obtain the affected source elements and uses g to store them in the target model. Afterwards, it may forget about the synchronization block as it has been processed.

If the change propagation is set to *OneWay*, the synchronization engine must react on changes causing the selecting lense f to have a different result. Therefore, it applies the incrementalization system $\mathcal{I}$ which for a given source element $a \in A$ yields an incremental value for the result $\mathcal{I}(f)(\eta(a))$. The engine can then use the value of this element and proceed as if there was no change propagation switched on. However, the incremental value is stored and event handlers are registered for the event that it changes its value. In that case, these changes are transmitted to the RHS with a dedicated flag that the change must be processed regardless of the original synchronization direction. Both initially and in case of change notifications, the target value is stored using the Put method of g.

Formally, if we have again the situation of Figure 6.6 and $a \in A, c \in C$ such that $(a, c) \in \Phi_{A-C}$ and the system is in state $\omega \in \Omega$. If change propagation is enabled for the direction LHS to RHS in the current configuration, we actually store a reference to the incrementalized getter f. We refer to this variable as $\hat{f} := \mathcal{I}(f \nearrow)(\eta(a, \omega))$.

Now, consider that some global state change $\Delta\omega \in \Delta\Omega$ occurs. The first step to perform is to check whether this state change actually has any effect on the synchronization block, i.e. whether

$$value(\hat{f}) \neq value(apply(\hat{f}, \Delta\omega)).$$

In the implementation, this check is implemented by an event raised by $\hat{f}$ if a state change happened that changes the current value. Should that be the case, $\hat{f}$ is assigned the new value $apply(\hat{f}, \Delta\omega)$ which in the implementation

can be neglected since the objects realizing $\mathcal{I}(f \nearrow)$ are mutable and adjust their state automatically. Afterwards, the change is propagated by $\mathcal{R}_R$ as in the non-incremental case.

Compared to a repeated execution of the same model synchronization without change propagation or alternatively, a repeated execution of an unidirectional regular model transformation, the activated change propagation may lead to speedups. The speedup, however, depends on the used lenses, their performance characteristics and also the change sequences in which the models are changed. As we are striving for an extensible approach where developers can simply implement a lens, if they feel that the language misses one, a speedup can, however, not be guaranteed. Rather, the incrementalization is best effort.

The change propagation mode *TwoWay* works very similar, except that for both sides, the refined incrementalization functor $\mathcal{I}$ is applied and the $store$ transformation is used as a replacement of the PUT operation. The synchronization engine registers for change notifications on both incremental values and enforces the changes with direction *LeftToRightForced* or *RightToLeftForced*, respectively.

6.3.5. Inheritance and Superimposition

In practice, many metamodels use inheritance to avoid duplication of concepts in the metamodel. This poses a challenge for model synchronization because isomorphisms between more abstract concepts often are required to be broken down to isomorphisms that are more specific, i.e., have a smaller domain. Based on the more specific isomorphisms, more synchronization blocks may apply.

In our running example, consider the case that we added a composite structure both to finite state machines and Petri Nets. This means, states may either be simple states or composite states and likewise, places may either be simple places or composite places[50]. Clearly, we would like to keep synchronizing states of a state machine with the places of a Petri net, but in case the state is a composite state, a composite place shall be created and

[50] If this concept was added in only one of the metamodels, the information simply would not be propagated but no additional synchronization would be required.

the inner state machine should be synchronized with the inner Petri net of that composite place. However, one would still like to continue using the abstract isomorphism between states and places, but be able to refine this isomorphism in case of more specific elements.

In the implementation, we use a concept we call synchronization rule instantiation, very similar to the transformation rule instantiation concept used in NTL [92, 106]. Thus, whenever the synchronization engine is asked to create a new element for a given synchronization rule, it looks out for instantiations of that synchronization rule to create the element, because the target concept may also be different according to the true type of the source element. In case of the example above, a composite place should be created if the source model element was a composite state. Such a rule instantiation may be mandatory in case the targeted class is abstract.

To implement synchronization rule instantiation, we simply reuse the rule instantiation concept of the underlying NTL transformation rules. That is, if a synchronization rule $\Phi_{concrete}$ is instantiating a synchronization rule $\Phi_{abstract}$, then simply $\Phi^{\leftarrow}_{concrete}$ instantiates $\Phi^{\leftarrow}_{abstract}$ and $\Phi^{\rightarrow}_{concrete}$ instantiates $\Phi^{\rightarrow}_{abstract}$. Note that rule instantiation can be stacked, i.e., there may be another rule that instantiates $\Phi_{concrete}$ for more concrete classes. With this feature, also more complex inheritance hierarchies can be supported. In TGGs, rule refinements are an equivalent concept.

Furthermore, we also adopted the superimposition implementation of NTL, which in turn is adopted from Atlas Transformation Language (ATL) [213]. That is, synchronization rules may be easily overridden in refined model synchronizations, one may create a library of synchronization rules or create synchronization rule templates[51]. In particular, the considerations on inherited modularization concepts as described in previous work can be reused, but it is yet unclear to which extend this is useful also for incremental and bidirectional model synchronizations. However, a detailed analysis is out of the scope for this thesis and subject to future work.

[51] We have not yet used synchronization rule templates in practical use cases, yet. We suspect that they are harder to create than transformation rule templates because there is fewer support for abstract incrementalizable lenses than for regular abstract methods.

6.4. Synchronization of Finite State Machines and Petri Nets

In this section, we apply our concepts to the motivational example of synchronizing finite state machines with Petri nets. In parallel, we also describe how the proposed model synchronization would be specified in TGGs in order to draw a comparison. Because there is a textual language available that is easier to compare with and our impression is that it is the most actively developed TGG tool, we have chosen eMoflon [8] as concrete language. However, the results should be similar to any other TGG tool as well.

Like a model transformation in NMF Transformations that consists of multiple transformation rules represented by public nested classes inheriting from a `TransformationRule` base class, model synchronizations of NMF Synchronizations consist of synchronization rules. These synchronization rules implicitly define two transformation rules for NMF Transformations, one for each direction. A minimal example for a model synchronization is therefore depicted in Listing 6.3.

```
1 public class FSM2PN : ReflectiveSynchronization
2 {
3   public class AutomataToNet : SynchronizationRule<FiniteStateMachine,
        PetriNet> {}
4 }
```

Listing 6.3: A model synchronization in NMF Synchronizations

This defines the isomorphism $\Phi_{AutomataToNet}$ between finite state machines and Petri nets but without any synchronization block. Synchronization rules in NMF Synchronizations define the LHS and RHS model elements they operate on through the generic type arguments of the `SynchronizationRule` base class they need to inherit from. Details of the execution semantics such as in particular synchronization blocks are specified by overriding certain virtual methods.

In particular, unlike eMoflon, we do not require a correspondence declaration. To declare the used metamodels, it suffices entirely that the compiler of the host language knows the metaclass implementations. The latter may be either in the same assembly or in a referenced assembly.

```
1  public override void DeclareSynchronization()
2  {
3    SynchronizeMany(SyncRule<StateToPlace>(),
4      fsm => fsm.States, pn => pn.Places);
5    SynchronizeMany(SyncRule<TransitionToTransition>(),
6      fsm => fsm.Transitions, pn => pn.Transitions.Where(t => t.To.Count > 0));
7    SynchronizeMany(SyncRule<EndStateToTransition>(),
8      fsm => fsm.States.Where(state => state.IsEndState),
9      pn => pn.Transitions.Where(t => t.To.Count == 0));
10   Synchronize(fsm => fsm.Id, pn => pn.Id);
11 }
```

Listing 6.5: The DeclareSynchronization method of AutomataToNet

The most important method to override in the definition of a synchronization rule is the method to determine when an element of the LHS should match an element of the RHS. For the `AutomataToNet`-rule, we simply return true since both RHS and LHS model elements are the root elements of their respective models and should be unique.

```
1  public override bool ShouldCorrespond(FSM.State left, PN.Place right,
       ISynchronizationContext context)
2  {
3    return left.Name == right.Id;
4  }
```

Listing 6.4: Definition that states and places should correspond based on their names

Other synchronization rules may have other strategies. For example, the correspondence of the `StateToPlace`-rule is based on comparing the names as depicted in Listing 6.4.

The second most important method to override is the `DeclareSynchronization` method. Here, we define what synchronization blocks the synchronization rule consists of. The `DeclareSynchronization` method of *AutomataToNet* looks as depicted in Listing 6.5.

The specification of synchronization blocks follows the *Object Scoping* design principle for internal DSLs [67, p.385]. The statements in Listing 6.5 create synchronization blocks: Lines 3 and 4 create the synchronization block we depicted earlier in Figure 6.7. When handling the synchronization of a finite

state machine with a Petri Net, the synchronization engine should establish correspondences between the states and the places using the `StateToPlace` rule, synchronizing the states of the finite state machine with the places of a Petri Net. This synchronization rule is straight forward, matches states and places based on their names (cf. Listing 6.4) and synchronizes them afterwards. For a given state of a state machine, the synchronization engine only looks for corresponding places in the `Places` reference of the corresponding Petri Net.

The advantage of synchronization blocks over eMoflon is here that the specification can be much more concise. Consider for example the synchronization block in Lines 3 and 4, also depicted in Figure 6.7. This synchronization block is able to repair inconsistencies arising from adding or removing states from a state machine. TGGs usually require an entire rule for such a specification which is usually does not fit in a single line of code[52]. In eMoflon, this rule consists of 36 lines. These savings are possible because a lot of declarations have to be done explicit in a TGG rule, while they can be inferred in NMF Synchronizations because of the synchronization rule a synchronization block belongs to.

Please note that the lenses at the Petri Net side violate the PutGet law in case a new transition has to be added because the count of the `To` collection is not reversible. The `Where`-operator is currently implemented to silently ignore this inconsistency and we oblige this to the transformation developer to take care about such cases. This is fine for all the cases that we have come across but if this should not match the transformation developers needs, he can simply override this behavior by extending the respective lens. Because it is usually only the Put operation that is problematic, the language also has some overloads to provide a custom Put method.

Similarly, the transitions of the finite state machine should be matched with the transitions of the Petri Net but only with those that have at least one target place. Therefore, lines 5 and 6 of Listing 6.5 create the synchronization block depicted in Figure 6.9. Note that although a synchronization block is an octuple, only three elements must be specified by the developer. The base synchronization rule and all of the types can be inferred. The developer only specifies the target lens and the two selectors.

[52] We ignored the linebreak in the Listing because editors usually allow 85 characters or more in a single line

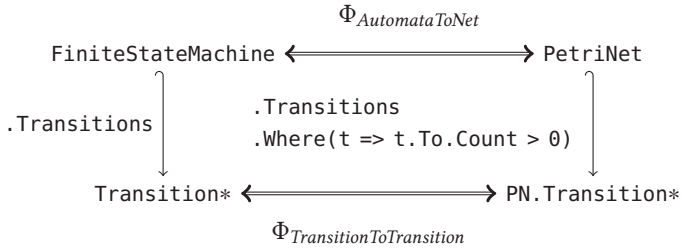


Figure 6.9.: Synchronization of the transitions of a finite state machine with the transitions of a Petri net

In particular, the synchronization block depicted in Figure 6.9 implies that if a new transition is added to the Petri Net transitions or an existing transition is assigned a first target place, then the synchronization engine will try to match this transition to an existing finite state machine transition. If conversely, a transition is added to the finite state machine, the synchronization engine will add the corresponding transition to the Petri Net, hoping that it satisfies the condition that the count is greater than zero. To find the corresponding transition on the respective other side, the `ShouldCorrespond` method depicted in Listing 6.6 is used.

This method uses the trace abilities of NMF Transformations that is still accessible in NMF Synchronizations, i.e. it accesses the corresponding place for a given state in the transformation rule $\Phi^{\rightarrow}_{StateToPlace}$ from LHS to RHS and uses it to decide whether the transitions should match. This trace entry exists regardless of the synchronization direction as the synchronization engine always creates both trace entries.

In eMoflon, the synchronization blocks to synchronize the transitions of a finite state machine as well as the synchronization blocks to synchronize the start and end states of such a transition all can be expressed in a single TGG rule, plus a second rule if self-transitions should be supported. In terms of lines of code, however, these rule still have 57+51 lines whereas the implementation in NMF Synchronizations requires 24 lines for the synchronization rule plus one for the synchronization block in the $\Phi_{AutomataToNet}$ synchronization rule. However, the latter also propagates partial changes

```
1  public class TransitionToTransition : SynchronizationRule<FSM.ITransition, PN
       .ITransition>
2  {
3    public override bool ShouldCorrespond(FSM.Transition left, PN.Transition
         right, ISynchronizationContext context)
4    {
5      var stateToPlace = SyncRule<StateToPlace>().LeftToRight;
6      return left.Input == right.Input
7        && right.From.Contains(context.Trace.ResolveIn(stateToPlace, left.
             StartState))
8        && right.To.Contains(context.Trace.ResolveIn(stateToPlace, left.EndState
             ));
9    }
10   public override void DeclareSynchronization()
11   {
12     Synchronize(t => t.Input, t => t.Input);
13     Synchronize(SyncRule<StateToPlace>(),
14               t => t.StartState,
15               t => t.From.SingleOrDefault());
16     Synchronize(SyncRule<StateToPlace>(),
17               t => t.EndState,
18               t => t.To.SingleOrDefault());
19   }
20 }
```

Listing 6.6: Matching transitions

while the respective eMoflon rule would propagate everything at once or not at all[53]. Which of these strategies is better depends on the application.

Lines 7–9 of Listing 6.5 create the synchronization block depicted in Figure 6.10 and indicate that the remaining transitions should be synchronized with the end states of the state machine. The symmetric correspondence check fails in this case because the synchronization engine will look for a suitable state in the end states of the machine. If the state is not yet marked as an end state, the synchronization engine will not find it. Thus, we have to override this behavior and particularly look for the state which is corresponding to the transitions origin.

[53] We believe that a partial propagation is also possible in TGGs but requires many more rules

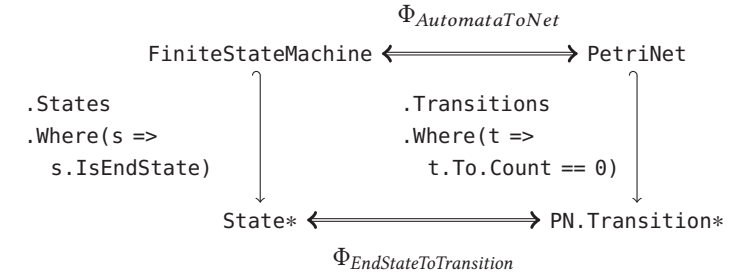


Figure 6.10.: Synchronization of the end states of a finite state machine with swallowing transitions of a Petri net

```
1 public override void DeclareSynchronization()
2 {
3   SynchronizeLeftToRightOnly(SyncRule<StateToPlace>(),
4     state => state.IsEndState ? state : null,
5     transition => transition.From.FirstOrDefault());
6 }
```

Listing 6.7: One way synchronizations

Next, it is necessary to connect or disconnect the Petri Net transition to the correct place. This only has to be done in the LHS to RHS direction since this information is already encoded in the `IsEndState` attribute in the finite state machine state. We have to limit the scope of this synchronization job because the synchronization initialization otherwise raises an exception since the conditional expression of the LHS is not a lens (we do not detect that there is a partition of values for true and false branch). This is depicted in Listing 6.7.

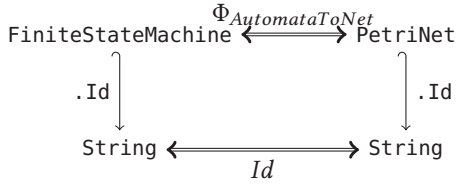


Figure 6.11.: Synchronization of the names of a finite state machine and a Petri net

Line 10 in Listing 6.5 specifies that the identifiers of both finite state machine and Petri Net should be synchronized. This creates a simple synchronization block as shown in Figure 6.11. The dependent synchronization rule is simply the identity on strings. In the syntax, this is expressed by omitting the synchronization rule. This means that both passed λ-expressions must have the same result type such that the synchronization engine may use the identity as isomorphism.

In eMoflon, the synchronization of end states and respective transitions also requires a single TGG rule that consists of 48 lines where the solution in NMF Synchronizations requires a synchronization block in $\Phi_{AutomataToNet}$ and a synchronization rule $\Phi_{EndStateToTransition}$ consisting of 21 lines.

The presented synchronization is a bijection, i.e. there is no information loss. In case there was, for example if we introduced multiple types of places (for example queueing places) with no equivalent information in the finite state machines, the synchronization would still be exactly the same, similar to the synchronization mode of eMoflon. The only difference would be that the information about the type of a place would not be transmitted to the state machine. In case the user adds a state to the state machine, the default place type would be used[54]. The information of the type of existing places is retained because NMF Synchronizations first tries to reuse an existing place to create a correspondence. This is decided based on the `ShouldCorrespond` method, so in this case based on the place name. More complex heuristics are possible but must be implemented by the transformation developer.

The entire synchronization between finite state machines and Petri nets in NMF Synchronizations consists of 4 synchronization rules in a single file with 92 lines of code in the usual C# coding style[55]. A functionally equivalent[56] implementation in eMoflon required 5 TGG rules 217 lines in total[57], plus a correspondence definition with 32 lines. The coding style in both languages is similar to some degree such that these numbers are roughly comparable, even though the lines in the NMF Synchronizations solution

[54] If the type for places was abstract, the synchronization would have to be changed to specify a default place type for this case.
[55] This includes 14 blank lines and 28 lines with only braces.
[56] Up to propagation of partial changes as discussed above
[57] This includes 41 blank lines and 47 lines with only braces.

tend to contain more words and more boilerplate code as for example method signatures of overridden methods must be repeated.

Both solutions are also available online[58].

Our synchronization language is extensible since transformation developers may easily implement new lenses as discussed in Section 6.3.1 and store them anywhere. eMoflon also allows the developer to extend the transformation framework but this requires the developer to switch the language and implement the extension in pure Java. Here, internal languages such as NMF Synchronizations have the benefit that developers do not have to switch the language as the integration of host language code usually can be done much more easily. In particular, as discussed in Section 6.3.1, extensions are simply methods with an annotation. These extensions can be made directly at the synchronization rule, in case it is only used in one place, or can be extracted into a library.

6.5. Summary

In this chapter, we introduced the concept of synchronization blocks, a formalism to describle declarative, incremental and bidirectional model synchronization that is suitable to be implemented in textual transformation languages. The formalism is sound in that we can prove that any inconsistencies can be repaired in both directions and the repair operator for both directions is hippocratic.

Further, we presented NMF Synchronizations, an internal DSL for bidirectional model transformation and synchronization that implements incremental model transformation using synchronization blocks in an internal DSL hosted in C#.

Besides the formal properties and the friendlyness for an implementation in an internal DSL, we think that a major advantage is that the formalism and the implementation in NMF Synchronizations unify the different types

[58] eMoflon rules: `https://github.com/NMFCode/SynchronizationsBenchmark/tree/master/eMoflon/FiniteStatesToPetriNets/src/org/moflon/tgg/mosl/rules`, NMF Synchronizations: `https://github.com/NMFCode/SynchronizationsBenchmark/blob/master/Transformations/SynchronizationsImplementation.cs`

of model transformation problems. In particular, the formalisms tackles RQ III.1, RQ III.2 and RQ III.3 alike. Besides exchanging a lens with a normal morphism[59], the only difference is in the way the transformation is executed. This makes it easier to extend an unidirectional model transformation into a bidirectional one and also makes it easier to mix these two types of transformation paradigms.

The flexibility of our solution is even more than the variety of research questions in RQ III: NMF Synchronizations is able to deduce 18 different operation modes based on a single specification. These synchronization modes are entirely symmetric, because synchronization blocks like TGGs draw no conceptual difference between LHS and RHS of a bidirectional transformation.

However, a formalization is only as good as it can be applied to practical problems and thus, we need to explore the practical applicability in Chapter 9.

[59] The research presented in Section 9.6 also indicates that this difference is characteristic as a large part of the most commonly used unidirectional model transformation language can be directly translated to synchronization blocks.

Part IV.

Meta-metamodel Extensions to Simplify Model Analyses

7. Using type system guarantees for model analyses

The goal of Part II was to improve the scalability of model analyses. The evaluation in Chapter 9 also shows good results but nevertheless, the analyses still need to be checked. However, we found that several analyses are not at all specific to a domain but rather compensate for accidental complexity in the metamodel. This accidental complexity is often due to inappropriate modeling facilities given by the metamodel. Several of such analyses can be made superfluous by extending the expressiveness of the meta-metamodel. In that case, the analysis result can be guaranteed statically and therefore, the analysis does not require *any* resources at runtime.

The contents of this chapter have been submitted to the MODELSWARD 2018 conference together with Kiana Busch and Robert Heinrich [99]. The contribution from the author of this thesis is the concept of refinements and structural decomposition, Kiana Busch and Robert Heinrich provided the case study.

The analyses we are interested are analyses that check whether models are consistent across multiple levels of abstraction: The metamodel defines the level of abstraction followed in the system model. However, it is often a challenge to choose the most appropriate level of abstraction for such a metamodel. If the metamodel is too general, it may easily allow instance models that do not correspond with the real system. In such case, model validation rules may help to reduce this risk. If features are specified in too specific subclasses, it gets hard to specify analyses because of case distinction. Therefore, it is often necessary to model an information on multiple abstraction levels simultaneously. However, this introduces the problem that the information has to be consistent. Therefore, one typically aims that a modeler only enters the information once and this information is used on all applicable levels of abstraction, consistently.

To compensate for this problem, the UML [156] has introduced concepts of refinements between associations in the form of subsetting, specialization and redefinition. This specification is also reused in the CMOF standard [151] but disallowed in EMOF and its most common implementation Ecore. Though the semantics of these declarations is not clear from the standard, several works [155, 48, 80] have defined semantics of these definitions and implemented them in OCL constraints. However, the interaction of these subsetting, specialization and redefinition with other constraints such as multiplicity constraints have been a source of various problems [144, 143] as the semantics turns out to be inconsistent.

Furthermore, OCL constraints need to be checked. This has mainly two disadvantages: It takes time and constraints may be temporarily violated. As consequences, we see a higher overhead for model manipulation tasks (because the constraints must be checked) and a higher implementation effort for model analysis tasks (because the case that a constraint is violated has to be taken into account).

In commonly used meta-metamodels such as Ecore, the most popular workaround is to create a feature in the most general concept and create derived features in more specific classes. An example for this is in Ecore itself where `ETypedElements` simply have a type. More specific classes such as `EAttribute` or `EReference` inherit this reference, even though they could be more specific: The type of an attribute must be a data type or enumeration while a reference always must be typed with a class. The metamodeler has to enforce this using a model validation constraint.

In an industrial context such as automated production systems, we see a very similar effect where sensors are generally equipped with a power supply but some kinds of sensors only accept certain power supplies. Here, one would like to gain the expressiveness to specify the correct power supply type without having to use case distinctions in the analyses. Because a wrong power supply may have dramatic consequences in the physical sensor, this constraint should be enforced as early as possible. On the contrary, we also see the case that the exact type of the sensor is not known, for example because the sensor is supplied by a vendor. Therefore, very specific model elements may be mixed with more general model elements and hence, the information must be available on multiple levels of abstraction simultaneously.

Existing approaches to simplify the metamodel in this regard such as VPM [208], Deep Modeling tools [11, 54] or CORE [181] require completely new

modeling paradigms that mostly break existing tools. However, the availability of stable tools is one of the major factors for the lack of industry-adoption of MDE [187, 153]. In addition, Meyerovich et al. [149] have shown that most developers only change their primary language when either there is a hard technical project limitation or there is a significant amount of code that can be reused. Therefore, we suspect that many metamodelers would still like to use their usual meta-metamodel.

In this chapter, we propose a formal approach how refinements of associations can be implemented in a non-invasive way into an Ecore-like meta-metamodel. The proposed approach is able to guarantee the correctness of the refinement, i.e. the consistency of the model in multiple levels of abstraction, through guarantees of the target platform type system. In the power supply example above, the modeler gets an immediate feedback in the form of an exception as soon as he tries to add a non-appropriate power supply to a sensor. Information that is exposed on multiple levels of abstraction simultaneously only has to be specified once, at the most concrete level of abstraction. We implemented our approach in NMeta and discuss its advantages over alternative metamodel fragments in the domain of production automation.

The remainder of this chapter is structured as follows: Section 7.1 introduces the domain of production automation more closely and defines the running examples. Section 7.2 defines the concepts of refinements and structural decomposition and applies them to the running example. Section 7.3 explains how refinements and structural decomposition are implemented in the NMeta meta-metamodel. Section 7.4 presents our approach how to guarantee the correctness of refinements through type system guarantees. Section 7.5 demonstrates how these concepts can be implemented using NMeta and compares it with alternative modeling strategies. Finally, Section 7.6 summarizes insights and achievements of this chapter.

7.1. Running Examples

This section introduces the domain of automated production systems that is used as a running example in the remainder of this chapter.

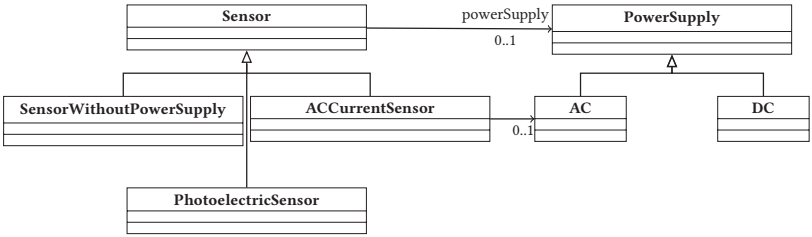


Figure 7.1.: Sensor example

We use this system hereafter as a running example to illustrate the idea of our approach. The domain of automated production systems involves software, as well as mechanic and electric parts. For the running example we use the example of sensor and power supply. In general, a sensor has a power supply. Several types of power supply exist, for example Alternating Current (AC) or Direct Current (DC). Additionally, there are several types of a sensor such as photoelectric, capacitive, or AC current sensors. Consider the example, that an AC current sensor must have an AC power supply, as illustrated in Fig 7.1. This fact must be specified either through an OCL constraint or using derived features in an Ecore model. However, we cannot be sure whether the constraints are enforced. The metamodel itself is able to express that an AC current sensor has a DC power supply. Our goal is to enforce such constraints already through the underlying type system. For example, we can guarantee that a model where an AC current sensor has a DC power supply, may not exist. Consider the example of a sensor, that does not need any power supply (`SensorWithoutPowerSupply`). For example, a surface acoustic wave sensor [164] obtains its energy from piezoelectric and pyroelectric effects. We want to ensure that the type system does not allow to model a case in which a sensor with no power supply has an AC or DC power supply.

On the other hand, we often face the problem that we do not know the details of every element that should appear in the model. Therefore, it is desirable to keep classes such as `Sensor` non-abstract, in order to model that we have a sensor where there is no information on technical details, yet still, we want to be able to reference its power supply.

For references with higher cardinality, we face the problem that very general references can be decomposed in more specific subclasses. As an example,

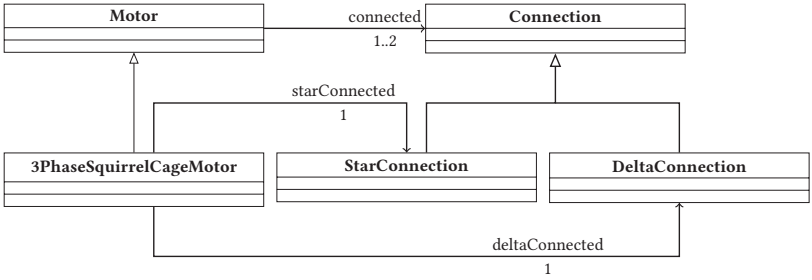


Figure 7.2.: Star and delta connections in motors

consider the power supply of a motor in a star or delta connection, as illustrated in Fig. 7.2. The star-connected motors have a central point where the similar ends of the wires are connected whereas in the delta-connected motors the opposite ends of wires are connected. Thus, the delta connection results in a higher torque and a higher motor speed. However, some motors require both connections. For example, a three phase squirrel cage motor[60] has to be started in a star connection. After the normal speed is reached, it has to be switched from a star to a delta connection.

7.2. Refinements and Structural Decomposition

In this section, we want to discuss and formalize the notion of structural decomposition.

To do this, we again apply the theory of mutable type theories introduced in Chapter 3. In the running example of this chapter, the features are the attributes and references in the metamodel. For example, we may consider the reference *powerSupply* a lens `Sensor` ⇀ `PowerSupply`. Its GET method returns for a given sensor the assigned power supply at a given model state. The PUT modifies the model state by setting the power supply of that sensor to the desired value. The PUT may also perform additional side-effects, for

[60] `http://www.pcbheaven.com/userpages/check_the_windings_of_a_3phase_ac_motor/`, retrieved 10 Jul 2017

example in order to also set the opposite feature or to cascade a model element deletion, in case a model element has been removed from its container.

The domain of *powerSupply* in this context is the set of allowable power supplies for each sensor in each model state. For example, for an `ACCurrentSensor` element, this set should only contain instances of `AC`.

Refinements and structural decompositions can be easily modeled in a MTC as a relation between morphisms, as shown in the following definitions.

Definition 46 (Structural Decomposition of Lists)**.** Let A and B be types. A set of features $f_1, \ldots, f_n : A \hookrightarrow \mathcal{K}(B)$ for types A and B and an $n \in \mathbb{N}$ is a *structural decomposition* of a feature $f : A \hookrightarrow \mathcal{K}(B)$ if we have that for each global states $\omega, \tilde{\omega} \in \Omega$ and $a \in A$ that

$$f \nearrow (a, \omega)(\tilde{\omega}) = f_1 \nearrow (a, \omega)(\tilde{\omega}); f_2 \nearrow (a, \omega)(\tilde{\omega}); \ldots; f_n \nearrow (a, \omega)(\tilde{\omega});.$$

We say that f is made of $f_1, \ldots, f_n$ and call the f_i components of a composition f.

Since there is an embedding from $A \times \Omega \to B$ into $A \times \Omega \to \mathcal{K}(B)$, we will also allow the features used for decomposition to be single-valued where we depict an element $\perp \in B$ that corresponds to an empty string in $\mathcal{K}(B)$. Likewise, we allow compositions to be single-valued. In this case, the value of the composition has to match the only component value that is not `null`.

Likewise, we define structural decomposition also for multisets.

Definition 47 (Structural Decomposition of Multisets)**.** Let A and B be types. A set of features $f_1, \ldots, f_n : A \hookrightarrow \mathcal{M}(B)$ for types A and B and an $n \in \mathbb{N}$ is a structural decomposition of a feature $f : A \hookrightarrow \mathcal{M}(B)$ if we have that for each global states $\omega, \tilde{\omega} \in \Omega$, $a \in A$ and $b \in B$ that

$$f \nearrow (a, \omega)(\tilde{\omega})(b) = \sum_{i=1}^{n} f_i \nearrow (a, \omega)(\tilde{\omega})(b).$$

Example 32. In our running example, the connections of a three phase squirrel cage motor can be structurally decomposed into the star connection and the delta connection: At any time, we would like the *connected* reference of a given three phase squirrel cage motor to only consist of the model

elements referenced by the *starConnected* reference and the *deltaConnected* reference.

The next proposition shows that the definitions are consistent.

Proposition 33. Let A and B be types. Let $f_1, \ldots, f_n : A \hookrightarrow \mathcal{K}(B)$ be components of $f : A \hookrightarrow \mathcal{K}(B)$. Then we have that for every $\omega, \tilde{\omega} \in \Omega$, $a \in A$ and $b \in B$ that

$$\iota_B^{\mathcal{K}}(f \nearrow (a, \omega))(\tilde{\omega})(b) = \sum_{i=1}^{n} \iota_B^{\mathcal{K}}(f_i \nearrow)(a, \omega)(\tilde{\omega})(b).$$

Proof. In the above situation, we have that

$$\begin{aligned}
&\sum_{i=1}^{n} \iota_B^{\mathcal{K}}(f_i \nearrow)(a, \omega)(\tilde{\omega})(b) \\
&\quad = \sum_{i=1}^{n} |\{j \in \{1, \ldots, n_i\} | (f_i \nearrow (a, \omega)(\tilde{\omega}))_j = b\}| \\
&\quad = |\{(i, j) \in \{1, \ldots, n\} \times \mathbb{N} | j \leq n_i \wedge (f_i \nearrow (a, \omega)(\tilde{\omega}))_j = b\}| \\
&\quad = \iota_B^{\mathcal{K}}(f \nearrow (a, \omega))(\tilde{\omega})(b).
\end{aligned}$$

□

We combine structural decomposition with a notion of refinement as per the following definition:

Definition 48 (Refinement)**.** Let A, B, $\bar{A}$ and $\bar{D}$ be types with $\bar{A} \preceq A$ and $\bar{B} \preceq B$. Further, let $f : A \hookrightarrow B$ and $f' : \bar{A} \hookrightarrow \bar{B}$ be lenses. We say that f' is a refinement of f if $f \nearrow$ and $f' \nearrow$ are the same on $\bar{A} \times \Omega$ and the setters are the same for elements of $\bar{A}$, i.e. the following equations hold for all $a \in \bar{A}$, and $\omega \in \Omega$:

$$\begin{aligned}
f' \nearrow (a, \omega) &= f \nearrow (a, \omega) \\
\mathcal{D}(f' \searrow) &= \mathcal{D}(f \searrow) \cap \bar{A} \times \bar{B} \times \Omega \subset \bar{A} \times \bar{B} \times \Omega
\end{aligned}$$

and if $(a, b, \omega) \in \mathcal{D}(f' \searrow)$, then we have that

$$f' \searrow (a, b, \omega) = f \searrow (a, b, \omega).$$

Example 33. In the running example, the sensor type `ACSensor` should always have an `AC` power supply, as discussed above. Therefore, we may think of a reference *acPowerSupply* : `ACSensor` $\hookrightarrow$ `AC` that refines *powerSupply* because for all elements of `ACSensor`, the power supply is in fact an `AC` element.

Remark 35. Example 32 actually combines structural decomposition and refinements: Using only structural decomposition, we could model that a three phase squirrel cage motor has a separate star connection and delta connection. Together with refinements, we may express the star connection cannot be an arbitrary connection, but must be a `StarConnection`, likewise for the delta connection.

Remark 36. The advantage of the information that a feature $f' : \bar{A} \hookrightarrow \bar{B}$ refines a feature $f : A \hookrightarrow B$ is twofold. If we want to use the GET of f for an instance from which we know that it is an element of $\bar{A}$ (e.g. through static analysis), then the refinement gives us that the result will be in $\bar{B}$. In implementations, this can aid the static analysis of the code. Conversely, if we want to use the PUT of f, then we know that this operation is only defined if the element to set is in $\bar{B}$. This is important to know because most implementation type systems cannot express that a function is only defined for a subset of its parameter spaces. Thus, $\mathcal{D}(f)$ will typically be implemented by throwing an exception if the given parameters are not in the domain (depending on the current state). Here, the (static) knowledge that f' refines f means that we can statically proof that the setter of f will fail if the value of a setter is not of type $\bar{B}$. This may result in a compiler warning.

Remark 37. An important special case here is the refinement by a constant reference $g \equiv b$ for some constant element $b \in \bar{B}$[61]. Usually, constant features are not explicitly modeled as they do not contain any information specific to an instance, but in combination with a refinement, they may carry

[61] Here, the $\equiv$ symbol means that the getter function always returns the same element b and the setter is only defined where necessary for (PutGet), i.e., if the value is b.

information that is known for some subtypes, but not in the general case for a given type A.

Example 34. For the last remark, consider again the passive sensor that does not require a dedicated power supply. Modeling a connection of such a sensor with a physical power supply cannot reflect the physical sensor and therefore should be forbidden. Using refinements, we can model this situation with a constant reference $passiveSupply$: `PassiveSensor` ⟶ `PowerSupply` that always returns ⊥.

Remark 38. An effect of refinements is that in some operations may raise exceptions in subtypes where they do not raise exceptions in their parent classes. According to the Liskov' Substitution Principle (LSP) [140], if type S is a subtype of T, then any property provable about an instance of T must be provable for an instance of S as well. The goal of this principle is to guarantee desirable properties of a program like correctness or termination when instances of T are replaced by instances of S. While this strong behavioral subtyping is undeniably beneficial for operations that are side-effect free, it limits the application of objects when the state of this object must be changed such as changing the value of an attribute or reference. In this case, sometimes the complete state of the model element must be considered as the true model type may have introduced additional validation constraints.

For example, meanwhile an `AC` sensor is a sensor – expressed by an inheritance relation of the `Sensor` class – and therefore theoretically may have an arbitrary power supply, attaching a DC power source to such a sensor often causes a failure[62]. However, the knowledge that a AC power supply is required is not necessary when querying the power supply of a sensor, only if the sensor should be notified.

As a reason, the power supply inheritance hierarchy is covariant to the inheritance hierarchy of the sensors. This is not problematic for methods like queries that return covariant instances such as more specific power supplies. It does become a problem when changing the state of the sensor since strong behavioral subtyping demands a contravariant inheritance hierarchy but the sensor's power supply is an instance of a subtype of the `PowerSupply` class.

[62] A failure also occurs when a AC sensor is attached to a wrong type of AC power supply, for example with a higher voltage than supported. Therefore, one may want to model the different types of AC power supplies on a further level of detail.

The solution in classic object oriented programming would be to design the sensor class as an interface where the power supply can be queried from but cannot be modified. This way, being able to add a power supply to a sensor without an exception being thrown is not a provable property and the Liskov' Substitution Principle is maintained. However, we argue that this is an implementation detail that should not be part of the metamodel since the purpose of a metamodel is to describe the domain on a high abstraction level. This excludes implementation details to comply with the LSP.

7.3. Implementation in NMeta

We have implemented refinements and structural decomposition in NMeta through the additional references and classes that enable references to refine other references and likewise for attributes. Further, additional classes `ReferenceConstraint` and `AttributeConstraint` are added. The changes are depicted in Figure 7.3, highlighted with double lines.

If multiple references refine a reference, that reference is structurally decomposed and the components are refined[63]. Therefore, the refined reference must be declared in an ancestor of the current class and the reference type of the refinement reference must be a descendent of the refined references type. A reference may also refine a reference that is already refined in an ancestor class. In that case, the original reference is structurally decomposed by all references that refine the original reference. In other words, a decomposition is always scoped for a given class.

Additionally, the metamodeler can add a constant reference into this structural decomposition through a dedicated model element called `ReferenceConstraint`. This means that the reference is also refined by a constant model element or a collection thereof, in case the reference is typed with a collection. Only a single `ReferenceConstraint` is allowed per class and reference.

The class `ReferenceConstraint` is only necessary because NMeta has no support for derived features, yet, because it lacks a support for OCL. Using derived features such as in Ecore, one could use a derived reference instead

[63] This definition is consistent because a single feature is a structural decomposition of itself.

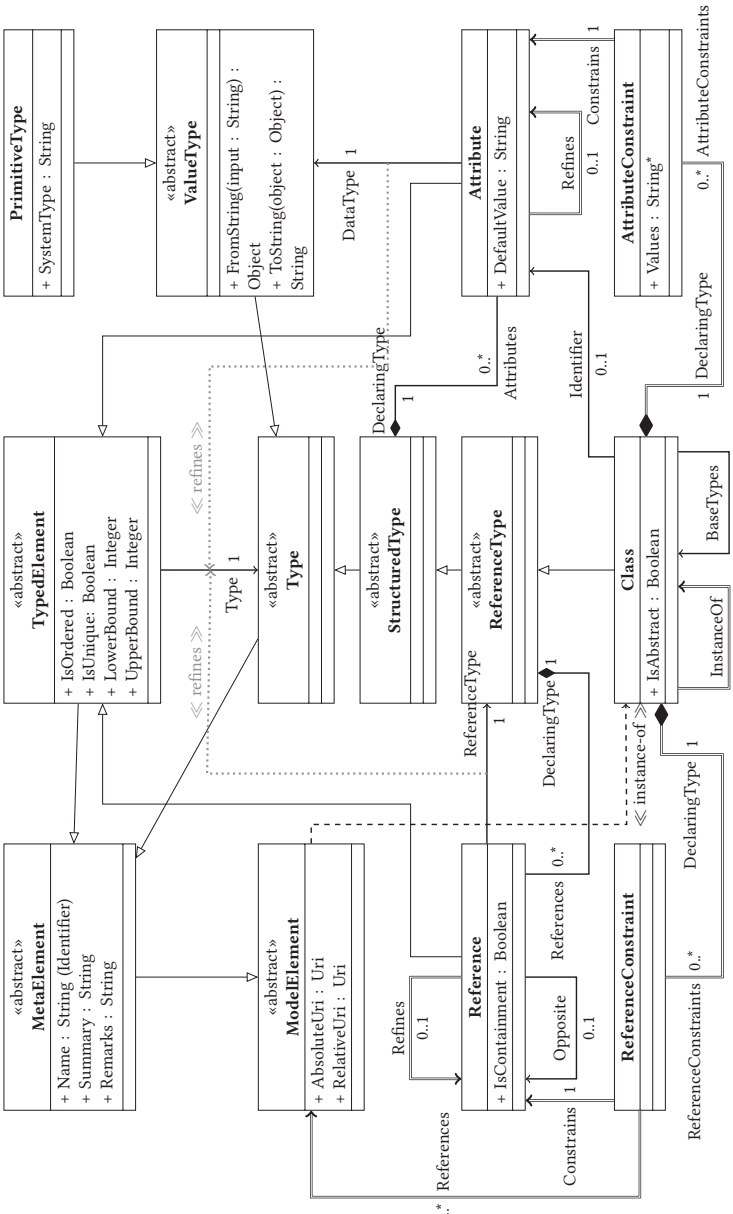


Figure 7.3.: NMeta meta-metamodel with extensions to support structural decompositions and refinements.

that simply returns a constant value, therefore making the `ReferenceConstraint` class obsolete.[64]

For attributes, structural decomposition works the same but the types must match exactly because the inheritance hierarchy of value types is not modeled.

For any composition of attributes or references, the multiplicity of the composition must be compatible with the multiplicity of the original feature. This means that the lower bound of the original feature must be smaller or equal to the sum of the lower bounds of components. Likewise, the upper bound of the original feature must be larger or equal to the sum of upper bounds of the components. Furthermore, we require that refinements of compositions are compositions. If a reference with an opposite is refined, we require that any refining reference has an opposite that refines the opposite of the original reference.

Reference refinement is used within NMeta for the `Type` reference that attributes and references inherit from `TypedElement` (the blue arrows in Figure 2.1). Similar to Ecore, a reference is only valid when the type assigned to it is a reference type, e.g., a class. Conversely, an attribute must be typed with a value type. In Figure 2.1, this relation is denoted with a dotted arrow from the `ReferenceType` or `DataType` reference of `Reference` and `Attribute` to the `Type` reference of `TypedElement`.

The semantic behind this assignment is that the `Type` of a `Reference` is refined by its `ReferenceType`. Conversely, if we set the `Type` of a reference, the setter internally sets the `ReferenceType`. However, this only works if the set value is an instance of `ReferenceType`. If it is not, an exception is thrown because a reference whose type is not a reference type cannot be valid. As a consequence, the validation that types of references must be reference types is already checked by the type system, making a constraint, e.g. written in OCL, obsolete. This semantics is very similar to UML redefinitions with the important difference that the `TypedElement` type does not have to be aware that its `Type` reference is refined in another class.

A reference may be constrained and refined at the same time. In that case, the reference consists of some referenced elements determined by the reference constraint and others determined by refinement references.

[64] The reason for NMeta to not support OCL is an incompatible code generation infrastructure.

Besides a reference of base classes, classes in NMeta are allowed to restrict inheritance to instances of a given class through the `InstanceOf` reference. This reference may only be specified for abstract classes. If a class *A* is an instance of class *B*, then only instances of *B* may inherit from *A*. Consider *a* an instance of type *C* which inherits from *A*. Because *B* was declared as an instance of *A*, *C* must be of type *B*. Thus, the type of *a* is an object of type *B*. Since this is known at compile-time, the generated code contains a refined method to obtain the model elements type of type *B*.

If the `InstanceOf` reference is left blank, this has the same effect as specifying that a class is an instance of `Class`. This is because the base class `ModelElement` is marked as an instance of `Class`. Moreover, when classes define an instance-of relation and one of its base class also specifies an instance-of relation, then the new instance-of class must be a subtype of the base class instance-of class. As an immediate consequence, all classes used in the instance-of reference must be subtypes of `Class`.

The class `DataType` describes simple structures of values such as vectors or complex numbers that are typically edited as a whole. The differentiation to normal classes, they are treated as values, are thus immutable and are not addressable through a global URI.

7.4. Code Generation

Similar to EMF, NMF provides a code generator for metamodels [96]. Because the meta-metamodel allows multiple inheritance, the code generator generates both an interface and a default implementation class for each class in the metamodel, again similar to the EMF code generator. To keep the generated code small, the code generator reuses default implementation classes for subclasses as much as possible.

For any attribute or reference (feature in the remainder), a property and a change event is generated. If the feature is not refined, this property is backed by a field. In case a feature is refined, a private getter and setter

implementation[65] is generated instead that composes or decomposes the property on the fly.

The consistent representation of information simultaneously modeled in multiple levels of abstraction is an immediate consequence: Any information is only stored on the lowest level of abstraction, only. Representations in higher abstraction levels are generated upon request.

As a consequence, an implementation of a metamodel class must not contain backing fields of a refined reference. Therefore, refinements impact the inheritance hierarchy of the implementation base class. In case a feature is refined, the code generator may no longer reuse any implementation class that contains a backing field for this feature.

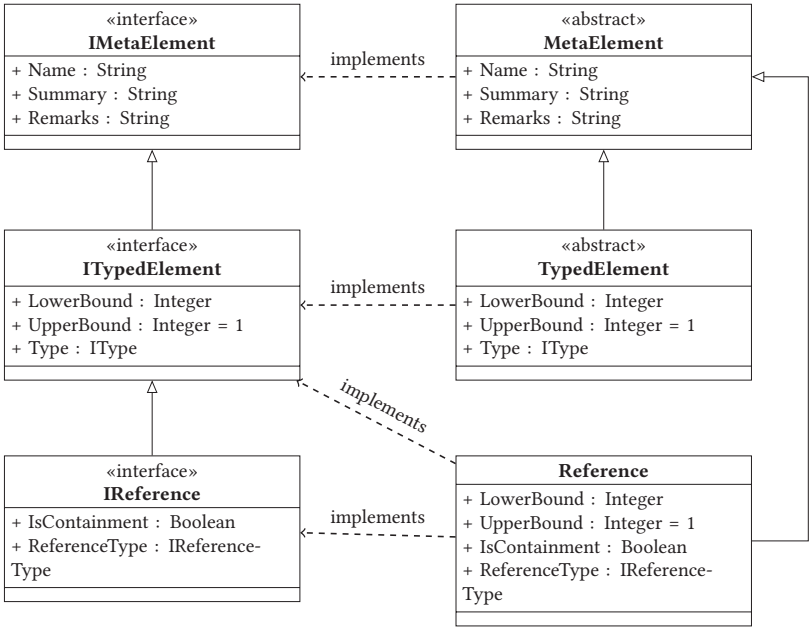


Figure 7.4.: Generated model representation class and interface for references

[65] .NET allows classes to privately implement an interface which means that the implementation is not visible from the class API but only through this interface.

For example, Figure 7.4 shows the interfaces and classes generated for metaclasses `MetaElement`, `Reference` and `TypedElement` of NMeta. For all classes, a class and an interface is generated. In accordance to .NET nomenclature, the generated interfaces are prefixed with the letter *I*. Because there is an inheritance relation between the metaclasses, the generated interface for `Reference` does inherit from the generated interface for `TypedElement`. However, the generated implementation type `Reference` does not inherit `TypedElement`, in order to avoid inheriting the `Type` reference backing field.

On the other hand, the code generator may still reuse the class `MetaElement` as a base class for `Reference` to avoid generating properties for name, summary and remarks again.

Furthermore, we may encounter diamond problems. If, for the sake of the example, a class inherited from both `Attribute` and `Reference`, the code generator must not reuse the implementation class of either `Reference` or `Attribute` because the decomposition of the feature `Type` is different to the one in `Attribute` or `Reference` where only one feature refines `Type`.[66]

As soon as an appropriate base class is found, we simply copy the generated code for the features that cannot be inherited and copy them into the generated type, as long as they are not refined.

To find such a base class, we propose the abstract algorithm depicted in Algorithm 1. It is essentially based on a reversed topological order of strongly connected components in a dedicated graph that is induced by inheritance relations and refinements. The result of Algorithm 1 for a class c, if not $\perp$, is a class c_b that

- is a base class of the class to be generated ($c \preceq c_b$).
- only contains properties that have not been refined by classes $\tilde{c}$ with $c \preceq \tilde{c} \prec c_b$.
- that does not contain a decomposition that is no longer valid. Here, an invalid decomposition refers to a decomposition of a feature f

[66] One may suspect that any class that inherited both from `Attribute` and `Reference` must cause an error because the sum of the upper bounds for the structural decomposition of the `Type` reference is 2 and thus greater than the upper bound of the refined `Type` reference. However, the implementation in NMF actually explicitly allows such a construct.

into $f_1; \ldots; f_n$ in the scope of a class $\tilde{c}$ with $c \prec \tilde{c}$, but f is decomposed in c by a larger list of features.

Algorithm 1 Find implementation base class

```
function AllFeatures(c)
    return ⋃_{c≤c_b} c_b.Attributes ∪ ⋃_{c≤c_b} c_b.References
end function
function Refinements(c)
    return {g | f ∈ c.Attributes ∪ c.References, f --«refines»--> g}
end function
function Edge(c_s, c_t)
    return c_s ⪯ c_t ∨ (Refinements(c_s) ∩ AllFeatures(c_t) ≠ ∅ ∧ c_t ⋠ c_s)
end function
function FindBaseClass(c)
    shadows ← ∅
    ancestors ← TransitiveHull(c, cl ↦ cl.BaseTypes)
    for all layer in ReverseTopologicalOrder(ancestors, Edge) do
        if |layer| = 1 ∧ layer ≠ {c} ∧
           shadows ∩ AllFeatures(layer[0]) = ∅ then
            return layer[0]
        end if
        for all l in layer do
            shadows ← shadows ∪ Refinements(l)
        end for
    end for
    return ⊥
end function
```

In Algorithm 1, the function AllFeatures simply computes the set of all attributes and references available in a given class, including inherited and transitively inherited. The function Refinements returns those attributes and references that are refined by attributes or references of the given class. More interesting is the function Edge that defines the edges in the graph the topological order is created for. This graph shall contain edge from a class c_s to a class c_t if the generated code for class c_s obsoletes the generated code for c_t. This may either be because $c_s \preceq c_t$ or because c_s refines a property of c_t. The latter case is not problematic for the case that $c_t \preceq c_s$, because in that case, the generated code for c_t is aware of this refinement.

The reversed topological order guarantees us that there is no incoming edge from ancestor classes not yet considered for the given class. It can be easily implemented by reversing the output of Tarjan's algorithm [199]. In Algorithm 1, we assume the latter to return a list of strongly connected components, each represented as set of classes.

The graph may contain cycles. Because inheritance is acyclic and we only allow features to refine features of base classes, such a cycle must come from a set of classes that refine features of a common base class, i.e. we are facing a diamond-shaped inheritance. Because the generated code for the bottom of the diamond must respect all refinements made in any of its base classes, no generated code for a class contained in a cycle must be reused. However, there still may be a common ancestor class whose features have not been refined, such as for example `MetaElement` in the example of `Reference`.

Applied to the class `Reference`, the reverse topological sort returns the strongly connected components {`TypedElement`}, {`MetaElement`}. `TypedElement` is not chosen because its property `Type` is refined but `MetaElement` is because it is the only element in its component and does not contain a refined property.

Because the class `Reference` does not inherit an implementation of `ITypedElement`, it implements this interface directly by duplicating the implementation of non-refined attributes and references. This code duplication is not problematic since the code is generated. In the metamodel, the duplication is avoided. The `Type`-reference is implemented in private where the getter simply returns the `ReferenceType` reference and the setter tries to cast the value appropriately and sets the `ReferenceType` reference, if applicable and throws an exception otherwise.

For decomposition, the setter will assign the value to the first component property that fits. This affects the default implementation class as any implementation class containing a backing field for a decomposed property must not be reused. However, it does not affect the generated interface. Thus, the substitution principle is maintained for any analysis that consumes the model and just relies on the interface.

Artifacts that modify model elements such as editors would rather operate on the real type of the model elements and therefore only see the public properties which are exactly the non-decomposed properties.

By default, the generated XMI code for a serialized model will not contain any information on decomposed features since they can be reconstructed by its components. If information about refinements is cut off (e.g. by exporting the metamodel to Ecore), the serialization simply needs to be configured to serialize also refined features and then tools not aware of structural decomposition are able to read the model just like any other model. Conversely, when reading the model, the refined features are just ignored in the deserialization such that models created by tools not enabled for structural decomposition can be loaded – the only problem here is that these tools may unnecessarily demand the modeler to specify features that are otherwise refined.

Therefore, existing tools not aware of structural decomposition and refinements can be reused without changes, though they may not offer the best convenience.

7.5. Alternatives to Model Production Automation

In this section, we show the applicability and advantages of using structural decompositions and refinements to model the domain of production automation. For this purpose, we take the running examples from Section 7.1 with their formal insights from Examples 32, 33 and 34 and discuss the implementation in NMeta in comparison to modeling alternatives supported by Ecore.

7.5.1. Sensors with an AC Power Supply

There are multiple options to model that sensors have a power supply but more specific kinds of sensors may have an AC power supply only using existing modeling technology:

Constraint + derived reference: We add a constraint that the power supply of an AC sensor must be an AC power supply. With this constraint, we can automatically check the model for consistency. If required in applications, we may add a derived reference that returns the power supply of the given sensor as an AC power supply.

The main problem that we see with this approach here is that the constraint is only enforced upon request, or the constraint needs to be checked before the model manipulation is performed. In the former case, the modeler explicitly has to validate the model in order to see that the constraint is broken. In the latter case, the constraint must be checked before any model manipulation which may be a costly operation. The type system and also the API allow the model to have an invalid reference. Furthermore, creating the constraint and the derived reference may be much more work than the assignment of a refinement.

Reference in concrete type + operation: Another possibility is to only create the reference in the specific class, in our case `ACCurrentSensor`. There, we already know that the power supply has to be a AC power source which also buys us type-system guarantees that the power supply is an AC source.

However, this approach is also problematic because it is unclear on which level of abstraction to set the power supply type. There might be a more concrete sensor type that requires an even more specific power supply type. Moreover, we also have other sensors than the AC sensors such as sensors bought by vendors or sensors that may accept different power supplies. As the generic `Sensor` class does not have a reference to the power supply, we have to create a subclass to specify the power supply type. Further, because we still want to capture the knowledge that every sensor has to have a power supply, we would need to add an operation to obtain the power supply of a concrete sensor. However, this is problematic since not all tools work with operations well. As an example, incremental tools cannot cope with operations as they cannot see inside the operation body.

In some scenarios, there is a further problem that one does not want to modify the base class just because of an additional constraint in a derived class, to avoid changing all the other subclasses. This is especially the case if the problematic reference is defined in an indirect ancestor such as for example the `eType` is defined in `ETypedElement` which is the base class of the base class of `EReference`.

Generic type: We could leave the concrete type of the power supply open and make the sensor type accept a type parameter. This leaves the decision for the used power supply type open until the sensor is used.

This version has a similar problem as the last one: We do not know whether there is a sensor that requires a more specific power supply type. Further, in an application, we have to know the power supply type. Otherwise, we must rely on Javas weak implementation of generics[67] that allows developers to essentially do not care about the concrete generic type argument as long as it respects some type condition. This is not possible on other platforms such as .NET that follow a hard implementation of generics[68]. Therefore, this modeling option makes a portability to other platforms impossible.

Subsetting: In UML 2, we may specify that the AC supply of an `ACCurrentSensor` must be a subset of the power sensors. As a consequence, the modeler has to assign the AC source of such an element himself and the source is not automatically assigned, unless the modeling system automatically sets the power source in case the modeler assigns the AC source. However, even in that case, we think it will be confusing for the modeler that there is a semantic difference between setting the AC source and setting the power source.

Specialization: According to Costal and others [48], specialization has the same semantics as subsetting and therefore, the same disadvantages apply.

Redefinition: UML redefinition of associations matches our notion of refinements. However, they are not available in EMOF and not implemented in metamodeling frameworks such as Ecore.

The modeling approach using refinements is depicted in Figure 7.5. We create an additional reference called *acPowerSupply* in the `ACSensor` class and set it to refine the original *powerSupply* reference[69]. As a result, the generated implementation class for `ACSensor` will not inherit the more general power supply field but includes a specific field to reference an `AC` element from which the *powerSupply* reference is populated upon request. Because the

[67] Information of a types generic type parameters is only available at compile-time

[68] The generic type parameters are still available and enforced at run-time. Type boundaries are only supported in interfaces if type parameters are either used only as inputs or only as outputs.

[69] NMeta does allow refined references to have the same name as the original reference but we use a different name for clarity.

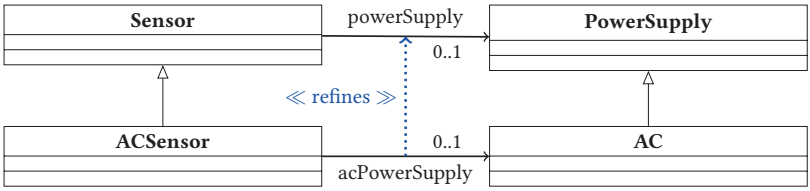


Figure 7.5.: Modeling that an `ACSensor` requires an `AC` power supply in NMeta. The inserted refinement is printed in blue.

type system of .NET does not allow this field to hold a reference to an object that does not fulfill the generated interface for `AC`, we can be sure that the power supply of an `ACSensor` instance is valid without checking any validity constraint. If the modeler tries to assign a DC power supply to such an element, he gets an immediate feedback, telling him that it is not possible to use this element as a power supply for an `ACSensor`.

7.5.2. Sensors without Power Supply

To model that sensors without a dedicated power supply do not have a power supply in traditional modeling, one would add a validity constraint saying that the *powerSupply* reference of a `PassiveSensor` must be empty, respectively `OclUndefined`. An alternative implementation could be that all instances point to the same power supply element. In this case, the path to this element has to be reconstructed in OCL which can be a brittle operation because OCL has no language feature to reference a static model element. Instead, one has to make sure the element can be uniquely identified based on an `allInstances` operation, the only operation that references model elements independently of the current context.

In both cases, the main drawback is that the constraint can be violated and must be enforced manually. An entirely different approach would be to omit the reference in the base class, but then, the same drawbacks as in Section 7.5.1 apply.

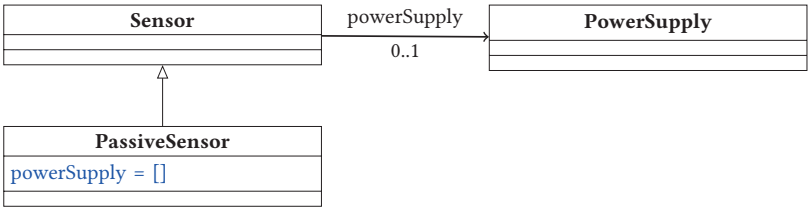


Figure 7.6.: Modeling that a `PassiveSensor` must not have a power supply in NMeta. The inserted reference constraint is printed in blue.

A viable alternative in the presence of redefinitions as in UML would be to redefine the power supply reference by a new reference with multiplicity 0[70]. In fact, this alternative is close to our implementation in NMeta.

The metamodel using NMeta is depicted in Figure 7.6. The notation of an assignment in the attribute compartment of a class here indicates an `AttributeConstraint` or `ReferenceConstraint` (depending on whether the given feature is an attribute or a reference). In this case, we model that the *powerSupply* reference is constrained to equal the contents of an empty collection. This constraint has the consequence that the generated class for `PassiveSensor` again does not inherit from `Sensor` but implements the power supply reference privately by returning `null`. The setter checks whether the assigned value is `null` and throws an exception that it is not possible to use the passed element as a power supply for a passive sensor. In fact, the error message is exactly the same as in the case of `ACSensor` because the code generator does not distinguish between a structural decomposition and a refinement, instead always assumes both.

7.5.3. Connections of a three phase squirrel cage motor

Besides association redefinition, all modeling alternatives that exist for refining single-valued references as in Section 7.5.1 are also available to model a scenario such as the three phase squirrel cage motor. However, as creating multiple derived attributes or references may be cumbersome, Ecore has dedicated support for decompositions using feature maps. While these

[70] An upper bound of 0 is explicitly allowed in the specification [156, p.34].

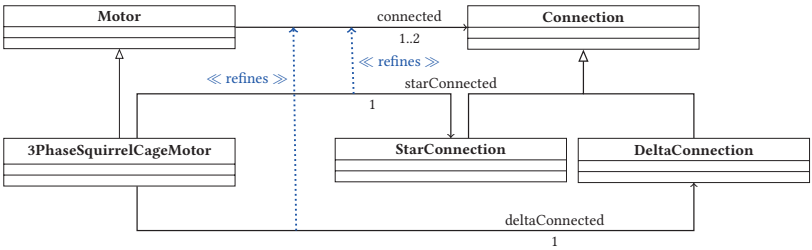


Figure 7.7.: Star and delta connections in motors modeled in NMeta. The inserted structural decomposition is printed in blue.

feature maps ease the specification of otherwise multiple OCL constraints, derived references and can be used for an improved editing support, they do not solve the problem that the constraints have to be checked and are not guaranteed by the type system.

For association redefinition, the situation is less clear because the case that multiple properties redefine a given property in the context of the same class is not described in the specification and ignored e.g. by Costal and others [48]. Here, our notion of structural decomposition may be adopted to specify the semantics clearer.

Moreover, the implementation of keeping the refined reference and implementing refinements as constraints on top of it does not necessarily work with multiple redefinitions. For instance, consider the case we added a new class that inherits both from `StarConnection` and `DeltaConnection` and the motor is connected to an instance of such a class. Then, it is no longer clear whether it should be referenced from the *starConnection* reference or the *deltaConnection* reference.

Using NMeta, the three phase squirrel cage motor can be modeled as in Figure 7.7. Here, we simply declare that both *starConnected* and *deltaConnected* references refine the general *connected* reference.

As a consequence, the generated implementation class of `3PhaseSquirrelCageMotor` does not inherit from the implementation class of `Motor` but implements the *connected* reference privately. This private implementation uses a generated custom collection implementation that internally enumerates the *starConnected* and *deltaConnected* references. Because these references are

the only ones that are backed by a field[71], the type system guarantees that the connections of a three phase squirrel cage motor cannot be anything else than at most one `StarConnection` and at most one `DeltaConnection`. Since these references have an upper bound of 1, we can even guarantee the upper bound for the *connected* reference, at least for a three phase squirrel cage motor.

Because the data is in the more detailed references, the refinement also works if we used the same class `Connection` for both cases or we introduced a derived class that inherits from both `StarConnection` and `DeltaConnection`. In that case, the modeler simply has to use the correct reference, either *starConnected* or *deltaConnected*, to achieve the intended model.

Our implementation allows manipulations of refined collections such as adding and removing elements. In case of adding an element, this is dispatched to add or assign the added element to the first reference that matches the type. In the example, adding a `StarConnection` element to the *connected* reference would implicitly set the *starConnected* reference. An element that does not match any refining references causes an exception to be thrown.

In addition to the type, we also take the multiplicities and the current cardinalities into account. Therefore, if we assume for a moment that the *starConnected* and the *deltaConnected* references were typed with `Connection` and we added a new connection to the decomposed *connected* reference, the system would check whether the cardinality of the *starConnected* reference is already at its upper bound. If so, it would try to assign the added element to the *deltaConnected* reference. If the latter is also set, an exception is thrown because the upper bound constraint would be violated by adding the connection element.

7.6. Summary

In this chapter, we have proposed a formal definition of refinements and structural decomposition, how they can be implemented in a meta-metamodel and how a code generator can be designed to ensure them through type

[71] Unless of course, they are themselves refined in a more specific subtype

system guarantees, in line with RQ IV.1. This can make many validation constraints obsolete. We have shown the applicability of the proposed concepts in a practical scenario and discussed the advantages of our approach over alternative ways to model the same situation.

Our notion of refinements matches redefinition of properties as defined in the UML. However, UML redefinitions do not consider the case that multiple properties redefine the same property and currently, it is unclear what the semantics should be in that case and even whether this should be allowed at all. Here, our approach proposes a semantics by extending the semantics of the refinement information to structural decomposition. As our implementation shows, these semantics can be enforced by underlying type-system guarantees in case the metamodel is generated to code.

Because refinements and structural decomposition can be stacked, these concepts make it viable to model a system in a fine granularity in order to ensure correctness while still being able to analyze the model at a high level of abstraction. We envision that metamodels may contain specific metaclasses down to a low level of abstraction, basically down to a level of manufacturers and makes of a certain type of component. While this allows very detailed correctness checks, it also bloats the metamodel and demands for concepts to specify the make of a component type in a different model. Therefore, we aim to solve these problems through techniques of Deep Modeling in Chapter 8.

8. Simplify model analyses through Deep Modeling

In this chapter, we go one step further than in the last chapter and investigate to what degree the proposed meta-metamodel extensions also support Deep Modeling, i.e. the support for modeling instantiation relationships between model elements.

Many of the most often used modeling formalisms, such as Ecore, do not allow their users to model the relation that a model element is an instance of another, except that all model elements are instances of their class [135]. This leads to accidental complexity as metamodels must be aided by helper constructs. Instantiation relations must be described using references, for example using some kind of connector classes. The semantics of instantiation is then restored by introducing OCL constraints that ensure a valid usage to mimic instantiation relations. While this solves the modeling problem in the first place, it has a range of problem attached. First, it makes the creation of instances more difficult as modelers cannot express their intend directly. Further, it makes automated tool support difficult as such tool support has to analyze the OCL constraints to reconstruct the original intention.

For model analyses, this accidental complexity yields more complex analyses that have worse non-functional properties. In the context of incremental execution, this added complexity is even more a problem because it induces larger DDGs and a higher memory consumption. Moreover, the accidental complexity in the metamodel also implies an accidental complexity for the types of changes that may occur in the model which in turn results in a worse incremental performance.

As pointed out by de Lara and others [135], a domain where instantiation relationships between multiple model elements are rather common are architecture description languages of component-based systems. These languages typically define a concept of components where the usage of a component

in a softwae architecture can be seen as an instantiation of the component, often called assembly [21]. The properties of such an assembly are partially determined by the instantiated component because it needs to be connected to other assemblies that provide interfaces required by the component.

Approaches that aim to directly support instantiation relations between model elements are referred to as Deep Modeling or Multi-Level Modeling concepts[72]. The latter term originates from the idea that such approaches not only support the usual two levels of metamodels and models, but that a model repository may contain non-transitive instantiation chains of arbitrary length. This means that a model element *A* can be an instance of another model element *B* which itself is an instance of model element *C*. However, unlike inheritance, instantiation is not transitive so that *A* is not an instance of *C*.

If such an instantiation relationship between model elements is mimicked in two-level modeling standards, for example using connector model elements that assign a value for a given instance element for a given reference element, then it gets more complex to obtain the actual value for a given reference. In the case of software architectures for component-based systems, it may be necessary to traverse a large part of the architecture model to find out what assembly is connected to a given other assembly for a given required interface.

For an incremental execution of such analyses, this is a problem because many elementary model changes must be considered when an analysis traverses such a reference. In the example of software architectures for component-based systems, the assembly connected to a given assembly for a given required interface may change either if a connector is added or deleted or if any of the connectors change their connections. If, however, the assembly was an instance of the component, then we could raise an event when the connected assembly for a given requird interface changed and pick up this event in an incremental model analysis.

Most existing approaches to describe these instantiation chains use the concept of potencies [15] to describe these instantiations. Therefore, we tried to apply potency concepts to create a Deep Modeling version of the Palladio

[72] The terminology Deep Modeling was decided on at the MULTI 2014 workshop and we adopted it in this thesis.

Component Model (PCM). However, applying potency concepts to the PCM, we hit a wall trying to model composite components as they may contain instances of other components. This means that we cannot assign several classes such as the component class a fixed level – which is problematic for level-adjuvant languages for Deep Modeling in general. Furthermore, developers usually stay with the technologies they are used to as far as they can. Such mentality was shown for general purpose languages by Meyerovich [149] and we suspect that this is also true for modeling languages.

A further potential disadvantage is that all the subsequent tools such as model transformations have to be adjusted for Deep Modeling, as e.g. done by Atkinson [13] with an adjusted version of ATL called DeepATL. Given the plethora of model transformation languages where even the most commonly used ones have much smaller user bases than most general purpose languages, we think that few transformation languages will be adopted and maintained for Deep Modeling.

In particular, we would like to avoid having to adjust the incrementalization system from Chapter 4 or the incremental model transformation approach from Chapter 6 for Deep Modeling. Rather, we want to keep using these tools, but take advantage from the availability of dedicated change notifications.

Therefore, we propose a pragmatic approach how Deep Modeling, i.e. instantiation chains of arbitrary length, can be realized using only two *non-invasive extensions* to meta-metamodels aligned with EMOF, such as Ecore. The crucial advantage of this approach is that all the tools available for such a meta-metamodel can be reused and existing metamodels do not have to be changed. In particular, our approach allows us to use implicitly incremental model analyses and transformations also for Deep Modeling. Thus, Deep Modeling can be introduced in a stepwise evolution process and only where it is beneficial.

To validate the latter statement, we apply our Deep Modeling approach to PCM. Next to classical architecture description, the PCM also contains several metaclasses to describe control flow on an abstract level. Control flow can be described easily without the usage of instantiation relationships such that Deep Modeling concepts are not required. We demonstrate and explain the simplification of model analyses through a model analysis that checks the deployment of connected assemblies.

The remainder of this chapter is structured as follows: Section 8.1 discusses the advantages but also challenges creating a Deep Modeling version of an architecture-description language based on PCM. Section 8.2 introduces our approach of realizing Deep Modeling through structural decomposition. Section 8.3 applies our approach to create DeepPCM, a Deep Modeling version of PCM, and compares DeepPCM with the original PCM to reason on advantages and disadvantages taken. Section 8.4 discusses the interplay of the proposed Deep Modeling approach with incremental model analyses. Finally, Section 8.5 summarizes insights and achievements of this chapter.

8.1. Challenges and Potentials Applying Deep Modeling for Architecture Description Languages

In this section, we discuss the challenges and potentials when applying Deep Modelling to model component-based system architectures. The discussion is based on PCM, which we regard as an established architecture description language and analyze it for the spots where Deep Modeling could be advantageous. Section 8.1.1 discusses the advantages that can be drawn from Deep Modeling when implemented using the Potency concept by Atkinson and others. Section 8.1.2 discusses how the concept of composite components fits into this application and shows how the inclusion of this concept breaks the fixed level architecture required by Potency-based Deep Modeling approaches.

8.1.1. Architecture description languages using Potency concepts

This section discusses the potential advantages that we see applying Deep Modeling to reengineer architecture description languages such as PCM. In particular, we discuss the usage of potency concepts as introduced by Atkinson [14]. Throughout the rest of the chapter, we will refer to the approach created as *DeepADL*.

To answer where to apply Deep Modeling, one has to look for concepts that are best described with an instance-of relationship, for example using the

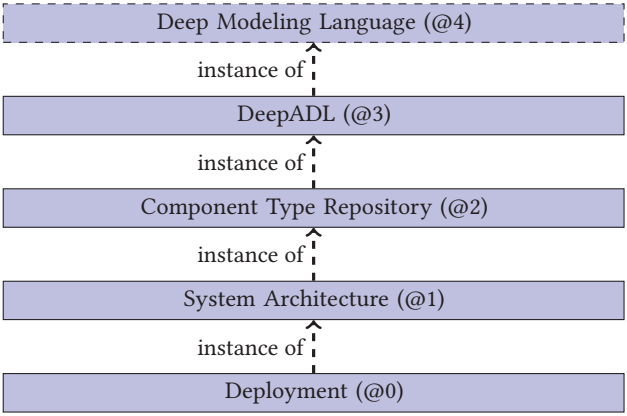


Figure 8.1.: Levels of DeepADL

patterns of de Lara et al. [135]. In PCM, we have identified several of these instantiation relationships.

At first, there is of course the definition of DeepADL itself. In traditional two-level metamodeling, one would usually have the metamodel as an instance of the meta-metamodel and there are at least opinions that such a self-description would be desirable for Deep Modeling approaches as well. However, in many level-adjuvant languages that we see nowadays, the metamodel is an instance of a Pan-Level-Metamodel (PLM) which is not equivalent to the instance-of relation for the other levels. This is why we have drawn this language box only dashed. All of these levels are labeled with a number, as usual in level-adjuvant languages.

The first step in developing a component-based system is of course the definition of components which are instances of the general component concept. Thus, we have an instantiation relation between DeepADL itself and the repository level.

Assembly contexts have been introduced by Becker and others as instantiated components [21] and therefore, the next instantiation level would be from the repository level that is instantiated by the system architecture level in which developers use instantiated components to assembly a system architecture.

Even below the system architecture, there is a further level of the systems deployment. In this deployment, the assembly contexts of a given system are assigned the resource containers to which they shall be deployed. This information, that instantiated components will eventually get deployed, is already clear at the level of assembly contexts and could be specified through Deep Modeling constructs such as potencies.

These levels reflect the development process of a component-based system. At first, a component model is created (or selected from the many already existing ones). Then, component types are created before they can be assembled to a component-based system which in the end can be deployed.

Because PCM is modeled in the traditional two-level structure, this multi-level development process is not reflected in the metamodel. Whereas the concept of components spans all levels depicted in Figure 8.1 (except for the modeling language), every occurrence of components in the model is modeled with a separate class with a reference to the class a level above. Further, additional classes such as connectors implement property slots that lose type information such as the information on a provided interface. This type system structure has to be preserved with OCL constraints.

Here, Deep Modeling languages and in particular level-adjuvant languages can help to reduce accidental complexity because they allow the metamodeler to express properties of components from all levels in a single class definition through the specification of a potency. Such a potency defines the meta-level on which a class may be instantiated. For example, one may give the class `Component` a potency 3 to specify that this class can be instantiated three levels deep. Likewise, references are assigned a potency to declare until what level the reference must be assigned a value.

A sketch of how a component-based software architecture description language similar to PCM based on level-adjuvant languages with potencies is depicted in Figure 8.2. In this figure, we have modeled the required interface of the encoding adapter from Figure 2.3 as well as the provided interfaces from the database adapter and the encoding component.

In particular, level @3 declares the abstract definition of the modeling language: It simply consists of the concepts of components, interfaces and resource containers.

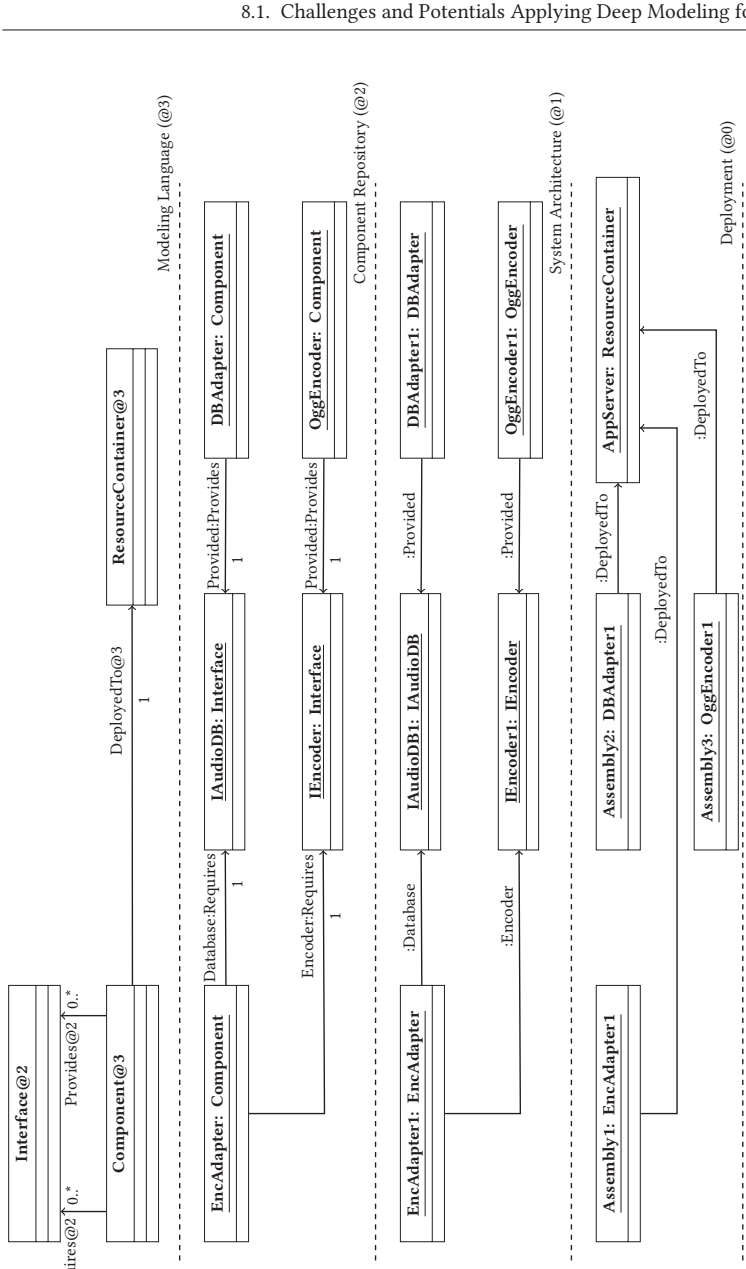


Figure 8.2.: Modeling architectures in level-adjuvant languages

In the next level @2, we define the component types such as the `EncAdapter`[73], `DBAdapter`, `OggEncoder` and the relevant interfaces. These component definitions are instances of the classes `Component` and `Interface` from the @3 level. In particular, the model allows us to model that a component requires an interface multiple times without rather artificial helper model elements such as roles.

Because `Component` and `Interface` have a potency greater than one, we can instantiate their instances again. The instances of the component types are the assemblies in level @1 that represents the system architecture. We instantiate the references defined in @2 that make up the required interfaces.

Lastly, because `Component` has a potency 3, we are allowed to instantiate the assemblies again in level @0 that represents the system deployment. In Figure 8.2, we simply deploy all assemblies to the same resource container called *AppServer*.

Figure 8.2 shows very nicely how Deep Modeling helps to avoid accidental complexity: The component language in Figure 8.2 essentially is about as expressive as the metamodels in Figures 2.4 and 2.5 taken together – with just three metaclasses.

From the perspective of an end-user that wants to model a given system, the Deep Modeling version is also potentially easier since much of the semantics how component instances for example can be connected to a system is already contained in the instantiation semantics and can thus be much easier supported by tools. For example, a recurring problem creating system architectures in PCM is that people forget some required interfaces when assembling a system architecture. There is a constraint in the metamodel to mark such a model as invalid, i.e. for each required interface of an assembly contexts component type, the assembly context must have an assembly connector that connects to an assembly realizing this interface. The constraint is formulated at the level of a system because the assembly connectors are not contained by the assembly contexts but rather by the enclosing system. However, this means that without analyzing the structure of the constraint, by default the system will give an error message to the modeler essentially

[73] In Figure 2.3 and in [21], this component is called `EncodingAdapter` but we abbreviated the name to make the figure more readable.

saying that the constraint is violated, not necessarily giving the modeler an insight why.

This is different for the Deep Modeling version because here, the same constraint is expressed in a multiplicity such as the specification that an *EncodingAdapter* needs exactly one connection to the *IAudioDB* interface. Thus, when a modeler creates an architecture where he forgets to specify an *IAudioDB* implementation for a given instance of *EncodingAdapter*, the system will be able to conclude that the cardinality of the `Database` reference of the assembly context is wrong (is 0 meanwhile there is a lower bound 1), giving the modeler a much better insight where the problem is.

For these reasons, a Deep Modeling version of architecture description languages such as PCM would be highly appreciated.

8.1.2. Composite Components

Up to the version of Figure 8.2, DeepADL does not yet support composite components. To decide on how to best integrate this concept into the Figure 8.2, we need to think about on which level composite components are instantiated.

Unlike basic components that are the smallest unit of implementation, a composite component bundles the functionality of multiple other components (possibly composite components as well) but does not contain any implementation on its own. In fact, composite components act like a facade to their inner component instances. They are very similar to software architectures in that they contain some assembly contexts, i.e. instances of components, that are assembled together (which is why composite components and systems share common base classes in PCM).

Nevertheless, composite components are components and thus belong to the repository level: the assembly of the inner components of a composite component is independent of the choice in which systems a composite component is used if used at all.

For a level-adjuvant language, this raises a problem because using the sketch from Figure 8.2, composite components require a reference to assembly contexts that are on a lower modeling.

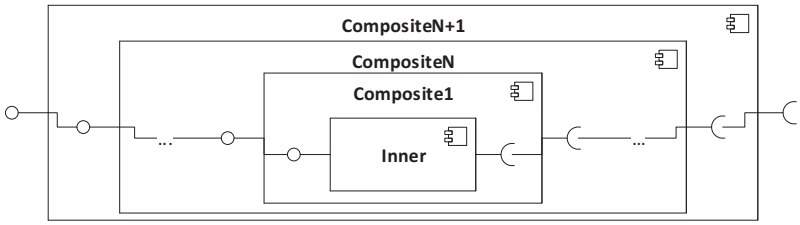


Figure 8.3.: Nested composite components that imply an arbitrarily deep level structure

This level-crossing makes it very hard to model composite components with level-adjuvant languages: Composite components may contain instances of composite components but since an instance of a model element must always be in a lower modeling level than its type, this immediately means that the modeling level on which composite components may be instantiated cannot be specified in advance. This also holds for the maximum number of modeling levels.

In particular, it is easy to construct composite component types that span over a given amount of modeling levels. We simply need an arbitrary component and wrap it in composite components. A second composite component then wraps the first one until we have a cascade of composite components, depicted in Figure 8.3.

Furthermore, the innermost component can still be instantiated at the level of any composite component and might as well be instantiated together with the outermost composite component. Thus, components may get instantiated at all levels, entirely contradicting the foundations of level-adjuvant languages.

To solve this dilemma, we see two possible solutions. The first one would be to discard the shared functionality of composite components and systems and implement a traditional two-level version only for assembling composite components. However, this implies many duplicated concepts which in turn cause maintenance efforts. These efforts may even be higher than the current two-level versions since both the two-level version for composite components as well as the Deep Modeling version for systems have to be maintained. Furthermore, we are sure that this approach would be confusing modelers, because they would have to model a component differently, depending on whether it is used in a composite component or directly. The other solution

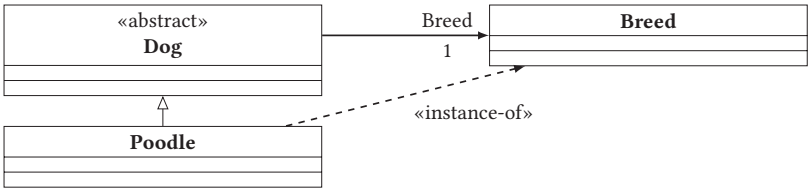


Figure 8.4.: Poodle is both a class and an instance

is thus to discard Deep Modeling concepts at all and implement the same two-level approach but this also means that all the benefits we have shown in Section 8.3.1 would be lost.

The underlying problem here is that we have a composite pattern where the composition crosses modeling levels which is why we refer to this situation as composite instantiation. It consists of a model element that can contain instances of other instances of its own type such as a composite component may contain instances of other composite components. As noted above, this pattern implies that the amount of levels can no longer be fixed.

Meanwhile a composite component may contain instances of composite components, it must never (not even indirectly) contain an instance of itself as this would lead to a an endless loop when determining the behavior that should actually be executed, a paradox often attached to level-blind[74] Deep Modeling approaches [12].

8.2. Deep Modeling through Structural Decomposition

In this section, we describe how non-transitive instantiation relationships can be modeled using structural decompositions and refinements. Transitive instantiations mean that there can be model elements *A*, *B* and *C* such that *A* is an instance of *B* and *B* is an instance of *C* but *A* is not an instance of *C*.

[74] The terminology of *level-blind* Deep Modeling approaches has been introduced by Atkinson and others [12]. We use it here because we are not aware of another terminology.

A prominent example depicted in Fig. 8.4 is that a concrete dog (a poodle) can be an instance of `Poodle` which in turn is an instance of `Breed`. In this example, `Poodle` acts both as model element (object) and as a class which is why it is often called a *clabject* to express this duality [10].

Such a situation can be described using the powertype pattern first presented by Odell [158]. However, many modeling environments such as Ecore currently do not support this pattern and even in the UML, it is an isolated concept. This means, the UML specification has dedicated support for this pattern, but powertypes are used nowhere in UML.

On the other hand, there is a very prominent example of a clabject even in traditional two-level modeling with self-descriptive meta-metamodels, namely the class `Class` which is an instance of itself. Its properties as a class are described by the references and attributes of `Class` because the meta-metamodel effectively describes a type system. Essentially, the class `Class` describes that its instances can be instantiated, typically done through a mapping to a platform class that we call code generation.

The core idea of this chapter is to reuse this duality of the `Class` element on a broader scope. Thus, whenever we conceptually face a clabject, a metaclass whose instances can be instantiated again, this metaclass should be a subtype of `Class`.

In that sense, our approach is slightly related to UML stereotypes where developers may attach a domain-specific stereotype to a class to express that it is in fact a domain element. An often perceived problem with this usage of stereotypes is that besides the stereotype, the class is still a class and the properties as a class may be non-existent in a given domain.

However, unlike UML stereotypes, we hide the type system relevant information by decomposing them into domain references. For example, the base type references of a `Breed` can be decomposed into the reference to the group a breed belongs to or refined by an empty reference which essentially means to cut off the inheritance concept. Features that are not applicable or are constant for all instances (like class-level variables) are cut off using a refinement with a constant. In the poodle example, this includes attribute or reference constraints.

But as the domain concepts are still classes as they (possibly indirectly) inherit from the class `Class`, we can use the standard generators to generate model representation code for them. With this generated code, the class

nature of a clabject is represented by a mapping into the platform type system while the object nature of it is represented as a model element.

This mapping of the type facet to the platform type system also yields the consequence that an instance of a clabject *A* cannot be in the same model as *A* unless the model is manually bootstrapped such as done for `Class`. As a consequence, a clabject cannot easily be an instance of itself unless the model developer has explicitly expressed an intent that this behavior is desirable by bootstrapping the model. This neglects the various paradoxa presented by Atkinson et al. [12] for languages he referred to as level-blind.

On the other hand, the instantiation implies a stratification of the model and divides it into levels as suggested by Atkinson [12]. However, these levels can be crossed not only by instantiation relations. Our case study example in Section 8.3.1 will give an example where this enables us to keep a level structure in the presence of model elements that would otherwise break the level structure. Basically, this is possible by binding user defined clabjects to classes known in advance through the instance-of relation. These base classes are known before any model for the deep metamodel is created and thus can freely be referenced, including a usage for analysis or transformation purposes.

Besides refinements and structural decomposition, we found that it is also useful to be able to restrict inheritance to instances of a given class. In NMeta, this is implemented through the `InstanceOf` reference already depicted in Figure 7.3 on page 185. This reference may only be specified for abstract classes. If a class *A* is an instance of class *B*, then only instances of *B* may inherit from *A*. Consider *a* an instance of type *C* which inherits from *A*. Because *B* was declared as an instance of *A*, *C* must be of type *B*. Thus, the type of *a* is an object of type *B*. Since this is known at compile-time, the generated code contains a refined method to obtain the model elements type of type *B*.

With this relation, we may specify that `Dog` is an instance of `Breed`, meaning that the type of each dog element will be in turn an instance of `Breed`. Thus, the `Breed` reference is just a converted type reference and hence, the

`InstanceOf` relation can be seen as a formalization of the powertype pattern, expressed with a single reference[75].

The relationship to stereotypes is hardly surprising since Henderson and Gonzalez-Perez have already shown a close connection of the powertype pattern to UML stereotypes some years ago [89]. We will see the `InstanceOf` reference being used later in Section 8.3.1 in a number of places.

If the `InstanceOf` reference is left blank, this has the same effect as specifying that a class is an instance of `Class`. This is because the base class `ModelElement` is marked as an instance of `Class`. Moreover, when classes define an instance-of relation and one of its base class also specifies an instance-of relation, then the new instance-of class must be a subtype of the base class instance-of class. As an immediate consequence, all classes used in the instance-of reference must be subtypes of `Class`.

8.3. A Deep Modeling version of PCM

In this section, we apply the approach presented in Section 8.2 to the architecture description language PCM and obtain a new architecture description language DeepPCM. We explain excerpts of its metamodel in Section 8.3.1 before Section 8.3.2 explains how it is used in an example and compares this to the original PCM.

8.3.1. An architecture description language using Deep Modeling

To validate our concept for Deep Modeling, we created DeepPCM as a language for architectural description using our Deep Modeling approach. DeepPCM reuses major parts of PCM as is: The performance-relevant elements of PCM such as service effect specifications or stochastic expressions are untouched. Many of these concepts do not benefit from Deep Modeling and are therefore not relevant for this thesis. However, our approach allows us to simply copy them from a strict two-level metamodel of PCM into a

[75] In the implementation, an instance-of relation does not imply a refinement of a classes base types and therefore two model elements are required in this case.

Deep Modeling metamodel of DeepPCM, making this approach attractive for refactorings for Deep Modeling.

In this thesis, we only describe what we think is the core of an architecture description language – the assembly of a component-based system from several component instances. We show how this is modeled using the prototype of DeepPCM. The metamodel parts for other aspects such as modeling resource demands using stochastical expressions has not been the focus of our work. These aspects are modeled exactly the same way as the metamodel parts have just been copied.

The DeepPCM prototype has been extended to cover the full functionality of PCM in a bachelor thesis by Alexis Bernhard [27], supervised by the author of this thesis.

We applied the level hierarchy from Figure 8.1 and tried to keep as close as possible to the metamodel sketched in Figure 8.2, but show how our approach is capable to model also composite components.

8.3.1.1. Modeling System Architectures

We first explain the metamodel excerpt for component types. Component types are represented by the abstract class `Component`. Since component types can be instantiated, they are clabjects and therefore `Component` must inherit from `Class`. Same as PCM, DeepPCM supports two different kinds of component types, basic components and composite components. Instances of a component type are assembly contexts, so we create a reference constraint to the base types of `Class` and fix it to the `AssemblyContext` class. Conversely, we specify that `AssemblyContext` is an instance of `Component`.

Further, a component type *is* an implementation of its provided interfaces. The most convenient way to model this is that the component types inherit from the interfaces that they provide. Thus, `Interface` itself must be clabject as well and the provided interfaces of a component type refine its base types. Here, we introduced a subtle difference to PCM since components in PCM are allowed to provide the same interface multiple times through multiple roles. However, while the metamodel allows this, much of the tool support currently assumes that an interface can only provided once per component type so that this restriction is actually more accurate.

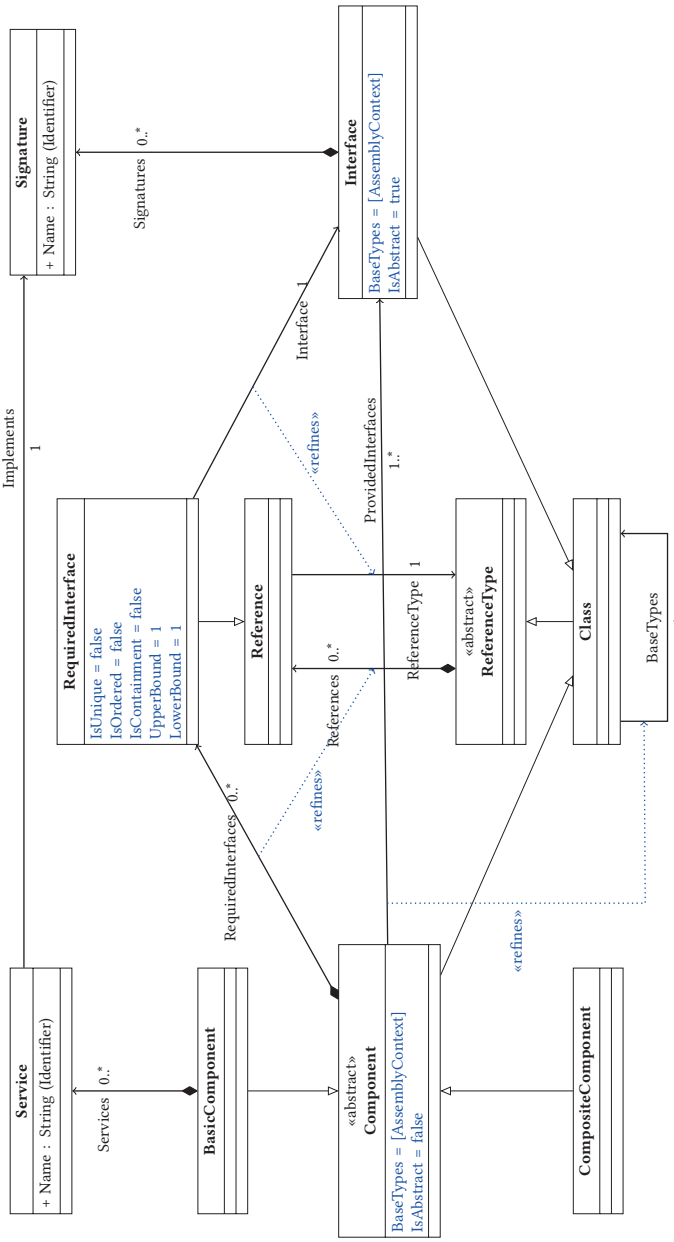


Figure 8.5.: Component types in DeepPCM

For each required interface of its component type, an assembly context must be assigned an assembly context whose component type provides the required interface. In PCM, this is modeled through an `AssemblyConnector`. In DeepPCM, we wanted assembly contexts to have a strongly typed reference for each required interface of the component type so that the ontological property of required interfaces of a component type becomes the linguistic element that the assembly context should have a reference. Hence, the required interfaces refine the references of `Component`. Thus, `RequiredInterface` must inherit from `Reference`. The `ReferenceType` reference of `Reference` must be refined since for a `RequiredInterface` element to be valid, the type must be an `Interface` element.

Because component types are classes, we can map them to a platform class and instantiate instances of component types. These instances are assembly contexts since any component type as a class also inherits from `AssemblyContext`. This makes the component type of an assembly a part of its identity since the type of an object cannot be changed during its lifecycle. Thus, unlike e.g. in PCM, the component type of an assembly cannot change and assembly contexts cannot exist without a component type. Here, important validation rules are ensured directly by the type system which we see as a big advantage.

The signatures offered by an interface are not relevant at the system architecture level since the fact that a component type provides a particular interface already implies that a component provides services that implement the interfaces signatures. As a consequence, the ontological properties of an interfaces signatures or a component types offered services are no linguistic properties and are therefore usual references, i.e. no refinement references. Furthermore, also the disjunction of components into basic components and composite components has no effect on the system architecture.

The entire metamodel up to this point can be seen in Figure 8.5 where we again used blue, dotted lines to represent reference refinements. The classes `Class`, `Reference` and `ReferenceType` are imported from NMeta and we have omitted their details for brevity. Furthermore, the attributes compartment of the classes also contain reference constraints and attribute constraints. Note in particular the attribute constraint of the `Interface` class that sets the `IsAbstract` attribute to True, meaning that instances of `Interface` as classes are abstract.

```
1 public interface IAssemblyContext : IModelElement, IReference {
2   IComponent GetComponent();
3   IAssemblyContext GetRequiredInterfacesValue(IRequiredInterface reference);
4   ...
5 }
```

Listing 8.1: The generated interface for an `AssemblyContext`

Because `AssemblyContext` is marked as an instance of `Component`, we can generate a method that returns the component type of an assembly context. Furthermore, as the references of a component type are decomposed into its required interfaces, we statically know that each component instance will have an assembly context assigned for each required interface. Therefore, we can generate a method that takes a `RequiredInterface` and returns the corresponding `AssemblyContext` element connected on the reference generated for the required interface. The name of this method is constructed using the name of the reference that is used as a component for the references of a `Component` as a class.

The interface for an `AssemblyContext` is depicted in Listing 8.1. Any class generated for an instance of a component type implements this interface. To implement the `GetComponent` method, the generated code for a particular component is aware of the component it was generated for and resolves the URI of that component. The implementation of `GetRequiredInterfaces-Value` simply gets the value for the given reference and casts the result to the statically known type. In this case, we know that the reference type of a required interface is an instance of `Interface` and therefore must have the base type `AssemblyContext` and must be single-valued.

We believe that these generated methods make the generated API even simpler to use than with the workaround in traditional two-level modeling where developers of a model transformation would have to iterate manually through connector elements such as `AssemblyConnectors` in PCM to find the assembly context connected for a given required interface.

Assembly contexts are composed in a system architecture where a modeler instantiates components from a repository and connect them appropriately. Such a system architecture can be deployed to multiple resource environments. This deployment can be seen as an instantiation where the deployed

system is an instance of the abstract system architecture. Thus, we model `SystemArchitecture` as a clabject, i.e. it inherits from `Class`. For each assembly context in a system architecture, developers must assign a resource container where the assembly context will be deployed to. Thus, the assembly contexts of a software architecture form its references as a class. Each such reference is typed with the resource container where the assembly context shall be deployed to.

This deployment view can be seen in Figure 8.6. In this diagram, the separation of the different modeling levels can be easily seen since the classes related to the deployment on the left hand of Figure 8.6 have no connection (except `InstanceOf`) to the classes representing the system architecture which are on the right hand.

A key observation here is that DeepPCM contains classes spanning all levels involved in architecture description, i.e. repository, assembly and deployment. It is not restricted to the highest (repository) level. The purpose of classes on lower levels such as `AssemblyContext` for the system architecture level is mainly to give them an application-independent structure that is usable also for consumers of the model such as transformations or analyses.

8.3.1.2. Composite Components

Next, we describe the representation of composite components. In Section 8.1.2, we discussed that composite components break a fixed level structure because a composite component may consist of instances of components.

NMeta is agnostic of modeling levels and allows this. In particular, the metamodel fragment supporting composite components is depicted in Figure 8.7. Composite components may contain arbitrarily many assembly contexts. These assembly contexts form the assemblies that the component uses to realize its functionality. A second reference `ExposedAssemblies` denotes the subset of assemblies that are exposed to outside world, i.e. the components that realize the interfaces that the component offers.

On the other hand, the component types of the assembly contexts used in the composite component may require interfaces. In order to have a valid model, all assembly contexts within the composite component must be connected to an instance of some class implementing the interface. This may be another

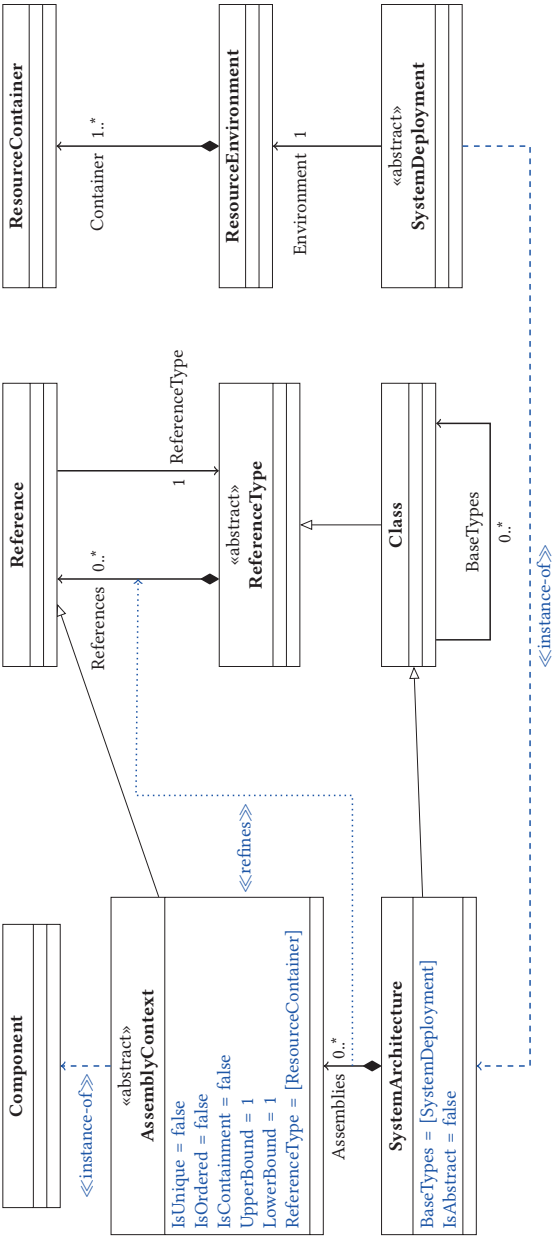


Figure 8.6.: System deployment in DeepPCM

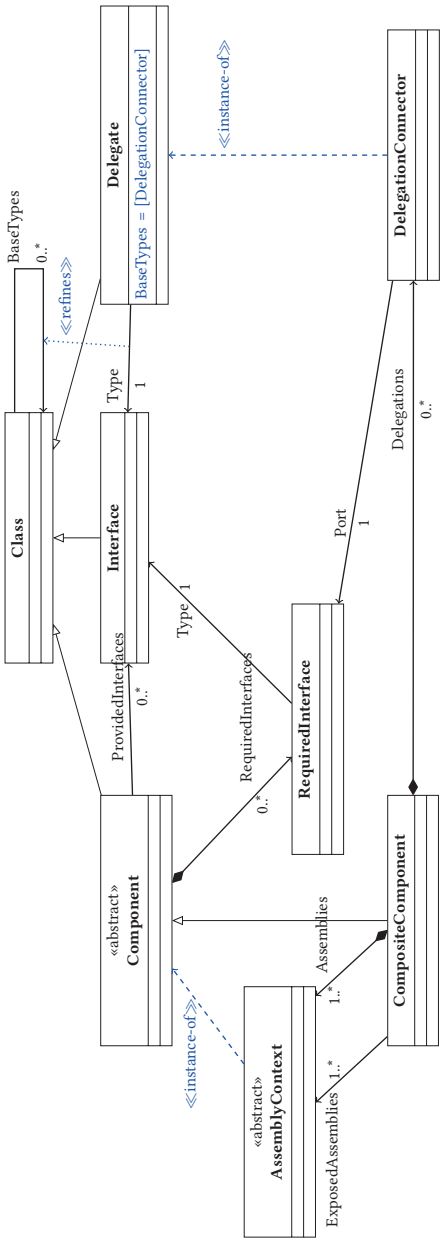


Figure 8.7.: Composite components in DeepPCM

assembly context within the composite component but it may also be a delegation to the required interface of the composite component. While the first does not require any additional model elements, the latter requires to model delegation connectors explicitly. These delegation connectors must be instances of the interfaces which they delegate to. We model this as being instances of a common `Delegate` class. These delegates act like delegate type definitions in .NET that basically simply define a method signature. Likewise, delegators in DeepPCM simply reference the interface which they delegate. Since they can be instantiated in delegation connectors, they are clabjects as well and thus inherit from `Class`.

This requires a new validity constraint as a delegation connector may only use a port with a type that is referenced from its delegate type.

The downside of our approach unveils when compared to a potency-based approach such as sketched in Figure 8.2. This sketch required as little as just three metaclasses to model the components in a component-based system, their assembly in a software architecture and their deployment. For each of these metaclasses, DeepADL requires roughly one metaclass per level on which the metaclass can be instantiated. For the class `Component` in the potency-based sketch, DeepADL requires the metaclasses `Component` and `AssemblyContext`. In this case, a third class can be saved because the instances of an assembly appear as references in the deployment.

8.3.2. Example Usage: A Media Store in DeepPCM

To evaluate the advantages of DeepPCM over architectural languages using traditional two-level modeling, we consider a concrete example. In particular, we modeled a simple e-commerce application called the *MediaStore*. This e-commerce application lets users upload and download media files that are persisted in a database. Uploaded media files are processed with a watermark and saved to the database. This system has been used as a case study for PCM already in 2009 [21]. The implementation of DeepPCM along with models and the generated code for all levels can be obtained online[76].

[76] Prototype: `http://github.com/georghinkel/DeepModelingDemo`, Full implementation by Alexis Bernhard: `https://github.com/ghmanager/DeepPCM/`, retrieved 2 Aug 2017

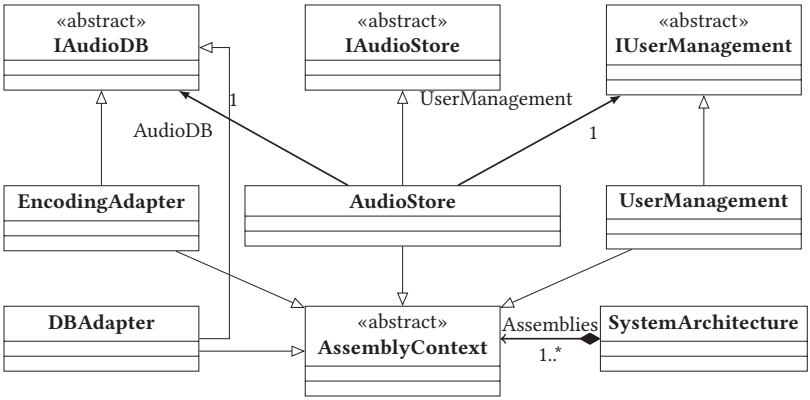


Figure 8.8.: Excerpt of the implied DeepPCM metamodel to specify the architecture of a MediaStore system based on a repository of component types

An overview of the MediaStore is depicted in Figure 2.3 on page 41. This figure contains all three levels that have been discussed above. The components and the interfaces of the MediaStore form the repository. These repository components are then used to create the system architecture by composing them together. Finally, Figure 2.3 also shows the deployment that all components are deployed to a single application server except for the database component and the web browser.

In the component repository, the differences between PCM and DeepPCM are small. This is reasonable since components are instances of the DeepPCM metamodel just as they would be using two-level metamodeling.

If we take a look to the next level, the situation is different. In PCM, assembly connectors are very generic (cf. Figure 2.4 on page 42). The fact that in a valid system architecture, each component must have an assembly connector for each required interface must be specified through an OCL constraint. Consequently, if a user forgets to add an appropriate assembly connector, he gets an error message saying that not all interfaces are connected unless a more appropriate error message is deduced from the OCL constraint or manually implemented. Conversely, one also needs to ensure that all assembly connectors of an assembly context are valid.

In the DeepPCM version the assemblies are much more specific to the component repository of the MediaStore. Since they are in fact classes, we can visualize them in a class diagram. An excerpt of this class diagram around the `AudioStore` component is depicted in Figure 8.8. Here, each required interface is turned into a reference because in fact a `RequiredInterface` model element *is* a reference. Each of these references has a multiplicity of 1, therefore implying a constraint that for example an `AudioStore` must have an `AudioDB` assigned. This constraint is enforced by the type system disallowing the modeler to connect assemblies if the interface types do not match. The error message that can be presented to the user if this contraint is violated, that the `AudioDB` of an `AudioStore` cannot be null, is much more specific and likely to be more helpful.

At the same time, the number of model elements required is drastically reduced. While in PCM, each component instance requires one model element for the assembly context plus one for each required interface of the instantiated component type, DeepPCM has only one model element per component instance where all relevant information for this assembly is combined and correctly typed.

The NMF code generator for classes generates us a model representation code for the `AudioStore` component type, i.e. a generated API. We use this API to (currently programmatically) create instances of this component type. This generated API only allows us to set domain-specific properties like the referenced `IAudioDB` component but it does not show us class characteristics like references or attribute constraints. The reference of references has been refined whereas the attribute constraints reference has been constrained. In fact, the `AudioStore` class does not inherit from `Class` and explicitly implements its interface.

A similar statement is true for the deployment. Here again, the solution in two-level modeling such as applied in PCM is to introduce a generic concept of an allocation context. Whereas the deployment in PCM consists of an allocation connector per assembly, DeepPCM concentrates all required information in a single model element. However, in this case the type system does not bring an advantage since the type of a resource allocation is independent of the assembly. An excerpt of the PCM metamodel regarding the deployment can be found in Figure 2.5.

On the other hand, the deployment for the MediaStore in DeepPCM is depicted in Figure 8.9. Because we created a SystemArchitecture called

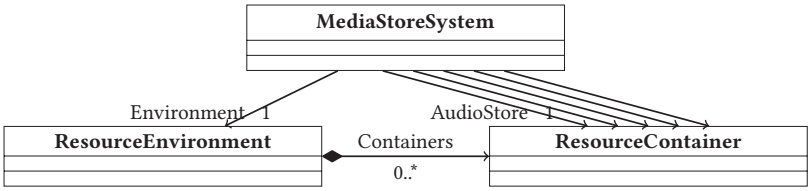


Figure 8.9.: Deployment of the MediaStore in DeepPCM

`MediaStoreSystem`, we can instantiate this clabject for the deployment of the MediaStore. Because an assembly context is a reference, we get a reference for every assembly context that is used inside the system.

Of course, all of this tool support can also be provided with two-level modeling techniques. However, to the best of our knowledge, there is no satisfying solution yet that analyzes the OCL constraints and uses this analysis to provide tool support up to such a level and instead, the tool support has to be created manually. The problem is that this information is widespread among multiple OCL constraints. In PCM, the metamodel excerpt of Figure 2.5 needs one validity constraint, the excerpt from Figure 2.4 already four. In DeepPCM, these validity constraints are already enforced by the type system such that no additional constraint is required.

8.4. Incremental Model Analyses for Deep Models

By construction, the Deep Modeling formalism introduced in this chapter is designed as an extension to existing modeling technology and therefore, the theory of implicit incrementalization systems extends to deep models – models with a chain of multiple instance-of relationships between different model elements – without any further action required. However, the style in which model analyses would be written for deep models often differs entirely from the way analyses are written for traditional models because the concrete type of the model elements are not known – usually even do not exist – at the time the model analysis is specified.

Consider again the domain of component-based software architectures; a typical model analysis that one would like to specify is the validation constraint that assemblies connected to each other should be deployed on the same

```
public IEnumerable<Tuple<IAssemblyContext, IRequiredInterface>>
     GetFaultyContainers
     (ISystemAllocation allocation) {
  return from ass in allocation.GetSystemArchitecture().AssemblyContexts
         let container = allocation.GetAssemblyContextsValue(ass)
         from req in ass.GetComponentType().RequiredInterfaces
         let connected = ass.GetRequiredInterfacesValue(req)
         where allocation.GetAssemblyContextsValue(connected) != container &&
          !allocation.Environment.Links.Any(link =>
            link.ConnectedContainers.Contains(container) &&
            link.ConnectedContainers.Contains(
              allocation.GetAssemblyContextsValue(connected)))
         select Tuple.Create(ass, req);
}
```

Listing 8.2: A model analysis whether assemblies are deployed to the same or connected resource containers in DeepPCM

resource container or otherwise there is a link between these resource containers specified. In a Deep Modeling scenario such as described in Section 8.3, the concrete types of the involved model elements are the component types and system specifications. The latter are specific for a concrete system described in a model and therefore unknown at the time the model analysis is specified.

As a consequence, such an analysis is specified relying on the reflection API of the generated code. For example, an implementation for the analysis whether connected assemblies are deployed to the same or connected components using DeepPCM is depicted in Listing 8.2. To navigate through the involved levels, the implementation uses reflection APIs such as `GetSystemArchitecture`, a type-safe version of `GetClass` that return the type of the current model element (which in case of a system allocation is a system architecture). Furthermore, the analysis requires access to referenced elements for a given reference such as `GetRequiredInterfacesValue`, a type-safe version of `GetReferencedElement` that returns the referenced element for a given reference.

From an incrementalization point of view, it is noteworthy that there is no action to be done to support the correct incrementalization of `GetClass` and its type-safe derivatives – they return the type of a model element that

cannot be changed during the lifetime of a model element[77]. This is different for `GetReferencedElement` because the model element referenced by a given reference may indeed change during the lifetime of the model element.

Therefore, the code generator generates a manual incrementalization of these methods along with their implementation. This generated incrementalization takes advantage of the change events generated for a given reference and therefore only issues a change notification when the value of the respective reference is changed.

The fact that the reflection methods simply use the generated events to notify clients when their return value changes also makes such calls very effective in an incremental setting.

On the contrary, consider the same analysis written for a strict two-level metamodel such as the original PCM. To yield a good comparison, we implemented the analysis from Listing 8.2 for a simplified version of PCM that in particular use the same reference names where applicable[78]. This implementation is depicted in Listing 8.3.

The first insight from a comparison of the analyses in Listings 8.2 and 8.3 is that they almost exactly have the same size in terms of number of lines and even number of characters. The reason for that is that the Deep Modeling version of the analysis has to navigate to the required interfaces of a component while the two-level analysis can make a shortcut and stay on the system architecture and deployment levels: The Deep Modeling version iterates the assemblies of a system and uses the repository level to find connected assemblies meanwhile the two-level analysis directly operates on the connections between assemblies.

Both analyses share that they require to find the resource container to which a given assembly has been deployed to for a given deployment model of a system architecture. While the Deep Modeling version of the analysis uses the generated level-crossing reflection API for that (the call `GetAssemblyContextsValue`), the two-level analysis picks the first allocation context that

[77] At least not in NMF. Other modeling environments more inspired by ontologies where elements do not necessarily have a type at all may support changing the type of an object at any time.

[78] All references in PCM are suffixed with the class name that defines these references which unnecessarily degrades the understandability of the analysis.

```
1  public IEnumerable<IAssemblyConnector> GetFaultyContainers(IAllocation
       allocation) {
2    return from connector in allocation.AllocatedSystem.Connectors
3           let providingAllocation = allocation.AllocationContexts
4                  .FirstOrDefault(c => c.Assembly == connector.
                       ProvidingComponent)
5           let requiringAllocation = allocation.AllocationContexts
6                  .FirstOrDefault(c => c.Assembly == connector.UsingComponent)
7           where providingAllocation.Container != requiringAllocation.Container
                   &&
8             !allocation.Environment.Links.Any(link =>
9               link.ConnectedContainers.Contains(providingAllocation.Container)
                      &&
10              link.ConnectedContainers.Contains(requiringAllocation.Container)
                      )
11          select connector;
12 }
```

Listing 8.3: An analysis whether components are correctly allocated

references the required assembly. In batch execution, the difference between these two ways to query the resource container is almost negligible since the generated reflection API internally simply checks the name of the provided assembly (which exists because all references have a name) and returns the first that fits, very similar to the `FirstOrDefault` function.

However, in an incremental execution, the difference is much more significant. When executed incrementally, the function `GetAssemblyContextsValue` does not only return the value for the given reference (typed as an assembly) but also picks up the change event generated for that reference. This event fires when the value for the given reference changes, which precisely describes when the value of `GetAssemblyContextsValue` called with that reference as argument changes. This is in contrast to the incrementalization of `FirstOrDefault` which has to create a DDG for each element of the underlying collection to incrementally evaluate the given predicate: Because the allocation context elements are not tightly bound to an assembly, same as connector elements, a model manipulation that changes the assembly of a given allocation context or the ends of a connector potentially changes the deployment information for *any* assembly. This has the consequence that a DDG has to be created for every pair of assembly connector and allocation context to evaluate the predicates in lines 4 and 6 of Listing 8.3. Thus, the

```
public IEnumerable<Tuple<IAssemblyContext, IRequiredInterface>>
     GetFaultyContainers2
     (ISystemAllocation allocation) {
  return from ass in allocation.GetSystemArchitecture().AssemblyContexts
         let container = allocation.GetAssemblyContextsValue(ass)
         from req in ass.ReferencedElements.OfType<IAssemblyContext>()
         where allocation.GetAssemblyContextsValue(req) != container &&
          !allocation.Environment.Links.Any(link =>
            link.Connects.Contains(container) &&
            link.Connects.Contains(allocation.GetAssemblyContextsValue(req)))
         select Tuple.Create(ass, req);
}
```

Listing 8.4: Alternative analysis whether assemblies are deployed to the same or connected resource containers in DeepPCM

incrementalized analysis has to listen to n DDGs generated for distinct allocation contexts where n is the number of allocation contexts, approximately the number of assemblies in a software architecture.

One could argue that these DDGs only cause a memory overhead but not affect the response times to changes as long as the assembly of an allocation context or the ends of a connector are not changed. This is possible by preferring to add new connectors instead of modifying existing ones. However, still, adding a connector requires to instantiate n new DDGs which eliminates the chance for an efficient response.

Notably, the level shortcut that we took in the two-level version of the analysis is also available in DeepPCM and is depicted in Listing 8.4. Instead of iterating over the required interfaces that make up the references of a given assembly context, one can also directly iterate over the referenced elements and filter them according to the type. In comparison to the version in Listing 8.2, this saves a `let` operator. This reduces the overhead of the Deep Modeling version of the analysis further.

8.5. Summary

In this chapter, we have proposed an approach how Deep Modeling can be achieved with a slight and non-invasive extension to existing and well-

accepted meta-metamodel such as Ecore. This brings us into the comfortable situation that we can apply Deep Modeling techniques such as non-transitive instantiation chains of arbitrary length with a self-describing and thus sound meta-metamodel and at the same time reuse all available tools to work with the models such as model transformation languages.

In particular, we can easily apply implicit incremental analyses and transformations to deep models.

At the same time, our approach circumvents paradox situations level-blind approaches to Deep Modeling have been blamed for in the past. We have applied our approach to the realistic scenario of creating a Deep Modeling version prototype of the popular Palladio Component Model (PCM) where we could simplify the generated API for the model and reduce the number of constraints necessary.

The approach does not only allow us to use our incrementalization system in the context of deep models, it also makes model analyses using this approach faster because the incrementalization can make use of dedicated change notification when a value for an instance reference changes instead of having to match and filter a multitude of changes of artificial connector elements.

Part V.

Validation and Evaluation

9. Validation and Evaluation

The goal of this chapter is to validate and evaluate the approaches and implementations of Parts II and III using several case studies. In particular, the goal is to demonstrate both the applicability of the presented approaches to a wide area of problems and to evaluate the approaches with regard to performance. Here, the most important metric is the response time from a model change to get an updated analysis result. The characteristics of the analysis result but also those of the changes made to the source model are specific to the case study.

The remainder of this chapter is structured as follows: At first, Section 9.1 introduces the goals of the validation. Section 9.2 briefly discusses how the case studies of this chapter achieve these goals. Section 9.3 discusses a case study on incremental queries in the railway domain, taken from the TTC 2015. Section 9.8 discusses a case study in the Smart Grid domain, submitted to and accepted at the TTC 2017. Section 9.4 discusses a case study for an incremental dataflow language, taken from the TTC 2016 Live Contest. Section 9.5 discusses the incrementalization of a bidirectional model transformation between state machines and Petri Nets. Section 9.6 presents a systematic analysis on the expressiveness of unidirectional synchronization blocks through a mapping from the ATL transformation language to NMF Synchronizations. Section 9.7 discusses a case study on model-based refactoring of Java code using a bidirectional model transformation to a simplified Program Graph model, taken from the TTC 2015. Section 9.9 presents a case study on bidirectional model transformations, taken from the TTC 2017. Section 9.12 discusses internal and external threats to validity. Finally, Section 9.13 summarizes the results of the validation.

9.1. Validation Goals

The correctness of the concepts for incremental model analyses and incremental model transformations has been formally proven, based on an abstract representation of these systems. Therefore, the correctness of the case study solutions is not in the center of the validation. Because the presented contributions work implicitly, we do not validate that they minimize the effort for developers.

The validation goals are thus as follows:

Applicability A formalism is only beneficial to developers if it can be applied to practical problems. For the incremental computation system, the applicability is clear, as the approach is based on a Turing-complete calculus. Nevertheless, we want to validate whether the usage is practical for common problems.

For model transformations, the situation is different as the concept of synchronization blocks is new. Therefore, it is not clear what kinds of model transformations can be expressed.

A similar statement holds for the presented approaches for meta-metamodel extensions. Even though refinements and structural decomposition are inspired by UML redefinitions and their applicability can be drawn from there, the lack of other implementations of this concept in the UML means that the applicability in practical problems is not well understood.

Performance The response time from an elementary model change to an updated analysis result is critical for incremental tools: Especially when change notifications are not of importance, the theory of incrementalization becomes useless if it is cheaper to recompute the entire analysis after each model change. Therefore, it is highly interesting to see whether the usage of incrementality actually improves the performance in the selected case studies.

For the meta-metamodel extensions, performance is with regard to the performance of model analyses based on metamodels that use these extensions. As argued in Chapter 7, several kinds of model analyses practically do not consume any time because they can be guaranteed through type system

guarantees. For others, it is questionable what the impact of refinements and structural decomposition to the performance of (incremental) model analyses is.

Memory Consumption Incrementality comes at the cost of memory that is required to save previously computed intermediate results of the analysis. Therefore, a goal of the validation is to find out how much memory is required. This again also holds for analyses based on metamodels that use the proposed modeling extensions.

Understandability One of the main goals of this thesis is to ensure an incremental evaluation of an analysis or transformation while maintaining the understandability of the batch specification. Therefore, it is an important validation goal to evaluate the understandability of our approach.

For the proposed meta-metamodel extensions, the goal to validate the understandability is twofold: For once, we want to validate whether model analyses built for metamodels using these extensions are more understandable, but secondly, we are also interested how understandable these metamodels are.

Correctness Even though the correctness is guaranteed by proofs, we use correctness indicators where available to make sure that the implementation is correct, at least with respect of what the used correctness indicators can tell.

Relaxation of Assumptions To validate the assumptions from Section 1.2, we are also interested how the contributions depend on these assumptions.

9.2. Validation Strategy

To tackle the validation goals, we use multiple cases that have been part of the Transformation Tool Contest (TTC). For these cases, solutions using other modeling technologies are available, often written by developers of the respective tools. Therefore, it is a reasonable assumption that these solutions are the best solutions possible with these tools, which therefore allows a

comparison of the tools. However, the cases at the TTC do not cover all aspects of this thesis and therefore, we added additional case studies.

For the proposed meta-metamodel extensions, we seek a comparison with other tools, in particular Deep Modeling tools, using the MULTI 2017 modeling challenge. Here, models according to multiple modeling standards could be submitted based on a common domain description. Additionally, we evaluate the incremental Deep Modeling analyses introduced in Section 8.4.

In the following, we present the strategy for all of the properties that we want to validate and evaluate.

Applicability For the applicability, it is important to have a broad range of problems in a multitude of different application areas. In particular, the multitude of cases from the TTC fulfill this criterion. In addition, the fact that all TTC cases except the Smart Grids case are not authored by the author of this thesis, a bias towards the approaches of this thesis can be excluded.

For the model transformation approach, we essentially picked all TTC cases from the most recent years that match the definition of a model transformation problem or a model analysis as in Section 1.2. In addition, we picked the live contest of the TTC 2016 that concentrates on creating a new transformation engine with a tightly defined execution semantics and an incremental execution.

To further validate the applicability of the model transformation approach from Chapter 6, we show that the language has roughly the same expressiveness as the declarative part of ATL, presumably the most common model transformation language in the community [203].

For the applicability of the meta-metamodel extensions, we created a solution to the Bicycle Shop modeling challenge at the MULTI workshop 2017. For this modeling challenge, similar to the TTC, a call for solutions was issued such that multiple models using other tools are also available.

Performance As indicated before, the most important performance metric in the context of this thesis is the response time from a model change to an updated analysis or transformation result. This metric is captured for all

input sizes if applicable. As the input sizes are hard to compare between the cases, we capture the following metrics in addition:

1. Speedup ranges with respect to batch execution
2. Speedup ranges with respect to other tools (incremental or non-incremental)

We then take the performance results of all case studies together to reason on the advantages and pitfalls of incremental computation.

To compare the results among case studies, all performance results are collected with an Intel i7-4710 CPU clocked at 2,50Ghz on a system with 16GB RAM. The only exception here are the performance measurements for the distributed incremental queries which are collected on a Microsoft Azure cloud cluster. All benchmarks are publicly available such that the results are reproducible. In most cases, the benchmark repositories also contain the R scripts that automatically reproduce diagrams such as depicted in this thesis.

Memory Consumption Similar to performance, we also measure the memory consumption for the case studies. Here, we are interested in the relative memory overhead caused by incremental computation. However, there are also many metrics available to measure memory consumption: peak main memory, average main memory, memory allocations or the working set size. The problem with average and peak main memory is that they are difficult to measure as they require a close monitoring of the process, which is often impractical. Furthermore, these metrics are inflated by the lazyness of garbage collection: Unless the system is under memory pressure, a process may easily consume more memory than necessary. Therefore, we have decided for the working set size as it is easy to measure and stable across multiple runs of a benchmark.

On the contrary, memory is always tied to a process which makes memory measurements hard in case multiple solutions run in the same process. Therefore, we do not measure the memory consumption in all case studies.

Understandability One of the merits of the TTC is that all solutions for a given case are presented at the TTC workshop and participants are given a short questionnaire about these solutions. The questionnaires are made by the TTC organizers, so there is little influence that solution authors have to these questionnaires[79]. Nevertheless, the responses often enable to reason on attributes such as understandability that are otherwise hard to manifest. Therefore, we review the available data from the open peer reviews and the questionnaire responses at the workshops for all of the TTC case studies.

The used questionnaires are available in the appendix for reference.

Correctness We expect all case studies to produce correct results. In fact, all correctness indicators in all case studies indicated correct solutions.

It is clear that these correctness indicators do not prove correctness of the system as they only show that at least in the considered cases, the case study solutions did not contain failures captured by these correctness indicators. Nevertheless, we think that checking these correctness indicators is better than not doing so. However, a complete formal proof of correctness of the implementation is left for future work.

To maximize the types of failures captured by the correctness indicators, we often use randomly generated change sequences such that a mixture of different types of changes occur, making the correctness checks more expressive.

Relaxation of Assumptions In order to apply the approaches presented in this thesis to the selected case studies, the problems in the case studies fit well into the assumptions made in Section 1.2. However, one case study leverages these assumptions, incrementalizing an imperative data flow with side-effects. Therefore, we analyze the effects when these assumptions are violated in the TTC 2016 Live Contest case study described in Section 9.4.

For an overview, we depict the validation goals of all case studies in Table 9.1 where checkmarks indicate that the validation goal was tackled by the respective case study.

[79] A negative example is the TTC 2015 where the questionnaire consisted only of a single question asking for the overall evaluation of the solution.

Section	Domain	Artifact	Origin	Speedup wrt. batch	Speedup wrt. other solutions	Memory Consumption	Understandability	Correctness
9.3	Railway	Pattern Matching	TTC 2015	✓	✓	✓	✓	✓
9.4	multiple	HOT	TTC 2016	✓	✓	–	–	✓
9.5	synthetic	BX	–	✓	✓	–	–	✓
9.6	multiple	HOT	–	✓	✓	–	–	✓
9.7	Java refactoring	BX	TTC 2015	–	–	–	✓	✓
9.8	Smart Grids	Model Views	TTC 2017	✓	✓	✓	✓	✓
9.9	synthetic	BX	TTC 2017	✓	✓	–	✓	✓
9.10	synthetic	Metamodels	MULTI 2017	–	–	–	–	–
9.11	synthetic	Pattern Matching	–	✓	✓	–	–	✓

Table 9.1.: Summary of the case studies presented in this thesis

In the following sections, we present the case studies. Most case studies have the same structure. At first, they present the benchmark setup, i.e. briefly introduce the problem and the tasks that had to be solved by benchmark solutions. Next, we explain the validation goals for the respective case study. Then, the NMF solution is presented before results from open peer reviews, questionnaire responses at the TTC workshop and performance measurements are discussed. Lastly, the presentation of each case study concludes with a summary of the case study.

9.3. Case Study: Incremental Queries on Railway Models

In this section, we analyze the Train Benchmark case [197] at the TTC 2015 to which an NMF solution was submitted [105]. This benchmark consists of five

analysis queries. The only incremental tools that participated in this contest were VIATRA Query[80] from the case authors and the NMF solution.

At the TTC 2015, solutions to the benchmark were submitted using NMF [105], FunnyQT [114], ATL [211], VIATRA Query [194] and Sigma [130].

9.3.1. Benchmark Setup

In the scope of this case study, we only investigate the TTC version of the Train Benchmark [197] that covers only a subset of the entire Train Benchmark [196]. We briefly describe the benchmark setup here, but further details can be found in these papers.

Besides the *SwitchSet* query briefly introduced in Section 1.6, the benchmark included four other queries: *PosLength* queries the segments in the railway network that have length 0 or less. *SwitchSensor* looks for switches without a corresponding sensor. *RouteSensor* looks for switch positions along a route that refer to switches that are not part of a sensor that is defined within that route. The most complex *SemaphoreNeighbor* query searches for routes that end at given track segment (which means that the next track segment belongs to a different route), but the exit semaphore of the route is not the entry semaphore of the route that starts with the connected segment.

The benchmark consists of four phases depicted in Figure 9.1: *Read*, *Check*, *Repair* and *Recheck*. In the *Read* phase, solutions of the benchmark simply load the model of the respective size. In the *Check* phase, the number of invalid elements is computed, i.e. the number of pattern matches where each pattern match represents a constraint violation. In the *Repair* phase, several of these (either 10 or 10% of all, determined by the parameter *Change set size*) constraint violations are fixed. After that, the number of invalid elements is refreshed in a *Recheck* phase of the benchmark. For each combination of input model (size) and change set size, the benchmark is run five times for each solution. Within each run, phases *Repair* and *Recheck* are repeated ten times.

During the execution of these phases, the benchmark collects metrics on execution time, number of invalid elements and memory usage.

[80] In 2015, VIATRA Query was called EMF-IncQuery.

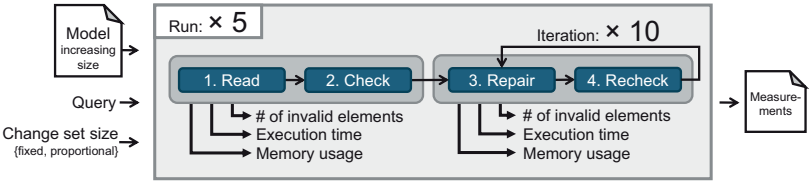


Figure 9.1.: Phases of the Train Benchmark [197]

In the scope of this case study, we are specifically interested in incremental revalidation, i.e. the time for *Repair* and *Recheck*.

To evaluate Incerator, we had to slightly modify the benchmark. By default, the only changes that are done during the benchmark execution are fixes of the constraint violations of the respective queries, resulting in very homogeneous change sequences. However, in a real world application, changes to the model are rather heterogeneous: switch positions, semaphore signals and lengths of segments might change arbitrarily. Therefore, we adapted the benchmark to fix constraint violations only in 20% of the changes. Otherwise, we perform random changes in the model. As we think that changes to switch positions or semaphore signals are the most common in this scenario, they also occur most often in the benchmark.

9.3.2. Validation Goals

This case study aims to evaluate the following aspects:

Applicability The case study is conducted in the railway domain, which is a very interesting example of a cyber-physical system. Furthermore, pattern matching is very interesting as this kind of analyses is typical in many domains.

Performance The case study yields a very good opportunity to evaluate the performance that can be achieved for incremental model analyzes using the incremental SQO implementations presented in Section 4.4. In particular, it allows us a comparison with VIATRA Query, a tool dedicated to incremental pattern matching.

Furthermore, we use this case study to validate whether distributed computing through Orleans or Incerator can give any additional performance benefits.

Memory Consumption We use the Train Benchmark to also measure the memory consumption induced by incremental computation. In particular, we are interested in the memory overhead required to run incremental computations. Further, we want to know how the memory requirements of our approach relates to other approaches, in particular VIATRA Query.

Furthermore, we want to evaluate whether and to what degree the memory consumption can be reduced by Incerator automatically.

Understandability As a TTC case study, the Train Benchmark case gives us the possibility to evaluate qualities of our solution that are hard to measure other than through perception. In particular, we are interested to see how the understandability of our approach compares to other solutions.

Correctness The Train Benchmark has a correctness indicator: The number of constraint violations after each *Check* or *Recheck* phase. We use these indicators to check correctness. To reproduce the exact same sequence of random numbers, the solution uses a random number generator that produces the exact same numbers as the random number generator built into Java.

9.3.3. The NMF Solution

The description of the NMF solution to the Train Benchmark is based on the original submission to the TTC 2015 [105].

In the solution, we use NMF Expressions to incrementally query the source model and cache the invalid elements continuously. However, this means that the phases *Repair* and *Recheck* get merged as the model manipulation automatically updates the incremental query result. In particular, the *Recheck* phase get meaningless as the updated results are always available and could be used for immediate feedback, while more computational effort is put to the model manipulation tasks in the *Repair* phase.

The implementations of the five tasks described in [197] is described below[81].

In the following we present the solution to the tasks, following the structure of the case description, though with omitted sort keys.

Please note that the parameter names such as `pattern` or `action` are optional, we only included them for better understandability.

```
1 Fix(pattern: rc.Descendants().OfType<Segment>()
2            .Where(seg => seg.Length <= 0),
3    action: segment => segment.Length = -segment.Length + 1);
```

Listing 9.1: NMF implementation of the *PosLength* query

The implementation of the *PosLength* query is depicted in Listing 9.1. It uses the `Descendants` operation of NMF to iterate all models contained somewhere in the railway container. To this collection of model elements, a type filter is applied that restricts the collection to instances of `Segment`. This collection of segments is finally filtered to those that have a length below 0. The repair operation simply sets the length as suggested in the case description 1.6.

```
1 Fix(pattern: rc.Descendants().OfType<Switch>()
2            .Where(sw => sw.Sensor == null),
3    action: sw => sw.Sensor = new Sensor());
```

Listing 9.2: NMF implementation of the *SwitchSensor* query

The implementation of *SwitchSensor* is depicted in Listing 9.2 and works very similar, though this time, elements of type `Switch` are selected and filtered for those where no sensor is attached. The repair operation creates a new sensor and assigns this to the `Sensor` property of the selected switch. Note that because `Sensor` is a container property, this moves the switch out of the model.

[81] The original NMF solution as depicted in [105] is slightly different. The difference between these versions is analyzed in Section 10.1.

```
var routes = rc.Routes.Concat(rc.Invalids.OfType<Route>());
Fix(pattern: from route in routes
          where route.Entry != null
            && route.Entry.Signal == Signal.GO
          from swP in route.Follows
          where swP.Switch.CurrentPosition != swP.Position
          select swP,
    action: swP => swP.Switch.CurrentPosition = swP.Position);
```

Listing 9.3: NMF implementation of the *SwitchSet* query

The implementation of the *SwitchSet* query was already explained in Section 1.6. We depict it again in Listing 9.3. This query (and all of the below) is based on a collection of routes. Because routes can only appear in two places, namely their correct location in the containment hierarchy and in the `invalids` reference, we make this more explicit to only look in these two places than traversing the entire containment hierarchy. Note that this does hardly make any difference for the incremental runtime because the containment hierarchy of the model is not touched in most queries. The repair operation of the *SwitchSet* query simply sets the current position of the switch to the position required by the route.

```
Fix(pattern: from route in routes
          from swP in route.Follows
          where swP.Switch.Sensor != null &&
           !route.DefinedBy.Contains(swP.Switch.Sensor)
          select new { Route = route, Sensor = swP.Switch.Sensor },
     action: match => match.Route.DefinedBy.Add(match.Sensor));
```

Listing 9.4: NMF implementation of the *RouteSensor* query

The implementation of *RouteSensor* is depicted in Listing 9.4. It iterates through all routes and all switch positions defined by these routes and selects those where sensor of the corresponding switch is not defined in that route. The repair action simply adds the sensor to the collection of sensors of that route. Because `DefinedBy` is a containment reference, this again moves the sensor within the model.

```
var connectedRoute = ObservingFunc<IRoute, IRoute>.FromExpression(
  route => (from sensor1 in route.DefinedBy
          from te1 in sensor1.Elements
          from te2 in te1.ConnectsTo
          where te2.Sensor != null && te2.Sensor.Parent != route
          select te2.Sensor.Parent as IRoute).FirstOrDefault());

Fix(pattern: from route1 in routes
           let route2 = connectedRoute.Evaluate(route1)
           where route2 != null && route2.Entry != route1.Exit
           select new { Route1 = route1, Route2 = route2 },
  action: match => match.Route2.Entry = match.Route1.Exit);
```

Listing 9.5: NMF implementation of the *SemaphoreNeighbor* query

The implementation of the *SemaphoreNeighbor* query is twofold as depicted in Listing 9.5. We use a helper function to compute the route that follows the current route of a route provided as input. For this, we iterate through the sensors of the route, iterate through all of its track elements, iterate through the connected elements and to those where the connected element is defined in a different route than the current one. The railway network allows at most one of such next routes. In the actual pattern, we then iterate through the routes, find the candidate for the next route and save it as a local variable, then filter this pair of routes to check that the entry of that route is not the same as the exit of the first route. The repair operation simply sets the entry of the second route to the exit semaphore of the first route.

The solution for the *SemaphoreNeighbor* query as depicted here differs from the one originally published [105]. We discuss the differences of the original and the improved version in Section 10.1.

The solutions to *SwitchSet*, *RouteSensor* and *SemaphoreNeighbor* use the query syntax of C# (cf. Section 2.3). This syntax is translated to the method chaining syntax by mapping the query keywords like `from` or `where` to SQO method calls supported by NMF Expressions. The `let` expression in the *SemaphoreNeighbor* query is converted to a `Select` method that maps a route to a pair of routes, represented by an anonymous type.

Because NMF Expressions allows to use the same specification both in a classic batch manner as also incrementally, our solution can also be configured to run in batch mode without any changes to the patterns. When executed in batch mode, NMF Expressions simply forwards the call to the Language

```
1  public void Fix<T>(IEnumerableExpression<T> pattern, Action<T> action) {
2    var patternInc = pattern.AsNotifiable();
3    foreach (T element in patternInc) {
4      action(element);
5    }
6    patternInc.CollectionChanged += (o,e) => {
7      if (e.NewItems != null) {
8        foreach (T element in e.NewItems) {
9          action(element);
10       }
11     }
12   }
13 }
```

Listing 9.6: A simplified implementation of the `Fix` function

Integrated Query (LINQ) to objects implementation. Besides a negligible runtime compilation effort, this utilizes the highly optimized platform LINQ implementation.

The patterns are *enumerable expressions* where developers can choose at runtime whether the pattern should be executed in batch mode or whether NMF Expressions should register for elementary change notifications to keep a cache of the result up to date. To specify patterns, we created a small method `Fix` that captures them.

The easiest implementation for the `Fix` function repairing any validation error as soon as they occur would be the one presented in Listing 9.6. In Line 2, we tell NMF Expressions that we want to obtain incremental updates for the given pattern. Line 3 repairs all occurences existing so far and Lines 4-8 handle new pattern matches. For the benchmark, we adopted the `Fix` function to account for the benchmark phases. In particular, the implemented version takes a third parameter to allow us to sort matches. Since these sort keys offer little insight, we omit them in the pattern presentation.

Tool	Correctness & Completeness	Conciseness	Readability
ATL/EMFTVM	15	12	13
VIATRA Query	12.5	12.5	12.5
FunnyQT	15	15	12.5
NMF	12.7	13.3	15
SIGMA	15	13.3	13.3

Table 9.2.: Results from the open peer review of the TTC 2015 Train Benchmark

9.3.4. Benchmark Results

The number of pattern matches is correct after each iteration.[82]

The results from the open peer reviews[83] are depicted in Table 9.2.

For the understandability, the NMF solution was the only solution at the TTC 2015 contest that received full points for understandability from all open peer reviewers. In particular, the solution was evaluated to be more understandable than all batch solutions written in FunnyQT [114], ATL/EMFTVM [211] or Sigma [130].

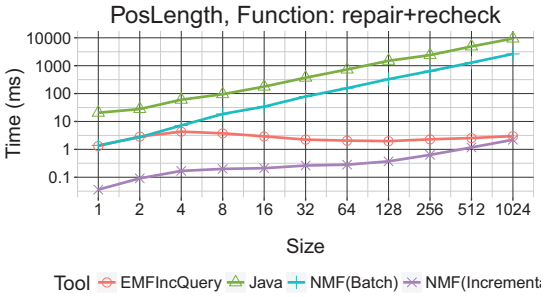
The performance measurements for the revalidation for all queries are depicted in Figure 9.2. We use the same size notation as in the Train Benchmark where a size 1 corresponds to about 1,300 model elements. In the largest considered size 1024, the model contains about 1.5 million model elements.

The NMF solution supported two execution modes, incremental and batch mode. In the batch mode, the analysis is rerun on the entire model in each step whereas the incremental version maintains intermediate results and invalidates them whenever elementary changes appear in the model.

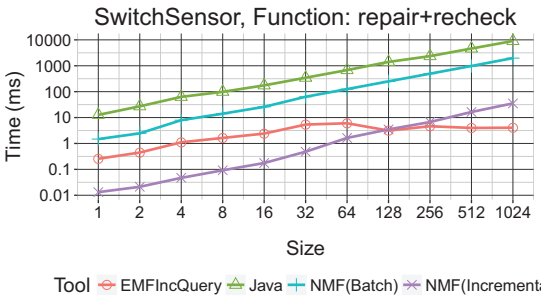
The results show that the incremental version of the NMF solution was able to keep up with specialized tools such as VIATRA Query for many model sizes and queries and even beat VIATRA Query by roughly a magnitude

[82] The open peer review refers to a preliminary version that had a minor flaw. Therefore, the scores for correctness are lower in the open peer reviews.

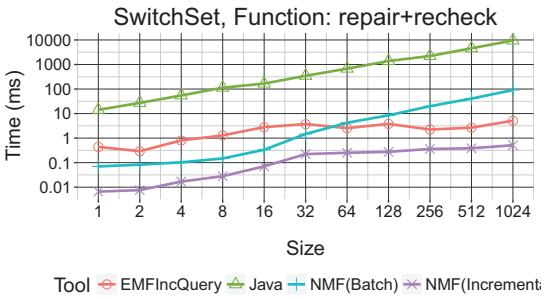
[83] `https://docs.google.com/spreadsheets/d/1WepbTGB8XbXFV6tYKDsdOn9kFvai8c4q_EoszeV3FsI/edit?usp=sharing`, retrieved 26 Sep 2017



(a) *PosLength*

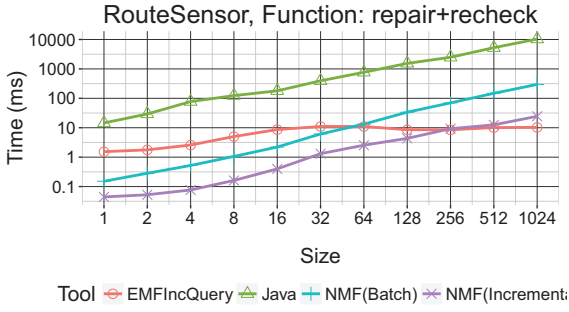


(b) *SwitchSensor*

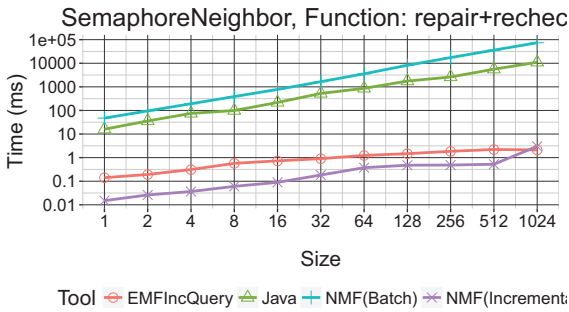


(c) *SwitchSet*

Figure 9.2.: Performance results for revalidation. The graph compares the NMF solution (batch and incremental mode) with the reference solutions in Java and VIATRA Query. Both axes are logarithmical.



(d) *RouteSensor*



(e) *SemaphoreNeighbor*

Figure 9.2.: Performance results for revalidation. The graph compares the NMF solution (batch and incremental mode) with the reference solutions in Java and VIATRA Query. Both axes are logarithmical (cont.).

in the *SwitchSet* query that we used in the motivational example. In other scenarios, our implementation eventually gets slower than the VIATRA Query solution.

Figure 9.2 also shows that for model sizes up to roughly 100,000 model elements (size 64), the incremental NMF solution was the fastest for all queries depicted.

To explain why the NMF solution is faster than VIATRA Query for the *SwitchSet* query, we depicted the Rete network created by VIATRA Query for the *SwitchSet* query introduced as running example for this thesis in Figure

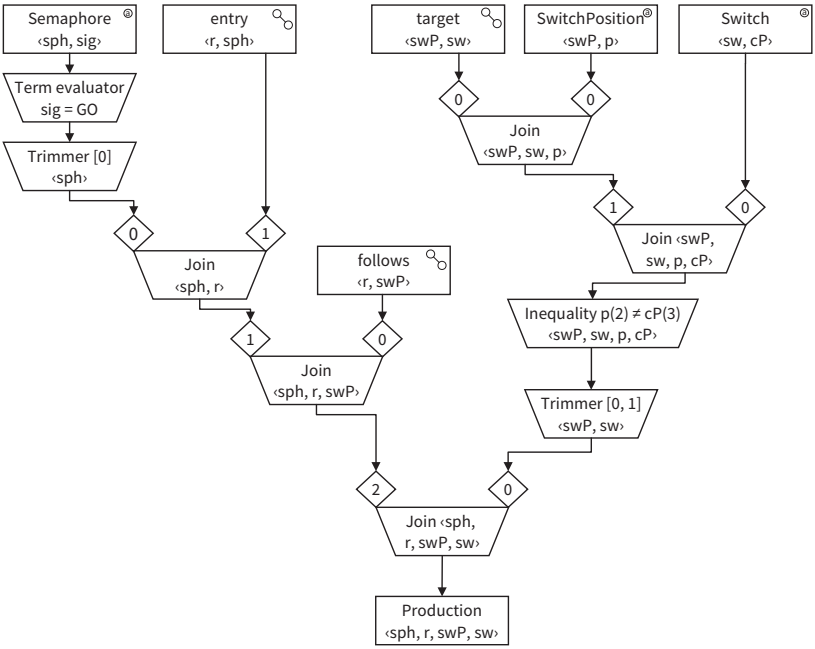
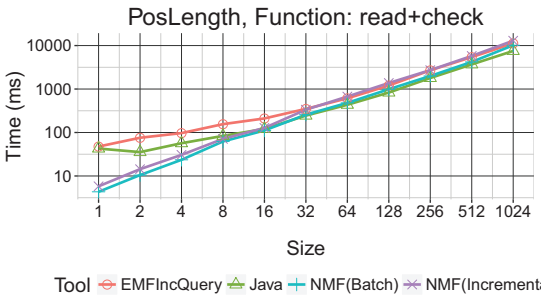


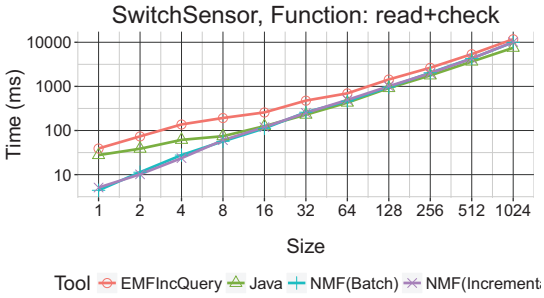
Figure 9.3.: Rete network created by VIATRA Query for the *SwitchSet* query of the Train Benchmark (cf. [195])

9.3. In the running incremental analysis, each node depicted in this graph represents a list of partial pattern matches. The network combines simple references and attribute accesses through *Join* nodes to pattern matchers. In the contrary, nodes in the DDG of NMF Expressions only represent an evaluation of an attribute or reference for a single model element.

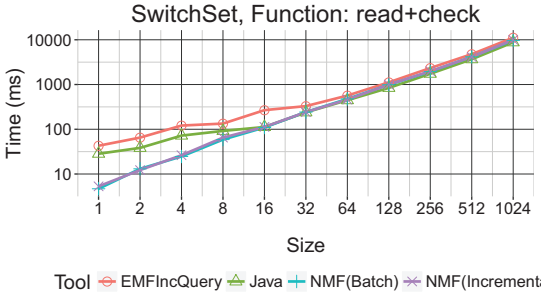
Besides the different granularity, the path in the data structure for a given change is different. In the NMF solution, a change of a signal position only affects the respective attribute evaluation node, the binary expression node that checks equality, the node for the `where` operator and lastly the node for a compiler-generated `select` operator. Meanwhile, the same change in the Rete network depicted in Figure 9.3 has to be propagated through seven nodes where each node handles much more data and therefore has a higher computational complexity.



(a) *PosLength*

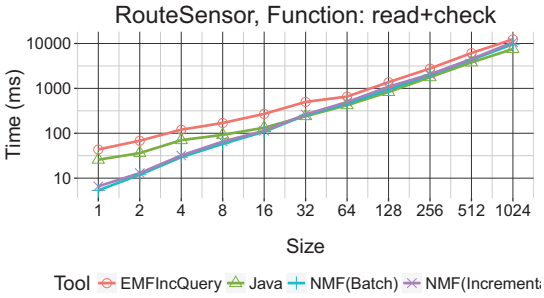


(b) *SwitchSensor*

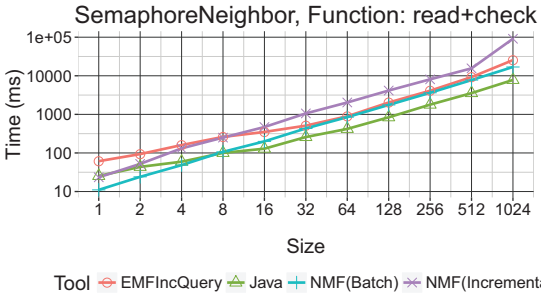


(c) *SwitchSet*

Figure 9.4.: Performance Results for batch validation of the NMF solution versions compared to the reference solutions in Java and VIATRA Query (both axes logarithmical)



(d) *RouteSensor*



(e) *SemaphoreNeighbor*

Figure 9.4.: Performance Results for batch validation of the NMF solution versions compared to the reference solutions in Java and VIATRA Query (both axes logarithmical) (cont.)

The results for the batch validation are depicted in Figure 9.4. Both NMF solutions had a relatively low constant overhead, indicated by the fact that the solutions were much faster than VIATRA Query or Java solutions. For larger models, the overhead of the incremental algorithm to set up caches for immediate results and register event handlers is similar to the query effort done in the batch mode.

In the *SemaphoreNeighbor* query, the overhead of creating the DDGs for the incremental revalidation is slightly higher than in the other cases due to the higher complexity of the query.

The incremental version has a drawback against the batch version: memory consumption. We experienced that the incremental version did not allocate

more memory than the batch version (since batch analysis has to reallocate memory for each evaluation run) but the DDG that is responsible for most of the memory consumption is continuously required and cannot be released until it is detached from the model. When the analysis is run in batch mode, the memory allocated to compute the analysis can be released once the analysis result is available.

However, since both the .NET runtime and the Java Virtual Machine employ garbage collection, the memory consumption is difficult to measure because memory no longer used may still be allocated because the exact time of a garbage collection is not known. To at least get an impression on the memory consumption, we depicted the working set of the benchmark queries in Figure 9.5.

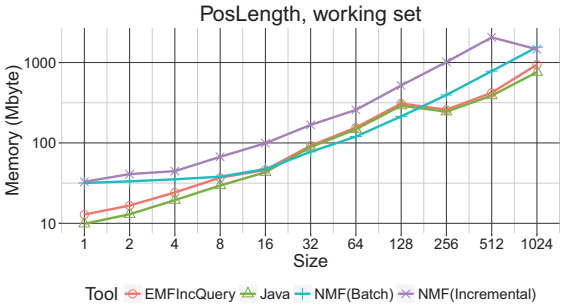
The results show that the working set was within half an order of magnitude difference as the Java or VIATRA Query solution, at least for the queries *SwitchSensor*, *SwitchSet* and *RouteSensor*. The working set also remained within the limit of 2GB even for the largest models which is why we think that the memory requirements are feasible for this scenario.

For the *PosLength* query, the memory consumption is very high, simply because of the large amount of segments that are contained in the model. For each segment, a DDG to check whether its length is less or equal to zero has to be instantiated. On the contrary, the model manipulation performed for the *PosLength* query is computationally inexpensive. Therefore, the overhead due to incremental computation is relatively higher.

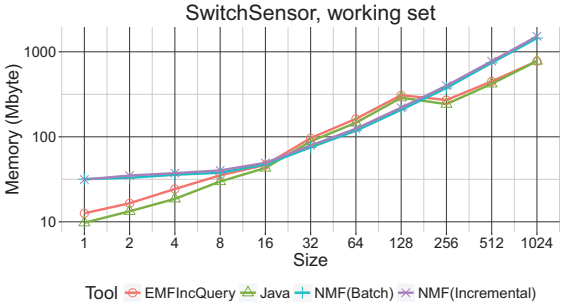
For the *SemaphoreNeighbor* query, this result is different. Rather, the memory consumption of the incremental execution mode is about an order of magnitude higher than the memory consumption of the batch mode execution and all solutions required much larger amounts of memory. We guess that this is due to the different query operators used, as especially the `SelectMany` operator turns out to be very memory intensive in the incremental setting.

9.3.5. Incerator Results

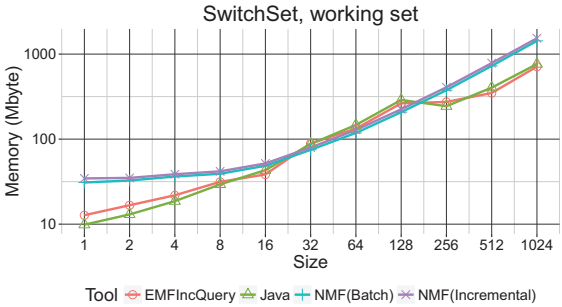
In this section, we present the results applying Incerator to the NMF solution of the TTC version of the Train Benchmark.



(a) *PosLength*

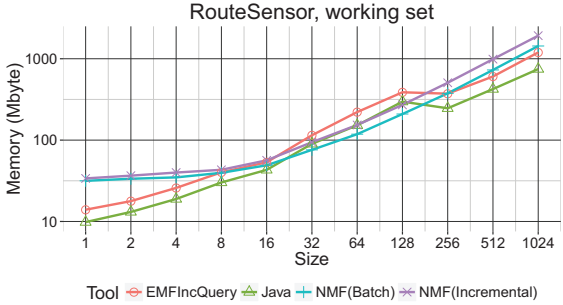


(b) *SwitchSensor*

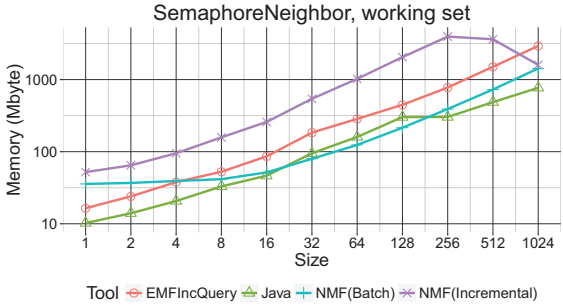


(c) *SwitchSet*

Figure 9.5.: Working sets in the Train Benchmark against relative model size (both axes logarithmical).



(d) *RouteSensor*



(e) *SemaphoreNeighbor*

Figure 9.5.: Working sets in the Train Benchmark against relative model size (both axes logarithmical) (cont.).

The original NMF benchmark solution lets users choose based on configuration whether the analysis should be run incrementally or in batch mode. In the scope of the evaluation of Incerator, we always use the incremental setting as otherwise the configuration does not have any effect. However, this means that the query operators in the benchmark queries are always executed incrementally and the configuration only influences the predicates used in the query operators. As a result, the design space of most queries is rather small ($\leq$ 27 configurations) as they contain only a limited number of query predicates and thus, we can perform a full design space exploration. Only the *SemaphoreNeighbor* query contains 10 predicates, making up a

design space of 59,049 configurations, though the implementation reduces this search space to 2,187 configurations through heuristics[84].

To apply Incerator, we created an adapter that executes the Train Benchmark for a given configuration and extracts the revalidation times from the benchmark output. Our optimization tool then uses this adapter to first run the benchmark collecting variation points for the incrementalization and then runs this analysis again for different configurations.

To evaluate the influence on different configurations to the performance of the benchmark, we first ran Incerator on the smallest benchmark input model with 1,311 model elements. To evaluate the influence of the model size, we then repeated the measurements on a medium-sized input model with 50,765 model elements (size 32) and compared the results to size 1. For each configuration, we recorded the *Recheck* and *Revalidation* phase of the benchmark that are the most interesting in our setting.

Figures 9.6 and 9.7 show the results for the smaller input size.

The *PosLength* query is very simple and therefore, the dependency graph of instruction-level incrementality is rather small. Instruction-level and argument promotion strategies yield very similar results. The strategy to reevaluate the segments on repository changes yields significantly worse results since all segments have to be reevaluated, not only the changed ones. This effect is more drastic in size 32 where listening to repository changes yields worse response times by a factor of almost 8000.

For the *SwitchSensor* query with size 1, the strategy to listen for repository changes is not as bad, simply because the model contains 1,010 segments but only 44 switches. However, the truely incremental strategies yield better results. While instruction-level incrementality is fastest for input size 1, argument promotion is slightly faster for size 32 but only by a factor 1.03.

The *SwitchSet* query contains five degrees of freedom and we thus see more data points (27). Incerator detects that indeed, the dynamic dependency graph of Figure 4.1 can be contracted into the dependency graph of Figure 5.1, similarly for the predicate in Line 4 of Listing 1.1. This yields a speedup

[84] Predicates that only consists of a single property access are directly set to instruction-level incrementality.

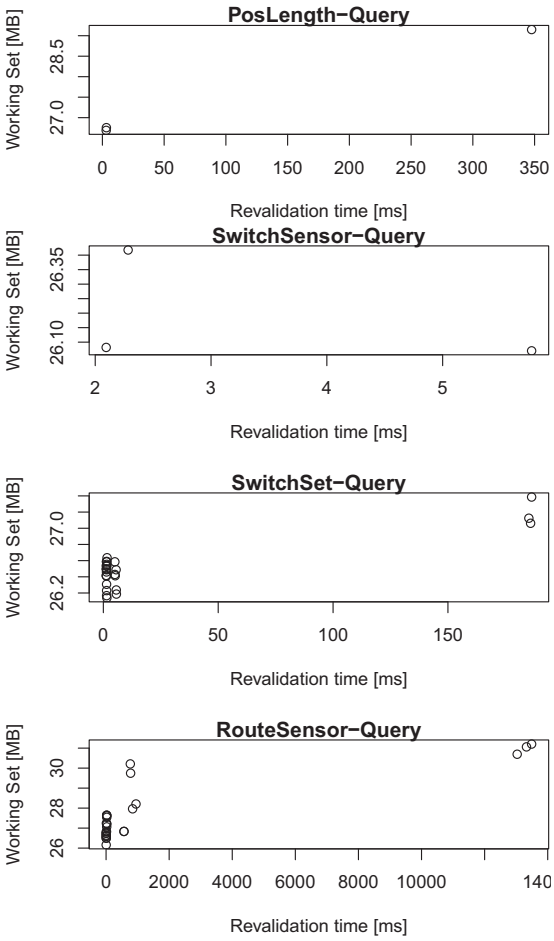


Figure 9.6.: Design space of different configurations for different validation queries of the Train Benchmark run with size 1
(1,311 model elements)

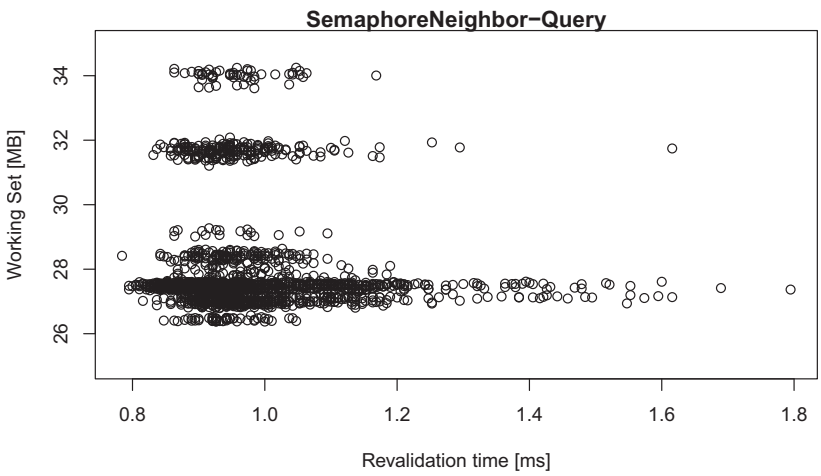


Figure 9.7.: Design space of different configurations for the *SemaphoreNeighbor* query of the Train Benchmark run with size 1 (1,311 model elements)

of 1.15 compared to instruction-level incremental solutions, both in size 1 and 32.

For the *RouteSensor* query with 27 possible configurations, the configuration with the best performance is the instruction-level incrementality for size 1 but for size 32, the argument promotion strategy is better by a speedup factor 2.22. This cannot be seen in the diagram, because the usage of inappropriate tuple types by the C# compiler causes many configurations to have an extraordinarily bad performance. However, Incerator is able to detect such a pitfall.

The design space of the *SemaphoreNeighbor* query depicted in Figure 9.7 is the largest from the queries of the TTC Train Benchmark. While we can see a lot of configurations that produce sub-optimal results, there is clearly visible front of pareto-optimal configurations. The fastest configuration for size 1 is faster than pure instruction-level incrementality by a factor of 1.23 but consumes slightly more memory. For size 32, the fastest configuration is faster than instruction-level incrementality by a factor 1.20 but only consumes 34% of the memory.

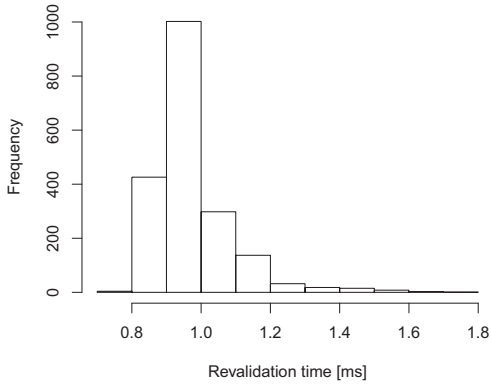


Figure 9.8.: Histogram of the revalidation times for the *SemaphoreNeighbor* query of the Train Benchmark run with size 1 (1,311 model elements)

Interestingly, the most configurations of the *SemaphoreNeighbor* query achieve similar revalidation times. To visualize this fact, we plotted a histogram of the revalidation times only in Figure 9.8. Roughly half of the configurations achieve similar revalidation times between 0.9 and 1.0 milliseconds, meanwhile the fastest configuration is only slightly below 0.8. Therefore, the differences within the configurations is much smaller than in the simpler queries such as the *RouteSensor*.

To visualize the influence of the input model size, we have plotted the ranks in the average times in Figure 9.9.

The results show that for the *PosLength* query, the order remained essentially the same. In the *SwitchSensor* query, the order of instruction-level and argument promotion strategies is exchanged, but the latter is only 3% faster for size 32. For the *SwitchSet* and *RouteSensor* queries, we can see that the worst configuration for size 1 are also the worst configurations for size 32. This does not always hold for the configurations that achieved the best (lowest) average revalidation times.

The reason here is that there are some configurations with very bad performance characteristics, shown in the upper right corners of Figure 9.6. Many of these configurations have the problem that the compiler-generated tuple types do not override the equality operators and therefore, tuples of

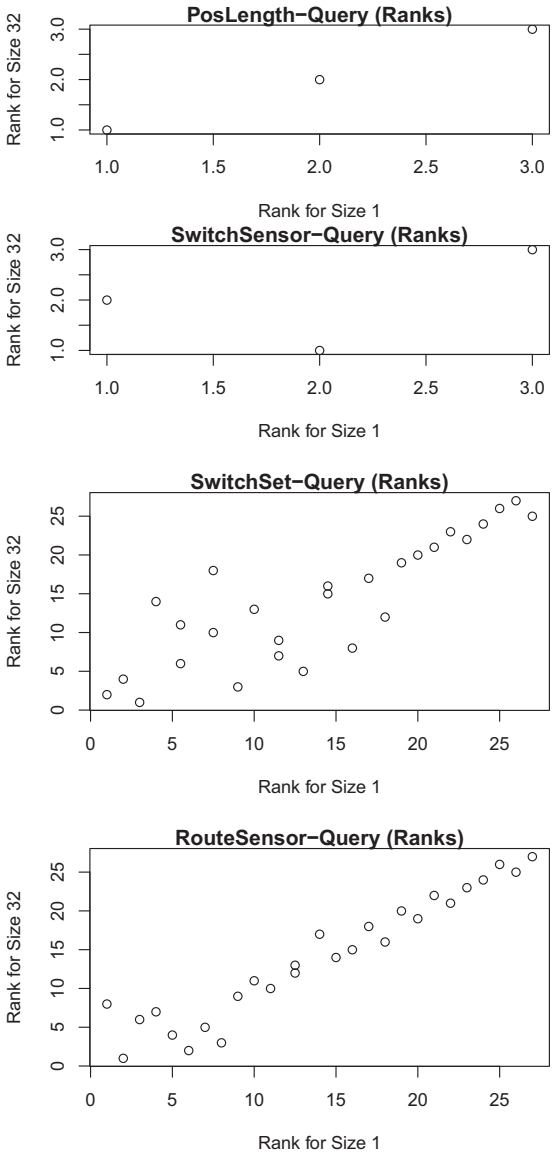


Figure 9.9.: Ranks of average revalidation times for size 1 (1,311 model elements) and 32 (50,765 model elements).

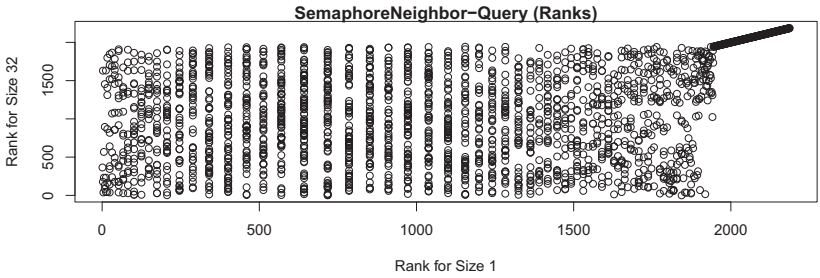


Figure 9.10.: Ranks of average revalidation times for size 1 and 32 for the *SemaphoreNeighbor* query

the same elements appear differently. This often causes dramatically worse revalidation times, quite regardless of the size of the input model.

For the configurations that are not affected by these effects, there seems to be a threshold amount of elements for which generating a dedicated DDG node type through the argument promotion strategy is beneficial or not.

For the *SemaphoreNeighbor*-query, depicted in Figure 9.10, we see that the rank for size 1 is mostly uncorrelated with the rank for size 32, except for the worst configurations that imply a bad performance both for size 1 and 32. This implies a limited validity for the obtained configurations for input models of a different size.

Concluding this analysis of results, we see that Incerator has a great potential to further improve non-functional properties of incremental model analyses, but one has to be very careful with these results as they are specific not only to the characteristic of the model analysis and the change sequences applied to the input model, but also even depend on the size of the input models.

9.3.6. Distributed Computing

We used the Train Benchmark to also evaluate the performance properties of the extension of NMF Expressions to incremental computing. This section presents the most important results. For more details, the interested reader is referred to the master thesis of Benjamin Wanner [214], the original source of this evaluation.

```
await Fix(
 modelPattern: await
  modelContainerGrain.SimpleSelectMany(
    model => model.RootElements.Single().As<RailwayContainer>()
      .Descendants().OfType<ISegment>(),
    factory, scatterFactor)
  .Where(seg => seg.Length < 0, GetScatterFactor(scatterFactors, 1))
  .ToNmfModelConsumer(),
 action: seg => modelContainerGrain.ExecuteSync(
  (container, elementUri) =>
  {
    var localSegment = (ISegment)container.Resolve((Uri)elementUri);
    localSegment.Length = -localSegment.Length + 1;
  }, seg.RelativeUri));
```

Listing 9.7: The solution to the *PosLength* query adapted for distributed computing

We deployed multiple silos and a client as an Azure cloud service. Unless stated otherwise, we used three *D2 V2* nodes for the grain silos and another one for the client. Each instance has two cores on a 2.4 GHz Intel Xeon E5-2673 v3 CPU and 7 GB of memory available. However, because resources in Azure are shared, other virtual machines may put load on the hardware or the network.

The code for evaluation has been published to GitHub[85]. It uses an adapted version of NMF Expressions that allows a distribution in Microsoft Orleans. This adapted version is also available online[86].

The implementation of the *PosLength* query is depicted in Listing 9.7. It shows that the distributed implementation adds some boilerplate code in order to specify scatter factors and specify a borderline what code should be run in a distributed manner. Also the action as a result of a fix has to resolve the local element before actually performing a change.

In particular, the Orleans framework requires developers to use asynchronous methods, in C# supported through the `await` statement. Further, the fact that we need to specify scatter factors and a factory of stream implementations means that the query syntax is no longer available and we have to use the

[85] https://github.com/bwanner/TrainBenchmarkNMFOrleans

[86] https://github.com/NMFCode/NMF/tree/distributed-expressions-orleans

method chaining syntax. Further, the border of responsibilities between the Orleans cluster and the local .NET application must be defined explicitly with a call to `ToNmfModelConsumer` in Line 8. The model manipulation must be performed in the Orleans master model instead of a local copy of the model. This complicates the model manipulation done as part of the *Repair* phase as depicted in Lines 9-14 of Listing 9.7.

The distributed implementation of the queries *SwitchSensor*, *SwitchSet* and *RouteSensor* is very similar to Listing 9.7. The simplicity of these queries allowed the benchmark to execute model sizes 2048 in a single node. For larger models, the main memory of the nodes does not suffice for the nodes.

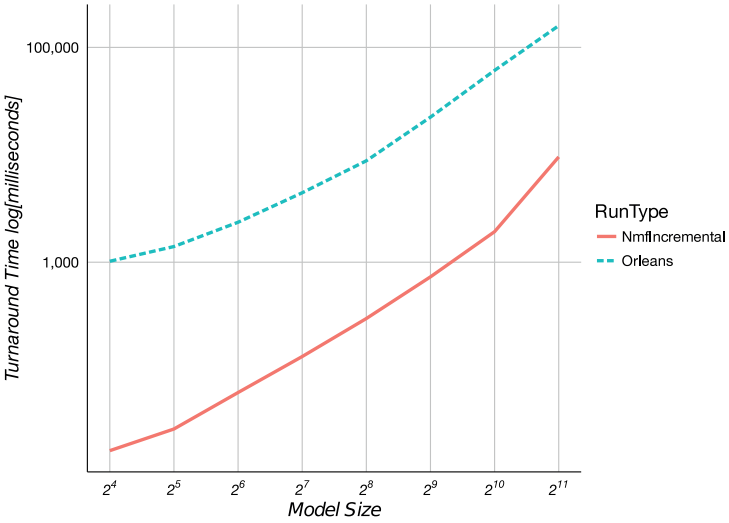
This is different for the distributed version of the *SemaphoreNeighbor* query that is based on the original NMF implementation (cf. Section 10.1) and has a very high memory consumption. This case essentially emulates the case that a DDG gets too large to fit into the main memory of a single node. The implementation of this query is depicted in Listing 9.8.

The performance results in terms of turnaround times for the entire benchmark execution is depicted in Figure 9.11 for different model sizes.

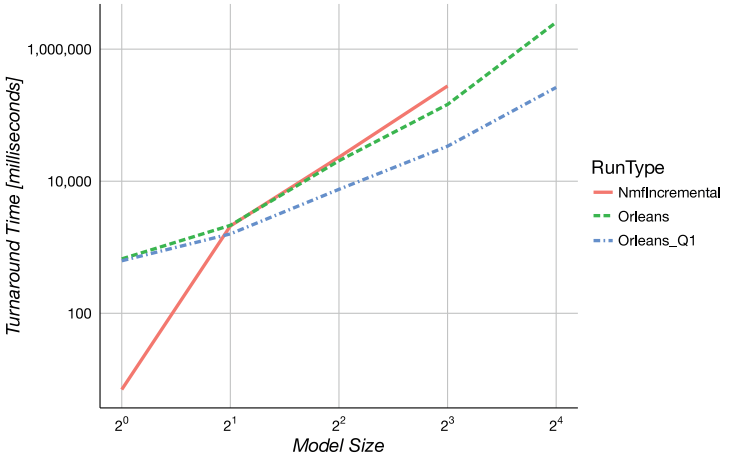
In the *PosLength* query, it is easy to see that the distributed implementation is more than a magnitude slower than the local implementation. This is due to the communication overhead: For each grain, the system has to query the master grain on which silo a connected grain is currently activated. This unnecessarily slows down the computation as also the grains themselves are rather computationally simple (they only contain a very simple filter condition).

This is different for the *SemaphoreNeighbor* query where the grains have a larger size. As a larger part of the query – lines 7-23 of Listing 9.8 – is condensed into a single grain, a higher proportion of computation is processed locally. Therefore, the difference between the local implementation and the distributed implementation is much smaller. Due to the utility of parallel resources in the distributed setting, the distributed implementation is also slightly faster.

More importantly, the distributed setting supports a larger model size. The original NMF solution to the *SemaphoreNeighbor* query used a DDG with more than 6 million nodes for the model size 8, growing quadratic with the model size. Therefore, a distribution to three nodes allows to execute the benchmark with model size 16.



(a) *PosLength* query



(b) *SemaphoreNeighbor* query

Figure 9.11.: Performance results for the Train Benchmark in incremental setting

```
modelContainerGrain.SelectMany(
  model => model.RootElements.Single().As<RailwayContainer>()
    .Descendants().OfType<IRoute>(),
  (model, route) => new ModelElementTuple<Model, IRoute>(model, route),
    factory, multiplex)
.ProcessLocal(enumerable =>
{
  var routePairs = enumerable.SelectMany(
    tuple => tuple.Item1.RootElements.Single().As<RailwayContainer>()
      .Descendants().OfType<IRoute>(),
    (tuple, route) => new {r1 = tuple.Item2, r2 = route});

  var res = from tuple in routePairs
    where tuple.r1 != tuple.r2 && tuple.r2.Entry != tuple.r1.Exit
    from sensor1 in tuple.r1.DefinedBy
    from te1 in sensor1.Elements
    from te2 in te1.ConnectsTo
    where te2.Sensor == null || tuple.r2.DefinedBy.Contains(te2.Sensor)
    select new ModelElementTuple<IRoute, IRoute, ITrackElement,
      ITrackElement>(tuple.r1, tuple.r2, te1, te2);

  return res;
})
.ToNmfModelConsumer();
```

Listing 9.8: The solution to the *SemaphoreNeighbor* query adapted for distributed computing

9.3.7. Summary

The queries and repair transformations demonstrate why we have sticked to the C# language. We think that it is very hard to get a more concise textual solution for this case and this opinion has been confirmed by a very good evaluation of the understandability for the NMF solution. At the same time, developers get the full tool support from e.g. Visual Studio and the query syntax that we use is used by thousands of developers already and widely understood.

The performance figures shows that the incremental version of our solution outperforms the batch mode execution of the same solution in all cases, often by multiple orders of magnitude. Compared to the incremental pattern matching tool VIATRA Query, we see that the performance is about as good

for most of the queries. Especially for medium-sized models, the revalidation times are better for all queries.

Another advantage of our solution is that it gives both a batch mode solution and an incremental solution our of the same pattern specifications. Thus, the same analysis code can be used in the case setting where incrementality is a clear advantage, or in a batch mode, e.g. when memory is a sparse resource or the analysis results are only required once.

Applying Incerator to the NMF solution of the benchmark, we were able to further improve the performance for heterogeneous change sequences. The evaluation shows that even for such simple analyses, further performance improvements of factor 2.2 are possible. In other cases, we were able to find configurations that are both faster and more memory efficient than pure instruction-level incrementality. However, the evaluation also shows that the choice of configurations is also sensitive to the size of the input model. Therefore, it is very important that developers provide example models of a realistic size.

The evaluation of the distributed computing abilities shows that the predicates are too simple for a distributed environment to be advantageous. Only in a purposely sub-optimal query, we ran into memory problems. However, the evaluation showed that in these cases, these memory problems can be solved using our approach for distributing the DDGs through a virtual actor implementation.

9.4. Case Study: Incremental Data Flow Transformations

In this case study, taken from the TTC 2016 Live Contest [69], the task was to create an interpreter for a model transformation language FlowM2M roughly inspired by block diagrams. Ideally, the interpreter should allow an incremental execution. An interpreter using Epsilon was provided. At the TTC, submissions using SIGMA [129], ATL [121], Mofongo[87] and NMF were submitted. After the TTC, solution authors were asked to revise their solutions,

[87] `http://mofongo.readthedocs.io/en/latest/`, retrieved 25 Jul 2017

supporting incremental execution where possible. The resulting paper was submitted to the ICMT 2017, but unfortunately rejected, partially because the NMF solution was the only one to support an incremental execution, despite some attempts for other tools.

9.4.1. Benchmark Setup

As the name shall suggest, the FlowM2M language resembles a data flow oriented model transformation language similar to block diagrams. An excerpt of the language primitives is depicted in Figure 9.12.

These primitives describe a model transformation as a data flow. In particular, each `Element` specifies how the interpreter should handle a given data row. This row may contain an arbitrary number of dynamically typed fields. An element may add fields, replicate the data row or perform side effects, depending on the type of element. For example, an Evaluate node computes an arbitrary expression based on fields saved in a data row and adds the computed expression as a new field. An AllInstances element replicates an incoming data row for every instance of a given type. The instance is saved as a field in each replica of the data row.

The exact semantics of all types of elements is not important in the scope of this thesis, but can be obtained from the live contest case description [69]. However, we want to describe elements that may perform side-effects here because they are important for the remainder of this section:

NewInstance: The purpose of this element is to create a new instance of a given type and add it to a field of the data row. However, this is based on a key: If an object has already been created for this key (e.g. by a prior NewInstance element), then the field should be set to this object rather than creating a new object. Therefore, this element implicitly accesses and builds up a trace.

AddToContainer: This elements adds a model element in a field of a data row to another element specified by a data row.

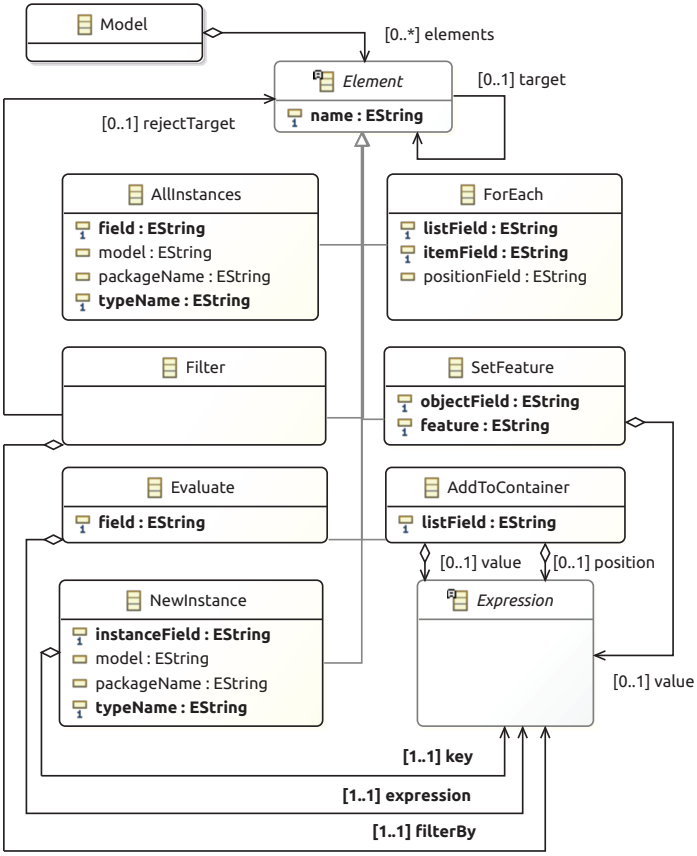
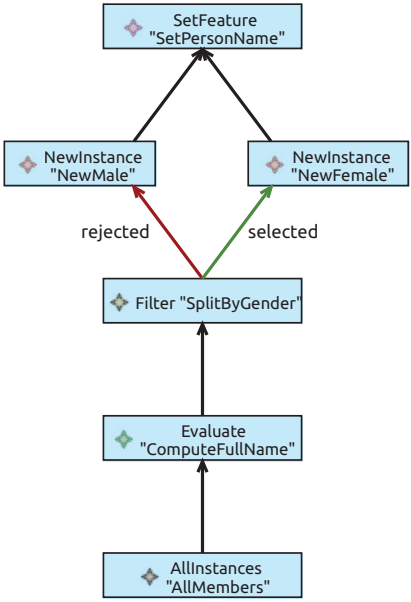


Figure 9.12.: Primitives of FlowM2M (simplified)

SetFeature: This element sets a feature of a model element saved in a field to a value stored in another field.

With these primitives, one can already describe model transformations such as the *Families2Persons* transformation from the ATL examples website[88] as depicted in Figure 9.13a.

[88] `http://www.eclipse.org/atl/atlTransformations`, retrieved 25 Jul 2017



(a) The *Families2Persons* transformation in FlowM2M

The task of the TTC live contest 2016 was simply to create an interpreter for this transformation language, i.e. a program that is able to apply transformation models to given input models. The authors of the live case especially asked for an incremental execution of the transformation specification.

9.4.2. Validation Goals

The goals of this case study are as follows:

Applicability The application area of this case study rather is unusual. However, the applicability of our approach to this case study shows its flexibility.

Performance In the context of this case study, performance is measured in terms of the time to process a model transformation together with its input

model. Further, we evaluate the incremental performance for some example model transformations to investigate for how many changes, an incremental change propagation is advantageous.

Relaxation of assumptions A large part of this thesis is built upon working hypothesis A1. This case study allows us to investigate what happens if this assumption is not fulfilled, which problems arise and how they can or cannot be managed.

9.4.3. NMF Solution

The (improved)[89] NMF solution is a compilation solution and consists of:

1. Transforming Ecore metamodels to NMeta.
2. Generating model representation code for the metamodels in C#.
3. Generating model transformation based on the FlowM2M language.
4. Compiling the generated code
5. Running the model transformation with the given models.

The results of steps 1–4 can be reused for subsequent runs of the transformation, like compiled Python files are reused by Python interpreters. The code generation is mainly necessary because NMF does not have an option to load metamodels dynamically.

To achieve incrementality, the NMF solution converts a dataflow into a mostly functional representation. This means, each dataflow node is understood as a function converting a sequence of input data rows into a sequence of output data rows. A dataflow node to compute the full name of a person is thus generated into the code depicted in Listing 9.9.

```
1 var computeFullNameFunc = ObservingFunc<...>.FromExpression(row =>
2   (((Families.Member)row["member"]).FirstName + "␣" +
3   ((Families.Family)((Families.Member)row["member"]).Parent).LastName));
4 var computeFullName = new DataflowNode(source =>
```

[89] Due to a misunderstanding of the semantics, the original NMF solution did not target incremental execution.

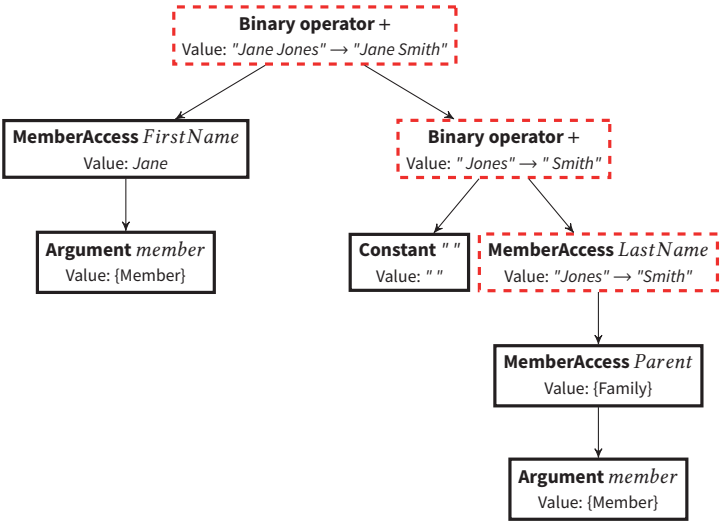


Figure 9.14.: The dynamic dependency graph template for the function in listing 9.9 applied to a member "Jane Jones" and the propagation of changes to her last name.

```
5 source.Select(row => row.With("fullName", computeFullNameFunc.Observe(row)))
      );
```

Listing 9.9: An evaluate node in the incremental NMF solution to compute the full name of a person

In this listing, the class `ObservingFunc` implements the incremental derivation of the given expression. This means, when passing an argument such as done in Line 4, the result will be an incremental value of string. For this, the system automatically registers event handlers when for instance the `FirstName` of a person changes.

In particular, NMF creates a dynamic dependency graph such as depicted in Figure 9.14. The nodes in this graph are specific to the model operations and register to elementary changes: The member access nodes register to the event that is fired when the respective member is changed. These changes are then propagated through the node.

The implementation of the `With` function in Line 5 contains an optimization that the incremental result is stored in the data row instead of the value itself.

As a consequence, the resulting output data row never changes for a given input data row but the current value of the `fullName` column may do so.

Problematic for the incrementalization are those dataflow nodes that cause side-effects, `SetFeature`, `AddToContainer` and `NewInstance`. The general approach of the NMF solution to handle these cases is to run the side-effect whenever the data rows of such dataflow nodes should be processed. This processing has to account for the fact that the input for these side-effects may change over time as for example `fullName` column of a data row may change because the `FirstName` of the respective person changed. Furthermore, it has to correctly handle the case that a row is withdrawn, for instance because it no longer passes a filter condition.

The side-effect introduced by `NewInstance` is an insertion of an element into a trace if no appropriate such element exists in there. This can be incrementalized by adding a counter to each element in the trace such that this counter can be decremented when a data row is removed for a dataflow node, for instance because a prior filter condition is no longer met. If the counter reaches zero, the element is removed from the trace.

Here, the incrementalization makes use of an important property of the trace entries created by a `NewInstance`-node: The order in which the system created trace entries does not matter. Performance-wise, this is a highly relevant insight as the position of a trace entry does not have to be tracked. Further, approaches for incremental computation that cope with side-effects in a generic way, such as Hammers self-adjusting stack machines [83], have to unroll the entire computation from the point a trace entry is revoked. A good understanding of the side-effect, though, allows to react to changes very fast. In case of a deletion, only the data rows affected by the deleted element are revoked while the transformation result for other elements stays intact.

The problematic node types are `AddToContainer` and `SetFeature`. For `AddToContainer`, the natural inverse operation seems to be to remove an element from its container. However, in NMF, removing an element from its container means to delete it and therefore to reset all references to this element. This behavior must be switched off which is currently not possible. For the `SetFeature` node, the problem is even more tremendous because it is unclear to what value a feature should be reset to when a data row should no longer be processed by a node as any previous value may be wrong. In

Tool	Language	Initial	Improved
Reference	EOL, Java	293 + 97	N/A
ATL	ATL, Java	104 + 30	100 + 123
Mofongo	Python	189	N/A
NMF	C#	682	1229
SIGMA	Scala	227	309

Table 9.3.: Complexity of the data flow transformation solutions, measured in lines of code

the implementation, we therefore do not reset a feature to any value if a data row is no longer processed by a `SetFeature` node.

An alternative implementation to reset the feature to the last value before the side-effect was applied may seem a better choice at the first glimpse but has severe problems. Consider the case where a `Filter` element is followed by a `SetFeature` such that the latter will invert the previous filter condition. In this case, the implementation to reset the feature to a value before the side-effect was applied will trap an incremental solution in an endless loop.

To produce correct results, the incremental NMF solution rests on an important assumption: Whenever a data row of a `SetFeature` or `AddToContainer` node is withdrawn, there is a `NewInstance` node that created the affected element for which the data row is also withdrawn. If this is the case, then the withdraw implementation in `AddToContainer` and `SetFeature` do not matter because the element will be discarded, anyways. Therefore, this assumption ensures the correct incrementalization of the side-effects.

9.4.4. Results

An overview of the lines of code for the different solutions is depicted in Table 9.3. From a conciseness point of view, the NMF solution was the least concise solution, particularly because operators that included side-effects had to implemented explicitly.

To evaluate the advantage of DDGs for incremental change propagation, we analyze the time to apply and propagate generated changes to the input model. We compare these times with the time necessary to recompute the output model from scratch, either by using the batch or incremental NMF

solution. In particular, for $n \in \{1, 10, 100, 1000\}$, we repeated the following process 20 times for each of the provided example models:

1. Run the batch solution for the unchanged model
2. Run the incremental solution from the unchanged model, measure the time, introduce n random changes, propagate the changes and serialize the changed source model,
3. Run the batch solution for the changed synthetic model.
4. Check that the resulting models are equivalent, up to the order of elements

The introduced random changes fall into one of the following categories:

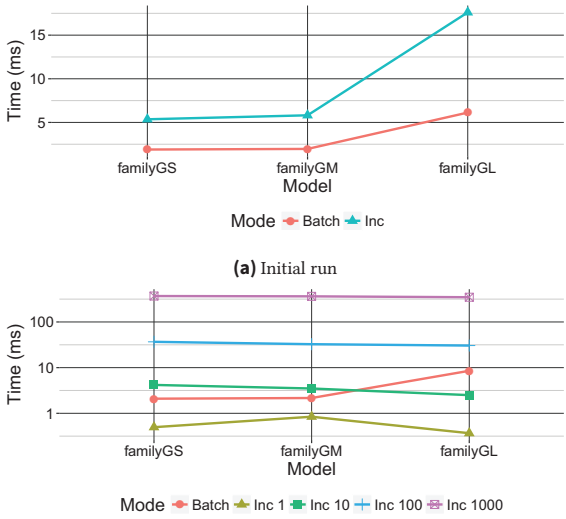
- Change the last name of a random family (20%)
- Change the first name of a random family member (40%)
- Remove a random family (5%)
- Add a new family, consisting of Homer, Marge, Bart, Lisa and Maggie Simpson (35%)

The results for the NMF solution for this benchmark are depicted in Figure 9.15. The plots show the time for the initial transformation and for propagating a set of changes (either by recomputing the entire transformation or by propagating the updates).

However, the provided example models were very small. The *familyGS* model only consists of 17 model elements, *familyGM* of 800 and *familyGL* of 15,000 model elements.

The times for the batch execution are depicted in Figure 9.15a. For all models, running the transformation in incremental mode and thereby creating the necessary DDGs for incremental change propagation is about three times slower than executing the transformation in batch mode.

The results for the incremental change propagation suggest that the dataflow transformation developed in this case study is already fully incremental – the runtime to propagate changes does depend on the size of the change (depicted as color in Figure 9.15b) rather than the size of the model. In the contrary, this does not hold for the batch strategy to recreate the transformation result



(a) Initial run

(b) Propagate updates

Figure 9.15.: Performance results for the NMF solution of the TTC 2016 Live Contest

after all changes have been applied: Recomputing the largest example model *familyGL* takes much longer than recreating the smaller models.

Besides these effects for large models, the results also show that in cases that the target model only has to be consistent with the source model rarely, recreating the target model from scratch may be faster. This is because for the smaller models, very large parts of the model are changed, in case of the smallest model *familyGS* more model elements are introduced than the original model had before the changes.

9.4.5. Summary

The case study shows, that the incrementalization system NMF Expressions is very flexible in the sense that it can be beneficial to use it also in contexts that violate basic assumptions such as the referential transparency assumption A1. However, the usage of NMF Expressions in such a scenario may require extra work to support the side-effects that the analysis has to perform.

In the scope of this case study, the side effects span the creation of an entire target model, meanwhile the actual result, the final set of data flows, is rather meaningless.

Other incremental solutions such as the AOF [20] or the incremental version of ATL implemented in ReactiveATL [122] were apparently not as flexible. At least, the author of ATL, Frederic Jouault, did not manage to come up with an incremental interpreter of the data flow language. For other incremental tools such as primarily VIATRA, there was no attempt for an incremental solution of the data flow case, so we cannot compare in this case study.

The case study also demonstrated the problems attached to side-effects in the context of incremental execution: It is not clear how a side-effect should be undone such that unintended consequences can be avoided, even if the side-effects are well-known. The incrementalization system simply cannot distinguish between changes caused by the user and changes caused by reverting a side-effect and thus, it is easy to trap it in an infinite loop.

The performance results obtained for a sequence of changes in the Families to Persons example regarding performance shows that the incremental change propagation of a single change is faster than rerunning the transformation by more than an order of magnitude for the selected example input.

9.5. Case Study: Incremental Model Transformation of Petri Nets to State Charts

In this section, we explore the performance gains that can be achieved using the incrementalization system integrated into the incremental model transformation approach presented in Chapter 6.

9.5.1. Benchmark Setup

To analyze the response time from elementary changes in the finite state machine to the updated Petri Net, we designed a benchmark where we generate a sequence of 20 elementary model changes to the finite state machine. After each model change, we ensure that the Petri Net is changed accordingly, either by performing change propagation or by regenerating

the net fresh from scratch. To take the different sizes of finite state machines into account, we performed our experiment for different sizes (10, 20, 50, 100, 200, 500, 1,000, 2,000, 5,000, 10,000, 20,000 and 50,000 states). The generated workload on these finite state machines shall reflect edit operations as done by a user. In particular, we generate the following elementary changes (percentage on the overall change workload in brackets):

- Add a state to the finite state machine (30%)
- Add a transition to the finite state machine with random start and end state (30%)
- Remove a random state and all of its incoming and outgoing transitions (10%)
- Remove a random transition from the finite state machine (10%)
- Toggle end state of a random state (5%)
- Change the target state of a randomly selected transition to a random other state (5%)
- Rename a state (9%)
- Rename the finite state machine (1%)

The evaluation works as follows: For every run of our benchmark, we generate a finite state machine of a given size n representing the number of states. We then generate a sequence of 20 elementary model changes acting on randomly selected model elements of the finite state machine. For each of these actions, the action itself must be performed and the Petri Net must be updated or newly created appropriately.

We compare four implementations of this task. The first solution is using NMF Synchronizations in batch mode, i.e. the synchronization is run as a transformation from its left side to its right side with change propagation switched off. Next, we use the same synchronization code without any modification and use it in incremental mode, i.e. from left to right with change propagation mode switched on to OneWay. Third, we use an implementation for this transformation task in NTL, basically taken from previous work [92]. This solution works similar to the batch mode version but lacks some of the overhead implied by the NMF Synchronizations implementation. NMF Transformations used with NTL showed good performance results compared with other (batch mode) model transformation languages at the TTC 2013

[103, 102] so we think it is a fair comparison. Lastly, we compare the NMF solutions with a solution in eMoflon, which has been discussed in Section 6.4.

We did two runs of the experiment. In the first run, we check the generated Petri Net after each workload item in order to test the correctness of NMF Synchronizations. Here, we basically assume the implementation in NTL correct. In a second run of the experiment, we evaluated the execution time to apply all the elementary model changes in sequence and updating the Petri Net accordingly after each change (either by rerunning the transformation or by propagating changes). The application of 20 elementary model changes and updating the Petri Net is still a matter of milliseconds but this way the precision gets in a reasonable scale.

To compare with eMoflon, we transformed the generated changes into delta specifications that can be understood by eMoflon. We then start a Java process running the eMoflon solution for the initial transformation. Afterwards, we subsequently load the delta specifications one after another and integrate them to the target model. Here, the time for transforming the changes into the delta specifications, serializing them in the benchmark driver and deserializing them in the eMoflon process is not taken into account. Rather, we only measure the time for the integration. However, this also means that we sum up 20 time measurements which means that these results are not as accurate.

9.5.2. Validation Goals

The goals of this case study are as follows:

Applicability The application domain of this case study is rather synthetic. However, we believe the synchronization of structurally similar yet different metamodels is an important use case. A remarkable feature of the transformation is that the existence of model elements in the RHS depends on an attribute value in the LHS.

Performance Also in this case study, we seek to compare the performance of incremental model transformations in NMF Synchronizations with their batch equivalent, but also with traditional non-incremental transformations such as a transformation written in NTL. However, we also implemented the model transformation of this case study in the incremental and bidirectional model transformation tool eMoflon to compare the performance to this tool.

Correctness We implemented a custom method to check the correctness of the model transformation by performing an in-depth comparison of the transformed Petri nets from the NMF Synchronizations implementation in batch or in incremental mode.

9.5.3. Results

Figures 9.16 and 9.17 show the performance results. The code for our used benchmark is available as open source on Github[90] so that the interested reader can obtain results for any other machines as well. In particular, Figure 9.16 shows the results for the initial transformation, Figure 9.17 shows the time to run the 20 generated changes.

For the initial run of the transformation, we can see that NTL and the batch mode of NMF Synchronizations are much faster than the incremental NMF Synchronizations or the eMoflon solution. The NMF Synchronizations implementation running in batch mode is about as fast as the unidirectional implementation using NTL while the incremental mode adds a slight overhead. This is because the engine has to create dependency graphs for all lenses but as these lenses are rather simple, this overhead is not very large. eMoflon also has to create data structures to perform incremental change propagation and is even slower than the incremental mode of NMF Synchronizations.

For incremental change sequences, the results indicate that even for very small models such as a finite state machine with just 20 states, it is already beneficial to use the change propagation built into NMF Synchronizations: Recreating the Petri net after every change takes more than 18 times as long

[90] http://github.com/NMFCode/SynchronizationsBenchmark/

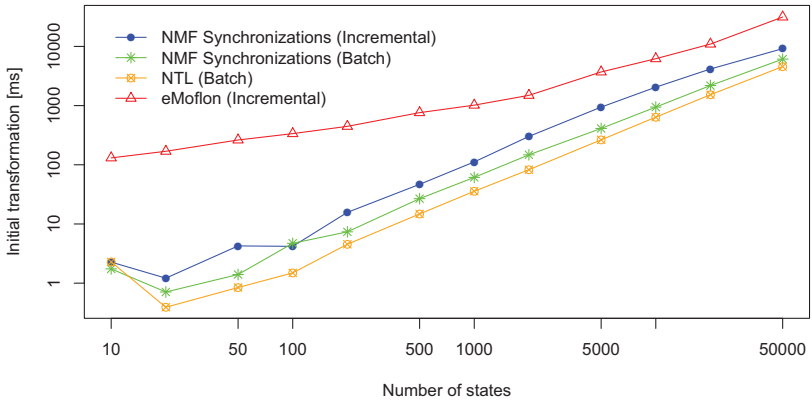


Figure 9.16.: Performance results for the initial transformation

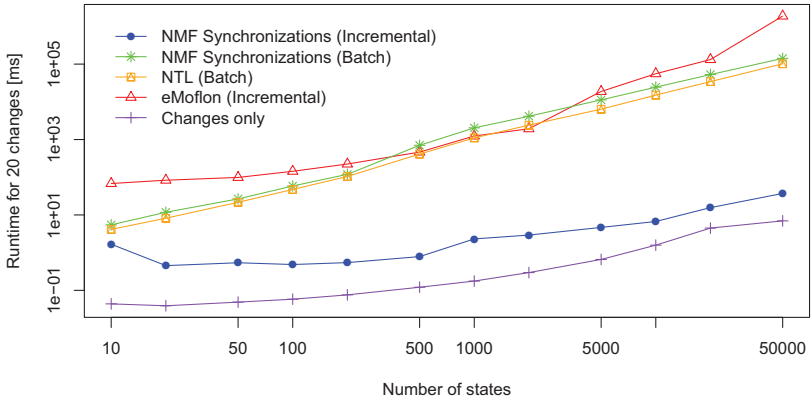


Figure 9.17.: Performance results running 20 randomly generated changes

as propagating the change. Compared to NMF Synchronizations in batch mode, the factor is even at 25. For larger models, the speedup gets higher and for the largest models with 50,000 states, the change propagation is factor 2,750 faster than recreating the Petri net using NTL or factor 3,800 faster than NMF Synchronizations in batch mode. This is because the time to propagate a change merely depends on the size of the change, rather than

the overall model size: The curve for incremental change propagation in blue is almost flat in a logarithmic plot.

However, there are some operations such as the model change operations themselves, that have a linear complexity[91], which is why the speedup does not grow linearly with the size of the model. As a consequence, the distance between the blue and the purple curve in Figure 9.17 becomes smaller the change propagation overhead becomes smaller in relation to actual model changes: While for 20 states, the actual model changes only take about 8% of the time for incremental change propagation, this share is at 28% for 20,000 states. Given that the incremental change propagation includes a roughly similar model manipulation in the RHS, this means that the overhead for change propagation becomes very small.

Comparing the results with eMoflon, we can see that eMoflon is slower than the incremental NMF Synchronizations implementation and for most model sizes also slower than the implementations in NTL or in batch mode. The eMoflon solution also does not scale well with an increasing model size: While the curve for the incremental NMF Synchronizations solution is almost flat for up to 2,000 states, indicating a fully incremental solution, the curve for eMoflon is steeper already for smaller model sizes, indicating a worse scalability.

However, the performance of the synchronization depends on many factors. Indeed, even in the very small example of the synchronization between finite state machines and Petri nets, some types of changes such as name changes are much faster to propagate than others such as adding or removing states. As a reason, the propagation for the latter includes the execution of a transformation rule meanwhile the propagation of a name change simply means to copy the new name over to the target model.

Nevertheless, the evaluation shows that the overhead for change propagation stays approximately at a constant level, indicating that NMF Synchronizations is fully incremental in the terminology of Giese and Wagner [74]. Especially for large models, the overhead is small. We see this as a consequence of the fact that changes to the RHS can be directly constructed from the model changes: The definition of the repair operators in Definitions 39 and 40 are straight and explicit.

[91] For example, deleting an element from an ordered list is a $O(n)$ operation.

9.5.4. Summary

The main objective of this case study was to obtain insights to the performance characteristics of NMF Synchronizations. Besides a comparison of the incremental version batch execution, we were particularly interested in a comparison of performance properties with other incremental model transformation tools, especially eMoflon after a comparison with regard to syntax and semantics has been already performed in Section 6.4.

With regard to the applicability of NMF Synchronizations, the case of state machines to Petri Nets also shows that NMF Synchronizations is able to handle a case that has limited support in TGG implementations: For example the considered approach eMoflon has no support to react on attribute changes.

The performance results indicate that the presented has a significantly better performance than recreating the target model from scratch. For large input models, propagating the changes is faster than recreating the target models by multiple orders of magnitude. The results also suggest that NMF Synchronizations is significantly faster than eMoflon in this case. For large models, the overhead to propagate changes to a target model even decreases. For large models with 60,000 model elements[92], performing the actual model change took 28% of the total time to perform the change and propagate it.

9.6. Case Study: Incremental ATL transformations

Model transformations are not always bidirectional. In particular, model transformations often contain some degree of information loss, e.g. some details in the source model may not be represented in the target model. Therefore, in such a case, a backward transformation is often very difficult but may not be required. In fact, ATL, the probably most popular [203] textual model transformation language in the community, by default only supports unidirectional, by default non-incremental model transformations.

[92] 20,000 states plus 40,000 transitions

Synchronization blocks do support unidirectional model transformations beyond just forgetting the backward transformation. However, the expressiveness of unidirectional synchronization blocks and also their performance compared to popular model transformation languages is unclear. Because unidirectional synchronization blocks are very similar to regular bidirectional synchronization blocks, one may be tempted to guess that the expressiveness of unidirectional synchronization blocks is roughly the same as for bidirectional synchronization blocks.

To demonstrate the applicability of synchronization blocks but also to be able to compare the performance of their implementation in NMF Synchronizations, we created a Higher-order Transformation (HOT) that maps model transformations in ATL to an incremental model synchronization specified in NMF Synchronizations.

The transformation is based on a master thesis by Nicolas Pätzold [161] to which the interested reader is referred for further details.

9.6.1. Validation Goals

With this case study, we follow the following validation goals:

Applicability Whereas on a first sight, it may seem that synchronization blocks are only expressive enough to cover the transformation of almost identical structures, this case study aims to show that it is expressive enough to cover many practical model transformations. In particular, we want to show that the formalism is expressive enough to cover the declarative part of ATL.

Performance The performance of the actual HOT is not of interest for this case study, but we are rather interested in the performance of the generated model synchronizations. As this performance depends on the exact transformation, the input model and a change sequence (in case of an incremental execution), we use the performance for some example model transformations, example models and example change sequences. In particular, we are interested to see whether the generated incremental model transformations are faster than rerunning the original ATL transformations after each model change.

Correctness We compare the resulting models for a series of example model transformations and instances taken from the ATL website.

9.6.2. Transforming ATL to Synchronization Blocks

In this section, we will describe in detail how ATL transformations are transformed in our approach called ATL2NMFs. For this, we describe how rules, bindings, helpers, OCL expressions and variables are transformed, how the matching phase is emulated and how multiple input or output pattern elements are processed.

A prototypical implementation of this transformation is publicly available online[93].

9.6.2.1. Rules

There are three different kinds of declarative ATL rules: *matched* rules, *lazy* rules, and *unique lazy* rules. Listing 9.10 shows an example of a matched rule (for simplicity without any bindings). The rule takes source model elements of the type `Element`, defined in the source metamodel `XML`, and transforms them into target model elements of the type `Book`, defined in the target metamodel `Book`. This happens only under the condition that the name of the element is `'book'`.

Lazy rules and unique lazy rules are defined similarly but use the additional keywords `lazy`, or `unique lazy` respectively. Each rule has one or multiple source pattern elements defined after the `from` keyword. For each source pattern element, the source type of the element, as well as its source metamodel, in which the type can be found, are declared. Each rule has also at least one target pattern element that is declared right after the `to` keyword. Every target pattern element specifies the type and the metamodel of the target model element that is created for the source model element for which the rule is called. Additionally, by declaring one or more bindings, it is possible to specify how the created target model element is initialized.

[93] `https://github.com/NMFCode/ATL2NMFS`

```
rule BookRule {
   from
      e : XML!Element ( e.name = 'book')
   to
      b : Book!Book (...)
}
```

Listing 9.10: ATL matched rule without bindings

The difference between the three kinds of declarative ATL rules lies in their execution semantics. While NMF Synchronizations requires the transformation developer to explicitly call any rule, ATL has a matching phase in which matched rules participate. Thus, they are called automatically for each suitable source model element, whereas the lazy and unique lazy rules have to be explicitly called from a binding for any source model element that they should transform.

ATL2NMFs maps all three kinds to synchronization rules that are created equally, because the difference of the rule type only has an impact on how the rules need to be called.

The name of the rule can be used as the name of the synchronization rule, implemented as a public nested class of the synchronization. As such, the generated rule needs type parameters for the source and target model element's type. First, we assume only one source and target element, other cases are described in Sections 9.6.2.8 and 9.6.2.7. The result of the mapping of the matched rule from Listing 9.10 can be seen in Listing 9.11 below. Since no bindings are shown in the ATL rule, the corresponding synchronization blocks (which are defined in the `DeclareSynchronization` method) are not shown either.

```
public class BookRule : SynchronizationRule<XML.IElement, Book.IBook> {
   public override void DeclareSynchronization() { ... }
}
```

Listing 9.11: NMF Synchronizations rule without synchronization blocks

A lazy rule creates a new target model element each time it is executed, even if the rule is called multiple times for the same source model element. Instead, a unique lazy rule creates the target model element only once for each source model element, even if the rule is called multiple times for the same source

model element. The execution behavior of an ATL unique lazy rule conforms to the default behavior of a NMF Synchronizations rule. The behavior of non-unique lazy rules can be enabled in NMF Synchronizations with a simple switch.

ATL rules may have guards attached to the rules. In NMF Synchronizations, guards are defined as filters whenever the rule is called. The developer has the freedom to decide when and if the filter should be applied for a source model element. It is even possible that different filters are used for the same rule.

In ATL2NMFs, ATL guards are transformed similarly to helpers. Each filter is mapped to a C# extension method that is defined for the input type of the particular ATL rule. The extension method is called whenever the specific rule is called. That way, code duplication is avoided and the code readability remains high. The filter condition, which is expressed by a boolean OCL expression, is mapped to a C# expression (cf. Section 9.6.2.5).

Rule inheritance of ATL can be mapped to rule instantiation of synchronization rules in NMF Synchronizations directly. However, there is a limitation because the current engine of NMF Synchronizations still does not support if the filter condition actually changes.

9.6.2.2. Emulation of the ATL Matching phase

The ATL matching phase is emulated by generating a main synchronization rule that also serves as an entry point for the synchronization.

Each synchronization in NMF Synchronizations needs a synchronization rule which serves as an entry point for the synchronization process. This synchronization rule usually declares the synchronization between the root element at the source model and the root element at the target model. In ATL, such a root rule often does not exist. Instead, the model elements in the source model are matched in the ATL matching phase.

The main rule is defined as synchronizing the input models with the output models, all typed as XMI containers.

To emulate the matching, ATL2NMFs creates various synchronization blocks for the main synchronization rule. For each matched rule, a synchronization

block is generated to call the rule for any elements contained somewhere in the source model(s).

A problem arises when a rule is called for a source model element from a synchronization block of the main rule, if this rule has already been executed for this particular source model element from another synchronization block. Because a model element can only have at most one parent model element, it would be removed from the correct position in the target model and again added as top-level element.

The problem can be solved by using a special collection proxy used in every synchronization block of the main synchronization rule. As before, the elements stored in the list represent the top-level elements of the target model. However, elements are only added to the list if they do not have a parent model element assigned, i.e. are not contained anywhere else in the target model. If the element already has a parent element, then this means that it is already contained in the target model and must not be moved.

None of the lazy or unique lazy rules have to be called from one of the synchronization blocks of the main synchronization rule, because these rules should only be called from a synchronization block where the corresponding binding calls the specific rule directly.

9.6.2.3. Bindings

A major difference between the two languages lies in the way rules are connected with each other, and how rule calls to transform model elements are specified. In an ATL transformation, dependencies between rules are specified implicitly – bindings are resolved using the traceability links created in the model element matching phase, or a suitable lazy rule is called. The decision which rule has to be called is not specified explicitly in the binding.

In a NMF synchronization, dependencies have to be declared explicitly in a synchronization block. Each binding of an ATL rule has to be mapped to synchronization blocks where the possible synchronization rules that have to be called to transform the model element must be declared explicitly. Possibly, multiple synchronization rules match a binding and therefore multiple synchronization blocks have to be generated, each with a respective filter condition. This is necessary, since only one synchronization rule can be called from a synchronization block.

```
b : Book!Book ( ...
  chapters <- e.childrenXml->select(c | c.oclIsKindOf(XML!Element))->
        asSequence()
)
```

Listing 9.12: An exemplary ATL binding

```
public override DeclareSynchronization() {
  SynchronizeManyLeftToRightOnly(SyncRule<ChapterRule>(),
    e => e.ChildrenXml.OfType<XML.IElement>().Where(x => x.ChapterRuleFilter()
         ),
    b => b.Chapters);
}
```

Listing 9.13: The synchronization block generated for the binding in Listing 9.12

To generate the synchronization block for a binding, we first analyze the type of the expression that should be bound and compare it with the type of the target pattern element member to which the expression should be bound. If these types match or can be easily converted (through a built-in type conversion such as from integers to floating-points), we use the identity as target isomorphism for the generated synchronization block. Otherwise, the synchronization rule for the applicable ATL rule is selected. In case there are multiple applicable rules, a synchronization block is created for each of them with an appropriate filter when this rule applies.

Furthermore, we require information on the multiplicity of the binding, i.e., whether a single-valued or multi-valued synchronization block has to be created. This information can be easily extracted from the binding target.

As an example, consider the binding of chapters depicted in Listing 9.12. The binding completes Listing 9.10 from above.

ATL2NMFs statically resolves the rule(s) applicable for the given binding and generates the synchronization block(s) depicted in Listing 9.13 in the generated synchronization rule `BookRule`. In this case, only one synchronization block is generated. Due to filters, there could be multiple synchronization rules applicable.

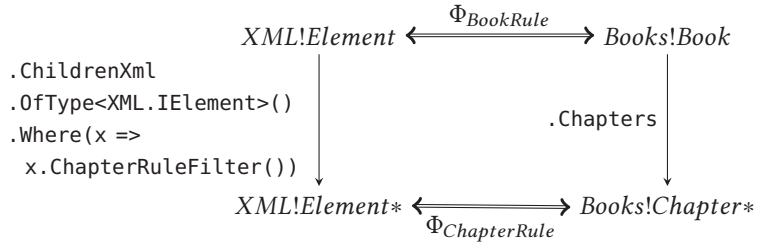


Figure 9.18.: Synchronization block from Listing 9.13

In Listing 9.13, Line 3 defines a predicate in the source model element to obtain the model elements to transform. This is a translation of the expression that should be bound, i.e. the right part of Line 2 in Listing 9.12. We explain details of this transformation in Section 9.6.2.5. Line 4 defines the feature of the target model where the transformed elements should be stored, i.e. the member that should be bound. In Listing 9.12, this corresponds to the left part of Line 2.

A graphical representation of this synchronization block is depicted in Figure 9.18. If the chapter rule filter was invertible, NMF would be able to invert the selector and thus, the synchronization block could be executed bidirectional[94]. However, currently ATL2NMFs does not perform this check and always generates an unidirectional synchronization block, as specified in Line 2 of Listing 9.13.

9.6.2.4. Helpers

Helpers can be used in any OCL expression in the ATL transformation, even in the body expression of another helper. In ATL, one can define two different kinds of helpers: functional and attribute helpers[95]. Each helper is defined with a name, a return value type, a body (which is an OCL expression), an optional context, and parameters, which have to be passed to the helper when called. Listing 9.14 shows the definition of a simple helper.

[94] A filter condition such `e.name = 'chapter'` is currently thought of as not invertible, because it is unclear how it should be set to `false`.

[95] Also known as operation and property helpers

```
1 helper context FSM!State def :
2   nameOfState() : String = self.name;
```

Listing 9.14: Definition of a simple ATL functional helper

For this helper, ATL2NMFs generates a C# extension method that is specified for the type of the context of the particular helper. When the extension method is called on an element, this element is automatically passed as a parameter to the extension method, similar to ATL. To further simplify the mapping process, the variable of this parameter has the name `self`, used in ATL as keyword to reference the context object. The return value of the extension method is a type of the source metamodel or an OCL type. In case of an OCL type, the corresponding C# built-in types, such as `int`, `bool`, or `string` are used. Also more complex OCL types such as maps, sets, sequences and ordered sets are supported.

Because NMF Expressions that is used in NMF Synchronizations is not able to look inside a compiled method, we also need to create a proxy method to properly incrementalize the helper. For the implementation of the proxy method, we may use the abilities of NMF Expressions to create a dynamic dependency graph template which just needs to be instantiated for the given input. The replication of the code is only required such that the non-incremental version does not have to load the dependency graph template.

The proxy method is referenced using a type and the method name. The generated type is used for the method proxies of all helper functions and is not visible from outside the class where the extension method is defined.

The result of the mapping process for the ATL helper from Listing 9.14 is shown in Listing 9.15.

To hide the implementation detail that we annotate the generated methods with proxies such that they can be used in an incremental setting, we generate all proxy methods into a private class `Proxies` that is shared for all helpers. The purpose of this class simply is to make the proxy methods invisible for developers.

```
[ObservableProxy(typeof(Proxies), "NameOfState")]
public static string NameOfState(this FSM.State self) {
   return self.Name;
}
private class Proxies {
    private static ObservingFunc<FSM.State, string> nameOfStateFunc =
      new ObservingFunc<FSM.State, string>(self => self.Name);
    public static INotifyValue<string> NameOfState(INotifyValue<FSM.State>
         self) {
       return nameOfStateFunc.Observe(self);
    }
}
```

Listing 9.15: C# version of the simple ATL helper of Listing 9.14 with attribute and proxy

9.6.2.5. ATL-OCL

ATL uses a tailored version of OCL expressions in many different places: as body expression of a helper, as filter condition of a rule or in a binding of a target pattern element. Therefore, the analysis and mapping of OCL expressions is an important topic in the concept of the ATL2NMFs HOT. The counterpart of OCL expressions which is used in NMF Synchronizations to define expressions are the SQOs.

Even though the ATL language is based on the OMG OCL standard, its implementation of OCL is different in a few points in comparison to the standard OCL definition [16]. Therefore, every time we mention OCL in this section, we refer to the OCL implementation of the ATL language.

We transform OCL expressions based on their abstract syntax tree representation. Each OCL expression element is transformed to a corresponding expression in C#, partially using SQOs:

Types: The different primitive types of the OCL language, such as `String`, `Integer`, `Boolean` can be represented by the corresponding built-in types of the C# language. The model types, which are specified in one of the used metamodels, are mapped to C# classes by the *EcoreInterop* tool of NMF.

Primitive Expressions: References to numeric literals or arithmetic expressions can be mapped to equivalent C# operators.

Conditional Expressions: Conditional expressions in OCL can be mapped to the C# ternary operator.

Model Navigation or Attribute Helper Expressions: Model Navigation Expressions, such as references to attributes or references, are converted to property expressions to the generated property of the respective feature. However, OCL also allows access to properties which are not defined in the static type of the source element, but rather in one of its subtypes. Such an access would not be type-safe and is thus not allowed in the type-safe C# language. Therefore, it is necessary that `ATL2NMFs` adds a type cast which casts the element to the correct subtype before the property access is made. To prevent runtime errors, a check is added to verify if the cast is valid before the property is accessed. Otherwise, `null` as the equivalent of `OclUndefined` is returned. An expression that calls a helper is converted to a method call to the respective helper function.

Operation Expressions: OCL defines several different operations that can be used. Most of them have a semantic equivalent in C#, Akehurst and others even state that the more recent versions of C# make OCL redundant [5]. A short list of example operations and their mapping by `ATL2NMFs` is depicted in Table 9.4. The *source* and *arguments* keywords in the table are placeholders for the specific OCL expressions, respectively the mapped versions of them. Depending on the operation, the *arguments* expression can either be empty, consist of one argument expression, or consist of multiple argument expressions.

Few operations, such as operations based on collection indices (`at`, `indexOf`, `last`), are not supported because the underlying NMF Expressions framework currently does not support indices in an incremental setting. The well-known `iterate` expression is not supported because the operation that is applied to the accumulator is not invertible in general. As a consequence, the engine would have to recompute the iterate statement for the entire underlying collection, which is slow and therefore not supported.

OCL Operation	C# expression
source.oclIsKindOf(*arguments*)	*source* is *arguments*
source.oclIsUndefined()	*source* == null
source.implies(*arguments*)	!(*source*) || (*arguments*)
source.sum()	*source*.Sum()
source.first()	*source*.FirstOrDefault()
source.includes(*arguments*)	*source*.Contains(*arguments*)
source.flatten()	*source*.SelectMany(x => x)
source.union(*arguments*)	*source*.Union(*arguments*)
source.select(*body*)	*source*.Where(*body*)
source.collect(*body*)	*source*.Select(*body*)
source.exists(*body*)	*source*.Any(*body*)
source.forAll(*body*)	*source*.All(*body*)
source.any(*body*)	*source*.FirstOrDefault(*body*)
source.sortedBy(*body*)	*source*.OrderBy(*body*)

Table 9.4.: Example ATL-OCL operations and their mapping to the corresponding C# expressions

An operation call expression may also be used to call a functional helper. In this case, ATL2NMFs generates a call to the extension method generated for the helper.

OCL Collections: OCL defines the collection types `Bag`, `Set`, `OrderedSet`, `Sequence` and `Map`. Besides `Bag`, all of these collection types have a corresponding class in NMF. Furthermore, C# defines a syntax to initialize a collection with members, an example is depicted in Listing 9.16. This syntax is supported by NMF Expressions and therefore available in NMF Synchronizations.

```
1 var orderedSet = new ObservableOrderedSet<int>() { 0, 8, 15 };
```

Listing 9.16: Collection initialization in C#

However, our tool ATL2NMFs does not have full support for collection initializations, yet.

```
1 rule BookRule {
2    from
3       e : XML!Element ( e.name = 'book')
4    using
5       name : e.name
6    to
7       b : Book!Book (...name...)
8 }
```

Listing 9.17: ATL matched rule with a variable

9.6.2.6. Variables

ATL rules may define variables to simplify the specification of bindings. An example of such a variable is given in Listing 9.17.

We suggest to simply expand the variables to their full definition in the binding. This works provided that the variable initialization is deterministic. However, there is currently no further support for variables in synchronization rules.

9.6.2.7. Multiple Output Pattern Elements

It is possible that an ATL rule declares multiple target pattern elements. Each target pattern element creates one target model element for each rule execution. This behavior cannot be easily achieved with one NMF Synchronizations synchronization rule.

Thus, ATL2NMFs creates separate synchronization rules for each target pattern element of an ATL rule. Each synchronization rule is declared with the stated source type of the source pattern element of the ATL rule, where the target type of the synchronization rule is declared with the type of the particular target pattern element for which it was created.

The first declared target pattern element of an ATL rule has the special role of being the default target pattern element. Whenever a source model element is referenced in a binding, the rule created for the default target pattern element is used to for the binding (cf. Section 9.6.2.3).

An ATL developer also has the possibility to reference a specific target pattern element which should be used to initialize a binding (using the `resolveTemp` function). In such a case, the target model element created by the referenced target pattern element is used in the binding instead of the default target pattern element rule.

To make sure that all rules created from an ATL rule are called, we replicate the calls of the default rule. We omit this step for every output pattern element that is referenced in a binding. If the rule is a matched rule, this means that we replicate the call of this rule in the emulated matching phase.

In case other target pattern elements are used in a binding, we create a synchronization block that uses the identity as left selector and the synchronization rule created for that target pattern element as child synchronization rule.

9.6.2.8. Multiple Input Pattern Elements

ATL rules may have multiple input pattern elements. In a called rule, the arguments are simply passed into the rule. In a matched rule, the rule guard is applied to the Cartesian products of the input model elements.

In many cases, the Cartesian product is inappropriate because the filter condition filters out most of the tuples. Therefore, it appears that other transformation languages such as SimpleGT are more suitable in such a case [211]. Thus, we think that this feature of the ATL language is not commonly used and we did not give particular attention to it.

Hence, the support for multiple input pattern elements is not fully implemented in `ATL2NMFs`, but the transformation is only sketched here.

In NMF Synchronizations, the support for multiple input pattern elements is also rather limited. As a reason, we experienced with NTL [92] that multiple input elements is a rare case, but required a tremendous amount of code to support it. At the same time, the advantages of a true support for multiple input elements over transformation of tuples is limited.

Therefore, the easiest way to support multiple input pattern elements in NMF Synchronizations is to simply use tuples as inputs. Then, the model matching has to be adapted to match tuples instead of elements. Compared to a dedicated support of multiple input elements, this has the drawback that

rule instantiation does not work because tuples are no covariant with the tuple types.

While a strict 1:1 mapping to a filter of the Cartesian product is possible, it is not desirable for performance reasons. NMF Expressions currently does not optimize queries for incremental execution and therefore, the generated transformation would also create a Cartesian product and create a filter. In an incremental setting, every tuple of the Cartesian product had to be kept in memory at all time, which would be very costly in terms of memory.

Using a manually optimized query based on model navigation or joins to find model element tuples in the source model, also a much faster pattern matching is possible than in ATL. For the example mentioned above where SimpleGT was used, the appropriate NMF expressions for the patterns are discussed in [105].

Implementing multiple input model elements similar to the implementation of multiple output elements, i.e. to create multiple synchronization rules, is not a good option. One the one hand, this is because usually multiple input elements are used in each output element (but not vice versa) and more importantly, it is much more difficult to get the source element for a given target than the other way round. This is not because of the lacking information, but rather because it inclines a change of the the transformation direction. This cannot be done in NMF Synchronizations without losing the advantages of declarative incrementality.

9.6.3. Limitations

ATL2NMFs is not designed to support complete ATL. In the following enumeration, a rough overview of the unsupported ATL concepts, or constructs respectively, is given with the reason why they are not supported:

Imperative constructs: Only declarative constructs are supported. Every single imperative construct of the ATL language, such as called rules or imperative code blocks, are excluded. An incremental execution of imperative code blocks is not supported in NMF. Although there are some approaches to incrementally execute imperative code [83], these approaches are not implemented in NMF Expressions yet and are therefore not available in NMF Synchronizations.

Refining mode: The supported subset of the ATL language is restricted further by excluding the refining mode. The main problem with the refining mode in the context of an incremental execution is that it is difficult to differentiate the changes performed in the course of the transformation from changes that the transformation needs to react to.

Iterative Target Pattern Elements: A noteworthy, but not critical limitation is, that iterative target pattern elements, which are represented by the `distinct foreach` keyword, are not supported. Their behavior, respectively their usage will not be described here, because they are deprecated since ATL version 2.0 [16]. It is recommended to use unique lazy rules instead, which are supported by the Atl2NmfS HOT. However, it is noteworthy, since iterative target pattern elements are still used in many example transformations of the ATL zoo.

Nodefault Rules: The ATL language also offers some constructs, which are very subtle, to influence the behavior of the ATL engine. Some of their repercussions must first be fully analyzed to be able to give a final assessment about the complexity of their mapping. An example for such a construct is e.g. the `nodefault` keyword. A matched rule can be marked with this keyword to influence the ATL matching phase. Normally, a transformation results in an error, if a source model element is matched for multiple possible rules in the ATL matching phase. By using the `nodefault` keyword, it is possible to allow such a situation. The ATL engine is in a position to choose one of the possible rules based on the `nodefault` keyword to transform the specific element. The mapping of this keyword must be considered in the created main synchronization rule, where the ATL matching phase is simulated. However, the mapping is not that simple, since the information can not be determined statically and thus is only known during runtime if a specific source model element matches multiple rules. It must be ensured, that the specific source model element is only synchronized by one synchronization rule, which was also chosen by the ATL engine.

Queries: Queries are not of any interest for this case study since they cannot be invoked from an ATL transformation and thus are invoked completely separate from the actual ATL transformation.

Variables: As mentioned above, we assume that variables are initialized deterministically.

Bidirectional Transformation: Synchronization blocks theoretically support bidirectional transformations. However, a synchronization created by the ATL2NMFs HOT only uses one-way synchronization blocks, which is why the backward transformation cannot repair any inconsistencies by changing the source model. In the future, we plan to check whether selectors can be inverted and use bidirectional synchronization blocks in that case.

Custom Isomorphisms: NMF Synchronizations allows users to override the rules that determine when existing target model elements should be reused to create new correspondences. For this, we require the transformation developer to specify when such a correspondence should be established, for example based on names or element IDs. We see no way to extract this information from an ATL specification, and thus, the generated transformation is not able to operate on existing target model elements. However, this specification can be added by the transformation developer, even separate from the generated code.

Notably, existing offline model synchronization engines for ATL such as SyncATL also need to decide when two elements are identical. They solved this problem by adding a requirement that all metamodels used with SyncATL have to have a mandatory identifier attribute for all classes of the metamodel(s).

9.6.4. Results

To test the correctness of the presented approach, ATL2NMFs has been applied to 21 example model transformations from the ATL examples website[96] or adapted to cover more ATL language features. The generated model transformations in NMF Synchronizations have been executed with available example models and the resulting target models were compared to the target models generated using the original ATL transformation. The comparisons

[96] http://www.eclipse.org/atl/atlTransformations, retrieved 15 Feb 2017

have been performed manually as diff tools such as EMF Compare produced phantom differences.

The chosen example transformations are *Families2Persons*, *XML2DXF*, *XML2-Book*, *Make2Ant*, *PetriNet2PathExp*, *PetriNet2Grafcet*, *PetriNet2PNML*, *PortV2*, *PortV3* and *PortV4*. These transformations cover matched rules, lazy rules, filters, multiple target pattern elements, attribute helpers and functional helpers. Additional adapted versions also cover unique lazy rules, libraries, multiple input models and multiple output models.

To evaluate the performance of the generated transformations, we extended three example transformations into a benchmark. We chose the *Families2Persons* example as it is well known, the *Make2Ant* example as it is supported by SyncATL and a simplified version of the *Class2Relational* as this is supported by Reactive ATL. For these transformations, we randomly generated change sequences of 20 elementary changes based on generated input models of different sizes. We assume that after each change, we need an updated target model. Thus, we either re-run the model transformation or simply propagate the change to the target model in case of the incremental NMF Synchronizations version.

To reduce the influence of chance, we repeated every measurement ten times. All NMF implementations work entirely online, so that no serialization or deserialization of models is involved. The ATL transformations run in a single process to reduce JVM warmup. For the ATL implementations, model serialization and deserialization is excluded from the time measurements. However, the measurements for NMF Synchronizations include the model manipulation[97], while the measurements for ATL only include the transformation. The introduced bias towards ATL is only marginal since the time for pure model manipulation can be neglected.

The benchmark is also publicly available online[98].

[97] Because NMF Synchronizations works online, it is hard to separate the time for change propagation from pure model manipulation.

[98] `https://github.com/georghinkel/atlbenchmark`

9.6.4.1. Families to Persons

The *Families2Persons* example is perhaps one of the most often studied example transformations in the MDE community. Both input metamodel, output metamodel and the transformation itself are entirely synthetic. With only 46 lines of ATL code, the transformation is also very small. Nevertheless, the transformation showcases a common model transformation problem: Both input metamodel and output metamodel represent people, but the representation of gender is fairly different. In the *Families* metamodel, the sex of a person is encoded through the containment hierarchy, i.e. a person is female if and only if it acts as a mother or a daughter of a family. Meanwhile, the *Persons* metamodel has a direct representation of gender using inheritance.

In terms of language features, the *Families2Persons* transformation consists of two matched rules with guards and helpers. Both transformation rules only have a single input pattern element and a single output pattern element.

As input, we randomly generate *Families* models of given sizes. For size n, we generate $\frac{n}{10}$ families each consisting of a father, a mother, two sons and two daughters. Afterwards, we randomly fill existing families with new sons or daughters until the desired number of elements is reached.

The change sequences are also generated based on the generated input models and consist of the following types of elementary changes (percentages in parentheses denote the probability of these changes):

- Add a new family consisting of a father, a mother, a son and two daughters (2% or if no family is present)
- Remove a son from a randomly selected family (10%)
- Remove a daughter from a randomly selected family (10%)
- Add a new son to an existing family (25%)
- Add a new daughter to an existing family (25%)
- Rename an existing son of an existing family (10%)
- Rename an existing daughter of an existing family (10%)
- Remove an existing family (8%)

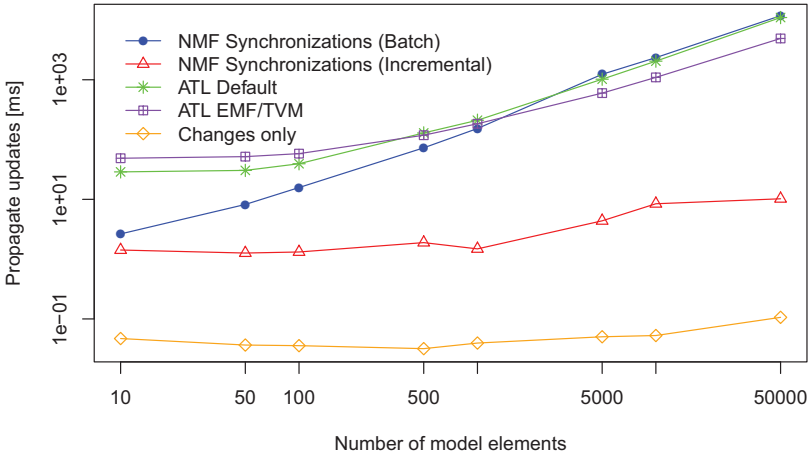


Figure 9.19.: Response times of a generated change sequence in the *Families2Persons* example for input sizes between 10 and 50,000 persons, both axes logarithmic.

The results for the *Families2Persons* example are depicted in Figure 9.19 with logarithmic scales in both axes. We depicted the runtime of NMF Synchronizations in batch mode and incremental mode, ATL in the default VM and in the EMFTVM and the time to only apply the changes to the source model.

The results show that already for very small input sizes, the incremental change propagation leads to smaller response times. For larger models, the gap between the incremental NMF Synchronizations solution and the batch solution increases, i.e. the speedup grows. For the largest considered model size of 50,000 model elements, the incremental NMF Synchronizations transformation is faster than reexecuting the original ATL transformation by a factor of 480 (EMFTVM) or 1074 (default VM). While the time for repeated transformation scales with the model size[99], the incremental change propagation scales with the size of the changes which is why the curves for model manipulation only and model transformation are roughly parallel on

[99] The execution of the ATL transformations seems to have a slight constant overhead. Probably, there is a more efficient way to automate running the ATL transformation. However, that overhead becomes negligible for large model sizes.

a logarithmic axis. Therefore, the results suggest that the generated NMF Synchronizations synchronization is fully incremental in the terminology of Giese and Wagner [74].

9.6.4.2. Make files to Ant build scripts

The *Make2Ant* transformation is about transforming between models of two different yet similar technologies for writing build scripts: Make and Ant. The difference between these models is only that the used metamodel for Make scripts has a boolean attribute denoting whether a shell line is shown to the user or not. The Ant model does not have this option but distinguishes more generally between code execution and informational messages to the user. The model transformation therefore has two different transformation rules depending on whether a shell line should be displayed or not. In the latter case, only an `Exec` element is created that executes the shell line. In the former case, both an `Echo` and an `Exec` element is created. With 88 lines of code, this transformation is slightly more elaborate than the *Families2Persons* example, but still very small.

In terms of language features, the transformation consists of five matched transformation rules, partially with guards and multiple output elements.

Similar to the *Families2Persons* transformation, we use generated models as inputs. To create n model elements, we generate $\frac{n}{10}$ macros, $\frac{n}{4}$ rules with a file dependency and a shell line. Then, we randomly add shell lines and file dependencies to randomly selected rules until the desired number of elements is reached.

The generated change sequences consist of the following elementary changes (again, probabilities in parentheses):

- Remove a random macro (4%)
- Add a new macro (7%)
- Change the name and the value of an existing macro (7%)
- Add a new file dependency to a randomly selected rule (10%)
- Change a randomly selected file dependency (10%)
- Remove a randomly selected file dependency (5%)

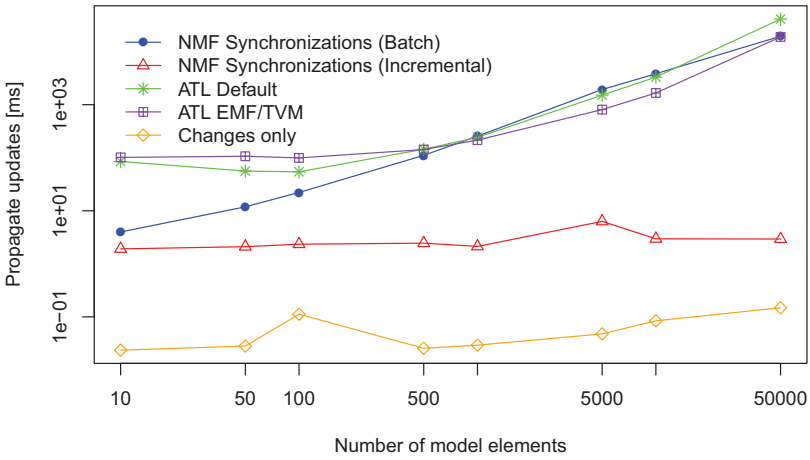


Figure 9.20.: Response times of a generated change sequence in the *Make2Ant* example for input sizes between 10 and 50,000 elements, both axes logarithmic.

- Add a shell line to a randomly selected rule (10%)
- Change a randomly selected shell line command (10%)
- Remove a randomly selected shell line command (5%)
- Add a new rule with a file dependency and a shell line (15%)
- Rename a randomly selected rule (10%)
- Remove a randomly selected rule (5%)
- Change a comment (3%)

The results of the *Make2Ant* example are depicted in Figure 9.20. They confirm the results of the *Families2Persons* example. For all model sizes, the incremental change propagation of the generated NMF Synchronizations transformation is significantly faster than reexecuting the ATL transformation after every model manipulation. For the largest model size of 50,000 model elements, we see a speedup of the incremental NMF Synchroniza-

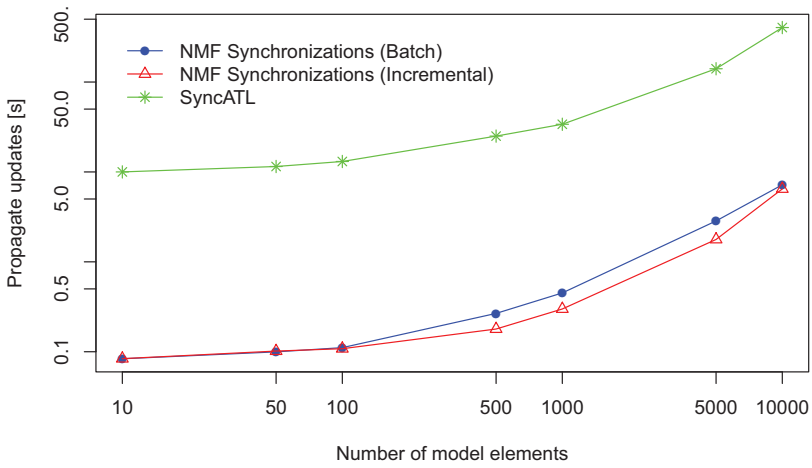


Figure 9.21.: Response times of a generated change sequence in the *Make2Ant* transformation. Time axis is logarithmic.

TIONS transformation of 5,175 compared to ATL executed in the EMFTVM or 11,367 compared to the default VM.[100]

Lastly, we compared the results with existing extensions to ATL that also aim at incremental processing. Unfortunately, both of these approaches are in a very early state such that it was not possible to compare the tools for a multitude of transformations.

For SyncATL [219], we were able to run the *Make2Ant* transformation. Similar to the last comparison, we randomly generated change sequences of 20 elementary changes and propagated them individually. For a fairer comparison[101] and because SyncATL works offline, we deserialize the last input model and serialize the result after each change propagation.

The results are depicted in Figure 9.21. Because the serialization overhead is added to both NMF SYNCHRONIZATIONS solutions, the difference between

[100] There is an outlier for model manipulation only at 5000 model elements. We think this partially due to the low accuracy for time measurements below 0.1ms and due to confounding effects such as garbage collection.

[101] Still, SyncATL needs to compare the models before each change propagation. However, we did not get SyncATL to compile and therefore used the compiled version from their website.

the two of them is marginal and the performance benefits of incremental computation are almost lost. However, still, SyncATL is roughly two orders of magnitude slower.

9.6.4.3. Class Diagrams to Relational Database Structures

The *ClasdsDiagrams2Relational* transformation takes a simple model of a class diagram as input and creates a relational database scheme capable to store instances of this class model. That is, a table is created for each class and all single-valued attributes are turned into columns. Multi-valued attributes are transformed into separate tables. In the scope of this benchmark, we had to use a minimalistic version of the example transformation in order to stay within the subset of the ATL language supported by Reactive ATL. In particular, unlike the original example transformation, this simplified version does not create a primary key column for each class. Columns with foreign keys do not have a type. Tables for multi-valued columns are not created.

In terms of used language elements, the used version of this example transformation is simplistic. It only consists of six matched transformation rules with guards and a helper, though the latter is not used.

As inputs, we again use generated input models. These input models consist of a package with primitive types and a set of randomly generated packages. We then generate $\frac{n}{2}$ classes and $\frac{n}{2}$ attributes where the attributes are randomly assigned to classes.

The generated change sequences consist of the following changes:

- Change the type of a randomly selected attribute (20%)
- Add a new attribute to a randomly selected class (20%)
- Rename a randomly selected attribute (20%)
- Remove a randomly selected attribute (10%)
- Rename a randomly selected class (10%)
- Remove a randomly selected class (5%)
- Add a new data type (5%)
- Add a new package (8%)

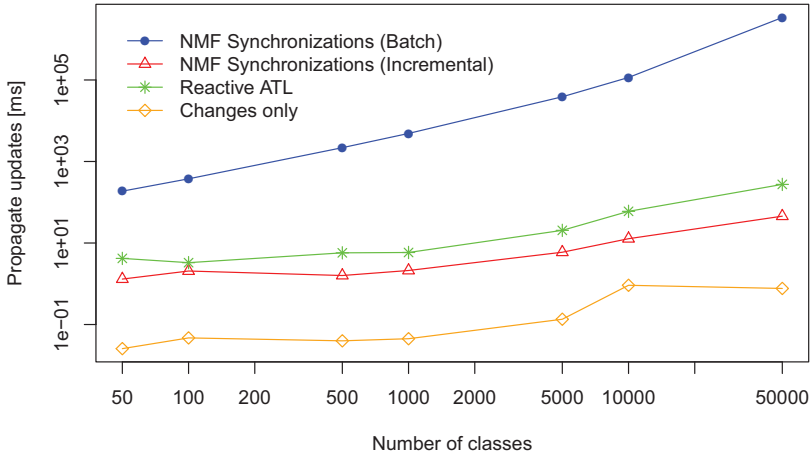


Figure 9.22.: Response times of a generated change sequence in the *Class2Relational* transformation for input sizes between 50 and 50,000 model elements, both axes are logarithmic.

- Remove a randomly selected package and all of its contents (2%)

Originally, we also planned to add classes or rename packages, but for some reason Reactive ATL threw a runtime error for these two change types.

The results achieved for the benchmark are depicted in Figure 9.22. Like NMF Synchronizations, the curve for Reactive ATL is flat, indicating that the response time to update the target model does not depend on the size of the model. However, the curve is above the curve for NMF Synchronizations with a constant distance. On a logarithmic scale, this indicates a constant factor: NMF Synchronizations appears to be roughly 2-5 times as fast as Reactive ATL. The average speedup of the incremental NMF Synchronizations version against the incremental version using Reactive ATL is 3.6 with a standard derivation of 1.4. Even though we achieved the fastest speedup of 6.0 for the largest model with 50,000 model elements, we do not think that there is a trend that the speedup is growing with the size of the input. Both incremental versions are much faster than repeatedly executing the transformation in a batch manner.

9.6.5. Summary

In this section, we presented a structured transformation from declarative ATL transformations to one-way synchronization blocks in NMF Synchronizations. This suggests that synchronization blocks are at least as expressive as the declarative parts of the ATL language. Most of this mapping is supported by our open-source tool ATL2NMFs.

Our approach is fundamentally different in comparison to already existing approaches to obtain an incremental execution of ATL transformations such as Reactive ATL and SyncATL. In these approaches, an incremental model synchronization for the ATL language is introduced by altering the ATL compiler, respectively extending the ATL VM. In this paper, an incremental model synchronization is introduced for ATL transformations by mapping the different concepts of the ATL language to NMF Synchronizations. The advantage of this approach is the support of incremental change propagation, which leads to better performance in the presence of small incremental changes to the source model. Further, the resulting transformation can be extended by the transformation developer to also account for offline synchronizations with appropriate heuristics when to create a correspondence between existing model elements.

The evaluation shows that our transformation can be used to gain a transformation that runs fully incremental and therefore is up to four magnitudes faster than a repetitive execution of the original ATL transformation run in the default VM or more than 5,000 times faster than repetitive execution of the original ATL transformation run in the EMFTVM. Compared to other approaches that execute ATL transformations incrementally, our approach has a speedup of roughly 3-4.

9.7. Case Study: Refactoring of Java Code

This section presents results based on the Transformation Tool Contest 2015 Java Refactoring case [131]. The idea behind this case is that most refactorings can be specified on a more abstract level than code. The case defined such a more abstract format and demanded solution authors for a bidirectional synchronization between this model and the Java code, represented in an arbitrary metamodel.

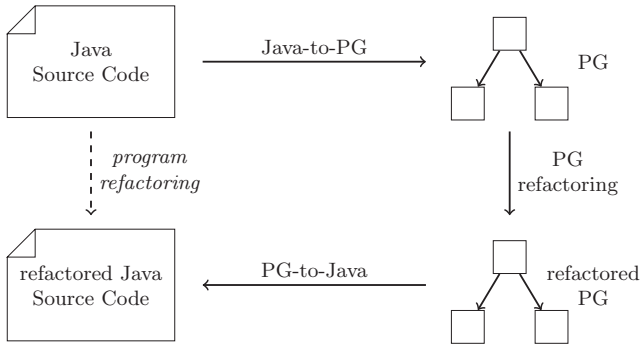


Figure 9.23.: Sketch of the Transformation Chain for the Java Refactoring case at the TTC 2015 [131]

At the TTC 2015, solutions to this case were submitted using FunnyQT [113], Spoon [160], NMF [94], VIATRA [189], eMoflon [163] and SDMLib [78].

9.7.1. Benchmark Setup

The benchmark intended three steps: Based on a Java model, at first a Program Graph (PG) model should be created. Then, refactoring operations should be performed based on this simplified PG model. Afterwards, the refactored PG model should be transformed back to a Java program. The transformation chain is depicted in Figure 9.23.

The PG model is somehow optimized for the proposed refactoring tasks as it already draws a very close connection between methods that have the same name (though defined in different classes with no inheritance relation between them). In particular, the PG model merely defines a method through its name and allows such a method to have different definitions in multiple classes.

A metamodel class diagram of the PG model is depicted in Figure 9.24.

The proposed refactoring operations are *Create Superclass* and *Pull Up Method*. The latter essentially assumes method definitions in two classes with a shared base class identical and replaces them by a method definition in the common

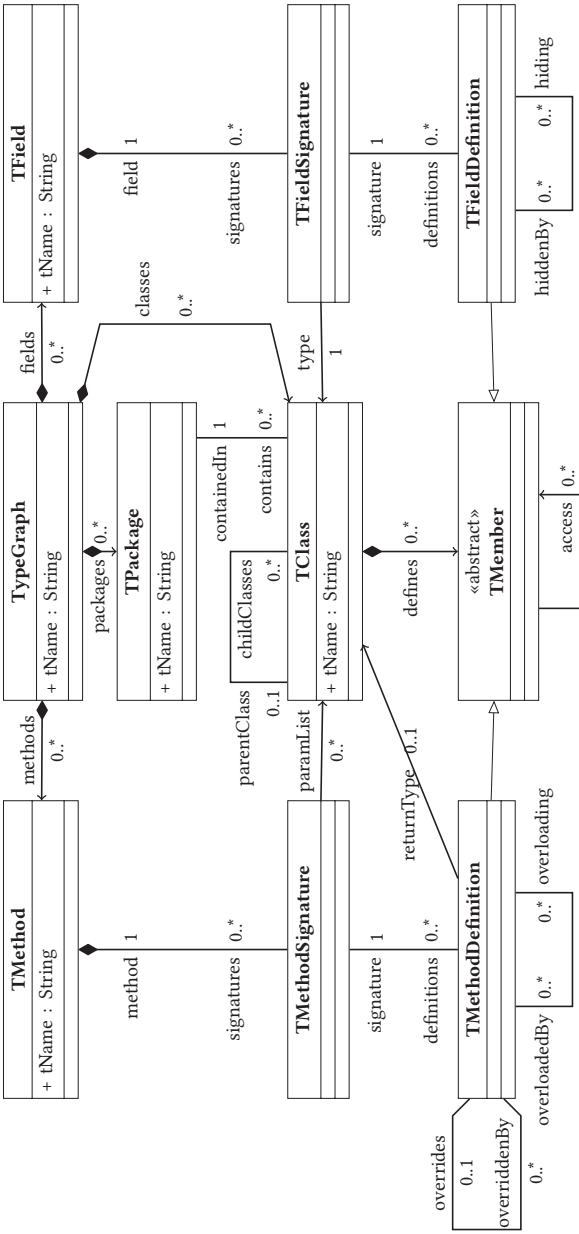


Figure 9.24.: The PG metamodel cf. [131]

base class. The *Create Superclass* refactoring simply creates a new superclass for all classes that have no superclass, yet.

Details on the benchmark setup can be obtained in the original benchmark description [131].

9.7.2. Validation Goals

The goals of this case study are as follows:

Applicability The reason that this case study is interesting in the scope of this thesis is that it is meant for bidirectional model transformations. In particular, a RHS model is created from a given LHS model and afterwards, changes to the RHS model should be propagated back to the LHS model. The application scenario of refactoring tasks is also interesting, though we think that the practical use of the suggested PG model is limited.

Understandability As a case study taken from the TTC, we have access to the open peer reviews, i.e. expert opinions from external researchers about the solutions. Therefore, we can compare the understandability of the NMF approach in relation to other solutions.

Correctness Again, we check the correctness indicators as indicated by the case description.

Unfortunately, the NMF solution could not be run in the ARTE framework that was used to record the performance of the solutions. Therefore, we do not compare the NMF solution with other solutions with regard to performance in this case study.

9.7.3. NMF Solution

The description of the NMF solution is based on the original solution submission [94].

```
1  class JavaPGSynchronization : ReflectiveSynchronization {
2     public class Class2Class : SynchronizationRule<IClass, ITClass> {
3        public override void DeclareSynchronization() {
4           Synchronize(cl => cl.Name, cl => cl.TName);
5           SynchronizeMany(SyncRule<Member2Member>(),
6              cl => cl.Members.Where(m => m is ClassMethod || m is Field),
7              cl => cl.Defines);
8           Synchronize(this,
9              cl => cl.Extends as IClassifierReference != null
10                ? (cl.Extends as IClassifierReference).Target as IClass
11                : null, RegisterNewBaseClass,
12             cl => cl.ParentClass);
13       }
14    }
15 }
```

Listing 9.18: Synchronization of classes in JaMoPP and the PG metamodel

We use JaMoPP [86] to load and write Java files and translate them into a model representation. Since the NMF meta-metamodel NMeta is compatible with Ecore, we can easily transform the JaMoPP metamodel to an NMeta metamodel and consume the JaMoPP generated XMI representations of the input files directly. In between, we use NMF Synchronizations for the transformations and regular C# code for the actual refactorings in the PG model.

To ensure that the refactoring operations performed in the model manipulation have an effect to the code model, we run the transformation from the JaMoPP model to the PG model with *two-way* change propagation. We actually do not require changes from the JaMoPP model to the PG model because we do not intend to change it, but NMF Synchronizations does not support a change propagation from the target to the source model of a model transformation, only.

Listing 9.18 shows an excerpt of the model synchronization that synchronizes classes.

In line 3, we declare that `Class2Class` is a synchronization rule synchronizing JaMoPP classes with PG classes. Line 7 specifies that whenever we found such two classes that correspond (decided by the method `ShouldCorrespond`), their names should be synchronized. Line 8-10 specify that each member of

a JaMoPP class should correspond to a definition in the PG. The details for this correspondence are left to the `Member2Member` rule.

Lines 11-15 specify that the base classes should be synchronized. The current rule `Class2Class` should be used to identify corresponding base classes as well, explaining the `this` parameter in Line 11. However, whereas the base class of a Java class in the PG metamodel is available directly as a reference, the base class in JaMoPP is encoded in a classifier reference, making the expression to obtain the base class slightly more complex. As a consequence, NMF Synchronizations is not able to infer how to revert the expression and we have to specify this (how a JaMoPP class is assigned another class as a base class) through another method, namely `RegisterNewBaseClass`. With this method, the behavior how to assign a JaMoPP class a new base class is implemented in regular imperative code.

The implementation of `Member2Member` in case of methods is presented in Listing 9.19. In this listing, again Line 1 declares `Method2MethodDefinition` as a synchronization rule from JaMoPP methods to PG method definitions. A JaMoPP method should correspond to a PG method definition in a given scope if the methods have the same name here. We specify the exact behavior in lines 3-10. Since the structure of the PG metamodel is very different to JaMoPP in this regard, the method is a few lines long.

Line 13 marks the synchronization rule instantiating for the `Member2Member` rule. This means, if a member is a method, then the rule `Method2MethodDefinition` should be used to synchronize members, regardless of the transformation direction. `Field2FieldDefinition` is in place for synchronizing fields.

Line 14 denotes that the name of a method in JaMoPP should be kept consistent with the name of the method in the PG model. Here, the two way change propagation has enormous consequences. If we changed the name of a method in JaMoPP, the change is propagated to the PG `TMethod` element. However, this change is propagated back to the JaMoPP model causing all methods that are connected to this PG method to change their name accordingly, regardless of their declaration scope or signature. So we have specified a very powerful rename refactoring in just a single line of code. While this behavior is consistent with the case description, we argue that it is hard to foresee. However, this is not a drawback of our approach but rather a consequence from the simplistic point of view of the PG model.

```
1  public class Method2MethodDefinition : SynchronizationRule<IMethod,
       ITMethodDefinition> {
2     public override bool ShouldCorrespond(IMethod left, ITMethodDefinition
          right, ISynchronizationContext context) {
3        var sig = right.Signature;
4        if (sig == null) return false;
5        var meth = sig.Method;
6        if (meth == null) return false;
7        return left.Name == meth.TName;
8     }
9     public override void DeclareSynchronization() {
10       MarkInstantiatingFor(SyncRule<Member2MemberDefinition>());
11       Synchronize(meth => meth.Name, meth => meth.Signature.Method.TName);
12       LeftToRight.Require(Rule<Method2MethodSignature>(), meth => meth.Name,
13          meth => meth.Parameters.Select(p => GetBaseClass(p.TypeReference)).
                 AsItemEqual(),
14          (meth, signature) => meth.Signature = signature);
15    }
16 }
```

Listing 9.19: The synchronization rule for method definitions

Since the underlying NTL transformation rules of a synchronization rule in NMF Synchronizations are still accessible, we can add a dependency to the `Method2MethodSignature` that creates a `TMethodSignature` element for the given name and parameter list. For a given name and parameter list, the transformation engine ensures that only one method signature element is created. This transformation rule calls another rule `Method2Method` that creates a method element for each string that appears as a method name in the JaMoPP model.

These transformation rules `Method2Method` and `Method2MethodSignature` are called any time the *LeftToRight* rule of the synchronization rule `Method2MethodDefinition` are called. That is done either initially for each method in the JaMoPP model (restricted to at most once per input names and parameter lists) and as well for any new JaMoPP method that is added to the JaMoPP model afterwards.

The refactoring part of our solution uses straight forward imperative code to achieve the refactoring operations. As the *Create Superclass* is straight forward to implement in classic C# code, we omit a description. The implementation of the *Pull Up Method* refactoring is shown in Listing 9.20.

```
public bool PullUpMethod(TypeGraph typeGraph) {
  foreach (var method in typeGraph.Methods) {
    foreach (var signature in method.Signatures) {
      var methodsGroupsToPull = from def in signature.Definitions
                          where def.Overriding == null
                          group def by (def.Parent as TClass).ParentClass into
                               methods
                          select methods;
      foreach (var methodGroup in methodsGroupsToPull.Where(group => group.
           Count() >= 2)) {
        if (methodGroup.Key != null) {
          var first = methodGroup.First();
          var firstParent = first.Parent as ITClass;
          methodGroup.Key.Defines.Add(first);
          firstParent.Defines.Remove(first);
          foreach (var m in methodGroup.Skip(1)) {
            (m.Parent as ITClass).Defines.Remove(m);
          }
        }
      }
    }
  }
}
```

Listing 9.20: The implementation of *Pull Up Method*

In this listing, we iterate through all methods and their signatures. For each signature, we query the method definitions that follow this signature and group them by parent classes. If there is a group with at least two items, which means that a method definition is present in at least two subtypes of a given class, then we pick the first method definition and add it to the base class. All other method definitions are removed.

Given the conciseness of this specification based on the `TypeGraph` metamodel, we see no reason to use a specialized language for the refactoring. However, due to the online synchronization, we have to be careful to always keep the model in a consistent state, we must not discard the method that should stay as otherwise the connected implementation in the JaMoPP model would be lost. This is very important because in the PG model, there is no information on how a method is implemented, this information is only stored in the original JaMoPP model. Using for example a new `TMethodDefinition` object in the PG model would be sufficient for the PG model, but

Tool		Comprehensibility	Readability
eMoflon	[163]	4.5	5
FunnyQT	[113]	3	4
NMF	[94]	5	5
SDMLib	[78]	4	3
Spoon	[160]	4.5	4.5
VIATRA	[189]	3.5	3.5

Table 9.5.: Open Peer Review Results for the TTC 2015 Java Refactoring Case

NMF Synchronizations would not be able to trace that new model element to existing code. By reusing the existing method definition, we allow the synchronization engine to trace the code for that method definition.

9.7.4. Results

Because the NMF solution could not be integrated into the benchmark framework, the tests executed by the framework ARTE have been checked manually.

Furthermore, the questionnaire given to attendees of the TTC workshop 2015 only contained a single question asking to which degree people liked the respective solution. The results are not publicly available and even if they were, their expressiveness is unclear because it is not possible to separate an evaluation of the tool from personal bias towards a technology or the quality of presentation.

However, the results of the open peer reviews are indeed publicly available[102]. The open peer review form included specific questions to the quality comprehensibility and readability. The average values for the open peer reviews is depicted in Table 9.5.

The table shows that the NMF was the only solution that got the maximum points for comprehensability and readability. This may be because the NMF solution was the only solution that implemented the intended behavior of a

[102] `https://docs.google.com/spreadsheets/d/1k1IOjjlXoldu9OLF6KlVXmRAdUoxBf6FOC3fJxmIe1c/edit?usp=sharing`, retrieved 26 Sep 2017

backpropagation of changes of the RHS model to the LHS. However, there were only two responses for each solution, so the results are not significant. The reference solution presented in [131] was not evaluated in the open peer reviews.

9.7.5. Summary

As the open peer reviews show, the NMF solution to this case is easy to read and comprehend. The open peer review results even suggest that it is easier to comprehend than other solutions to this case.

The main insight from the Java Refactoring case for us is that the bidirectional model synchronization of structurally different models is a powerful yet dangerous tool. Powerful because it allows to specify some refactoring operations like renaming in a very concise way. It is dangerous because it is opaque to the developer that the code model is synchronized with a refactoring model. This synchronization yields that when someone changes the name of a method in the code model, automatically all methods with the same name are renamed as well. On the other hand, if a methods name is changed into one that already exists, then the method elements in the program graph model are not merged, leading to an inconsistent behavior. In particular, as soon as this operation is performed and one changes the name of such a method in the JaMoPP model, some methods are renamed but others are not as they are synchronized with a different method element in the program graph model.

This is of course a more general problem of unclear semantics synchronizing structurally and semantically heterogeneous models with overlapping semantics. It not only related to our solution. A solution for this dilemma would be to disable two-way synchronization but restrict to one-way synchronization against the transformation direction, i.e. that changes in the target model are propagated back to the source. This is not implemented in NMF Synchronizations because we think that it is a rather exotic use case.

9.8. Case Study: Incremental Views in the Smart Grid Domain

The complexity of cyber-physical systems makes it inevitable to divide the system into multiple subsystems that operate in different domains. In many of these domains, standards exist that the respective subsystem has to comply with or for which a lot of tools can be reused.

For example, the smart grid is a cyber-physical system that spans the physical structures of the electricity network and the system of software systems that monitor, control, and repair the system in case of outages. Currently, many heterogeneous systems and standards have to interoperate to achieve the desired reliability, stability, and efficiency of the electricity network.

Because each of these standards describe different aspects of the system, models according to these standards often have to be combined if multiple aspects are required to gain insights about the system. Applying model-driven engineering, model views are a tool to extract information from multiple models without confronting the user with unnecessary information for a particular analysis.

In the area of smart grids, an additional challenge is the size of the models and the frequency of changes. In combination, this means that very large amounts of data have to be processed in a very short amount of time. However, the changes usually only affect small proportions of the model which is why an incremental view computation appears beneficial.

This case study was submitted to the TTC 2017 [97] and accepted as a contest case. Two solutions have been submitted, the NMF solution [95] and a solution using eMoflon [162].

9.8.1. Benchmark Setup

In the area of smart grids, the relevant standards are IEC 61970/61968, IEC 61850 and IEC 62056. A description of these standards can be found in the master thesis of Victoria Mittelbach [150] and is replicated here for self-containment of this dissertation.

IEC 61970/61968 The IEC 61970 standard defines the *Common Information Model (CIM)* which is used to describe the physical components, measurement data, control and protection elements. It is defined in UML notation. The IEC 61968 standard is an extension of the CIM for the distribution network [116]. It is also called *distributed CIM (DCIM)*

IEC 61850 The IEC 61850 standard is a series of standards for substations with the purpose of supporting interoperability of intelligent electronic devices (IED) in substation automation systems. It defines the *Abstract Communication Service Interface* with a mapping to concrete communication protocols, the XML-based *Substation Configuration Description Language (SCL)*, and the *Logical Node (LN)* model that describes power system functions [115].

IEC 62056 *COSEM (Companion Specification for Energy Metering)* is the international standard for data exchange for meter reading, tariff and load control in the domain of electricity metering. It works together with the *Device Language Message Specification (DLMS)*. Together, they provide a communication profile to transport data from metering equipment to the metering system and to define a data model and communication protocols for data exchange [9].

While these standards are useful in their domain, one has to combine the information represented by these standards to detect and prevent outage situations. Burger, Mittelbach and Koziolek presented a model-based outage management system based on the master thesis of Victoria Mittelbach that synchronizes models of these standards and consists of a set of 15 views to help operators to manage outage situations [150, 37].

In the scope of the proposed benchmark, we focus on two model views contained in the model-based outage management system. A rather simple view is created to detect outages while a second slightly more complex view supports the prevention of outages.

For both tasks, we present the implementation of the view in ModelJoin [36], a language to specify both the view type and the view definition in a single specification. From the ModelJoin specification, an idiomatic QVT-O model transformation is generated. Due to space limitations, we do not show the

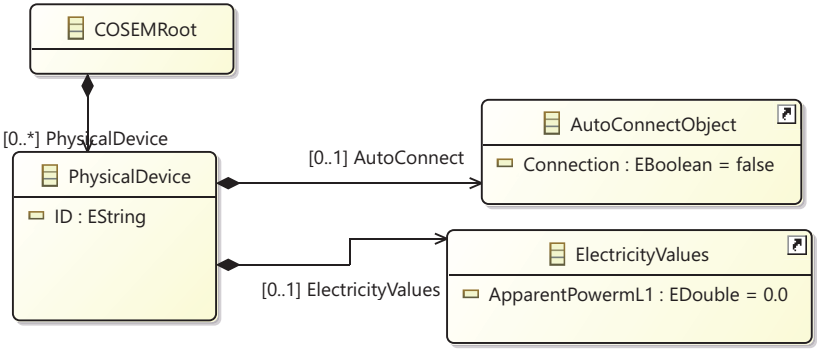


Figure 9.25.: Excerpt from the COSEM metamodel relevant for task 1

generated transformation but it is available in the benchmark resources[103] for reference.

9.8.1.1. Task 1: A simple view to detect outages

To detect an outage, we use the fact that a smart meter cannot send any data when it is cut off power supply. If this happens, the system can try to reach the meter but will receive a connection failure notification. A single cut off smart meter might be the result of a failure but if many smart meters in a closely connected area fail, this may indicate an outage. This is used to detect outages without relying on customer feedback.

The information that a connection to a smart meter is lost is depicted in the COSEM model. The relevant excerpt for this task is depicted in Figure 9.25. It has to be matched with the corresponding physical devices in the CIM model where its location is stored. A relevant excerpt of the latter is depicted in Figure 9.26.

An implementation of this view in ModelJoin is depicted in Listing 9.21. It consists of a single join statement that specifies that information from the `MeterAsset` elements in the CIM model and `PhysicalDevice` elements in the COSEM model should be joined based on their IDs. For each such a match,

[103] `https://github.com/georghinkel/ttc2017smartGrids`

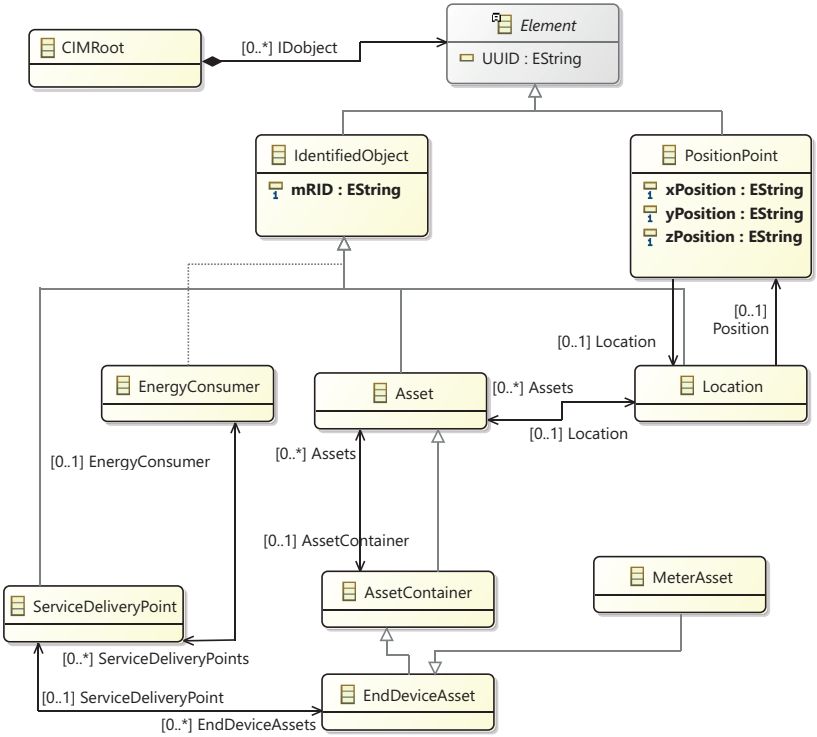


Figure 9.26.: Excerpt from the CIM metamodel relevant for task 1

the connection and electricity values from the COSEM model should be kept in the view as well as the location from the CIM model.

This reference to location and position point shall respect referential integrity. This means, if two meter assets in the CIM model reference the same location, their joins in the view should also reference the same `Location` element in the view.

Based on the view specification in Listing 9.21, the view type in Figure 9.27 is generated.

```
1  theta join CIM.IEC61968.Metering.MeterAsset with COSEM.PhysicalDevice where "
       CIM.IEC61968.Metering.MeterAsset.mRID␣=␣COSEM.PhysicalDevice.ID" as
       jointarget.EnergyConsumer {
2    keep calculated attribute "COSEM.PhysicalDevice.AutoConnect.Connection" as
         EnergyConsumer.Reachability:Integer
3    keep calculated attribute "COSEM.PhysicalDevice.ElectricityValues.
         ApparentPowermL1" as EnergyConsumer.PowerA:Double
4    keep calculated attribute "COSEM.PhysicalDevice.ElectricityValues.
         ApparentPowermL2" as EnergyConsumer.PowerA:Double
5    keep calculated attribute "COSEM.PhysicalDevice.ElectricityValues.
         ApparentPowermL3" as EnergyConsumer.PowerA:Double
6    keep calculated attribute "CIM.IEC61968.Metering.MeterAsset.
         ServiceDeliveryPoint.EnergyConsumer.mRID" as EnergyConsumer.ID:String
7    keep calculated attribute "if␣CIM.IEC61968.Metering.MeterAsset.
         ServiceDeliveryPoint.EnergyConsumer->oclIsKindOf(CIM.IEC61970.
         LoadModel.ConformLoad)␣then␣CIM.IEC61968.Metering.MeterAsset.
         ServiceDeliveryPoint.EnergyConsumer.ConformLoadGroup.SubLoadArea.
         LoadArea.ControlArea.mRID␣else␣CIM.IEC61968.Metering.MeterAsset.
         ServiceDeliveryPoint.EnergyConsumer.NonConformLoadGroup.SubLoadArea.
         LoadArea.ControlArea.mRID␣endif" as Consumer.ControlAreaID:String
8    keep outgoing CIM.IEC61968.Assets.Asset.Location as type jointarget.
         Location {
9      keep outgoing CIM.IEC61968.Common.Location.Position as type jointarget.
           PositionPoint {
10       keep attributes CIM.IEC61968.Common.PositionPoint.xPosition,
11       CIM.IEC61968.Common.PositionPoint.yPosition,
12       CIM.IEC61968.Common.PositionPoint.zPosition
13     }
14   }
15 }
```

Listing 9.21: Task 1 realized in ModelJoin

9.8.1.2. Task 2: A view to prevent outages

The analysis algorithms to detect system disturbances proposed in [37] work on *phasor measurement data*:

> Their basic concept is to compare the current phasor data of the traveling voltage wave with a historic set of normal phasor data and calculate an equality indicator like a correlation coefficient. This is compared with a certain benchmark. If it lies above, a failure is indicated. To enable this, the following information is necessary: a historic set of normal phasor data of that section, a

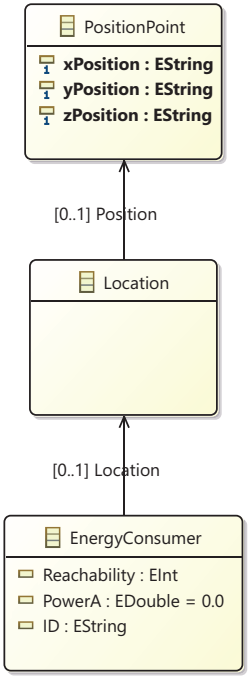


Figure 9.27.: The Viewtype for the Outage Detection Task

> matrix of the current phasor data and a calculation mechanism to compare the two followed by a comparison mechanism to decide if it is a failure or not. [150]

Task 2 requires to match elements from all three domain standards. For the COSEM standard, the used metamodel excerpt is very similar to Task 1. The relevant metamodel excerpts for CIM and the substation standard are depicted in Figure 9.28 and Figure 9.29, respectively.

The analysis viewtypes will not provide the analysis result but only the *matrix of phasor data* for the comparison. Six queries were defined in [150] that all have the same structure and provide the three-phase measurements of voltage, frequency, current, active power, reactive power and apparent power.

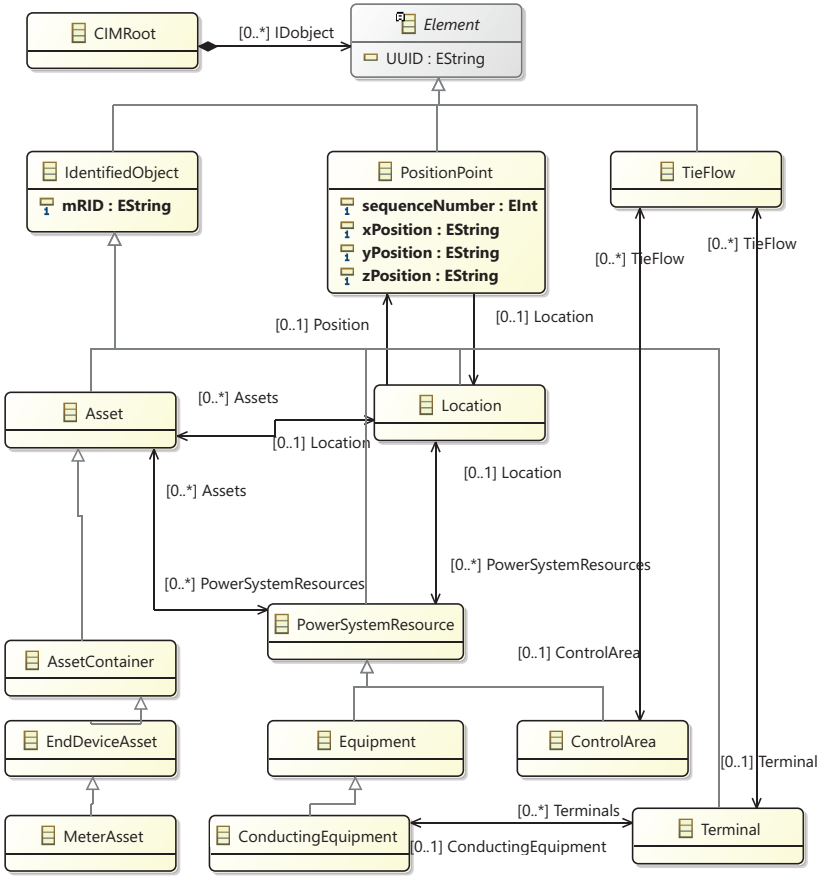


Figure 9.28.: An excerpt from the CIM metamodel relevant for Task 2

Such analysis viewtypes can be used together with a basic network topology view to calculate the exact location of a failure. Phasors are traveling waves in the system, which means that the failure travels with the wave through the grid. Therefore it is important to find its origin. The topology viewtype includes the length of the transmission line segments. They can be used together with the timestamp of the measured phasor to calcu-

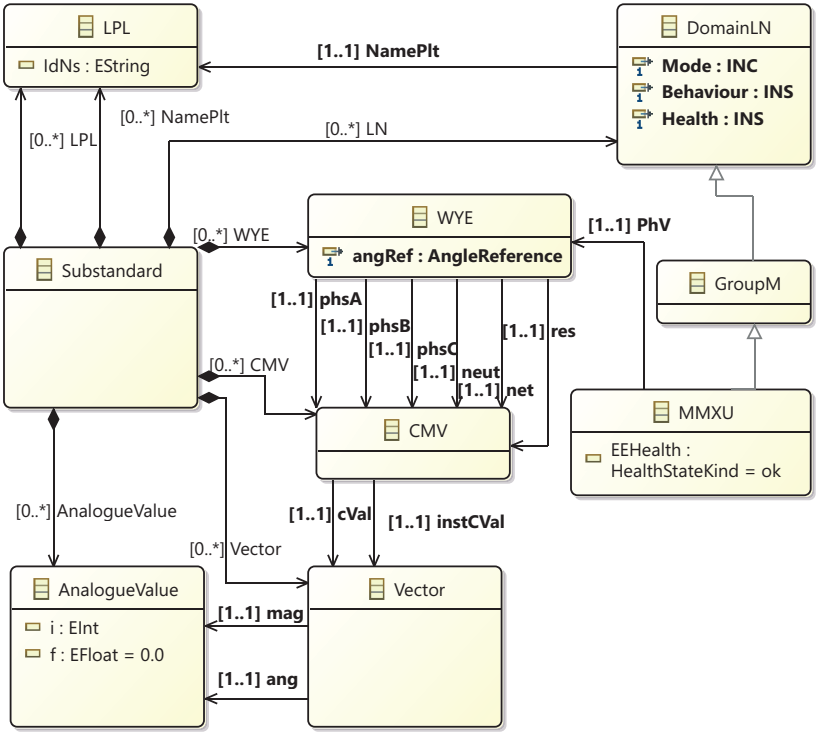


Figure 9.29.: An excerpt from the Substation standard relevant for Task 2

late from where the wave came and where it was when the failure started. This is its origin [167]. [150]

The view type generated from the ModelJoin specification of the query is depicted in Figure 9.30.

The biggest difference to Task 1 besides the increased complexity (the ModelJoin specification used as reference is more than four times as long) is the fact that this view definition keeps subtypes of some model elements. If an energy consumer in a service delivery point is a `ConformLoad`, then the view computation should be different to the case when the energy consumer is a `NonConformLoad` because the control area should be included in the view, but

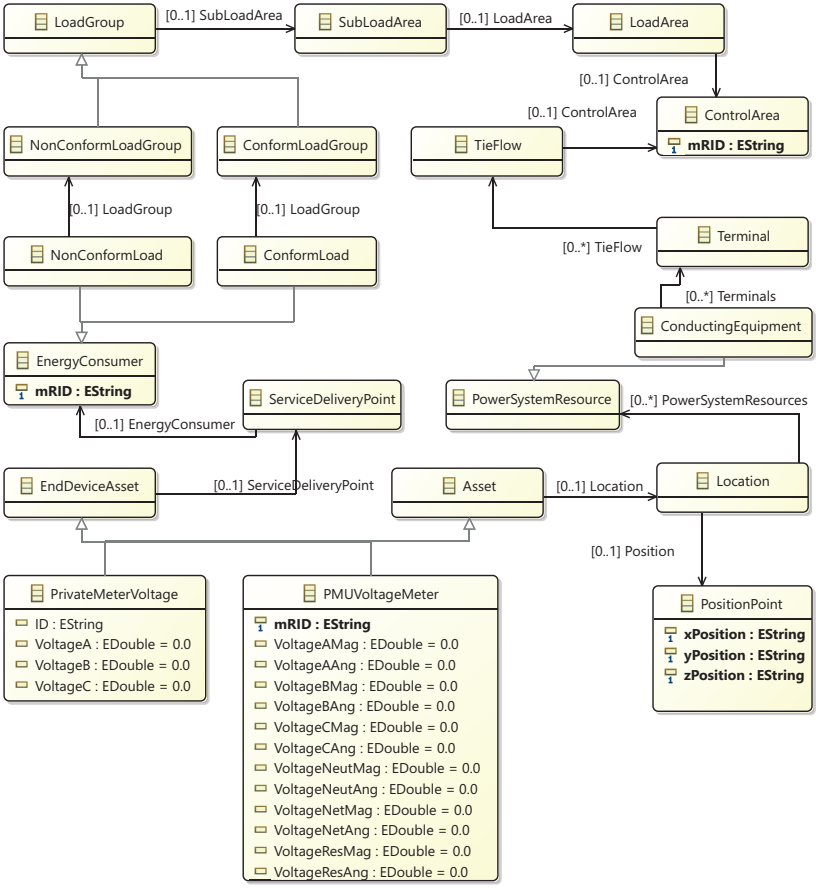


Figure 9.30.: The Voltage Three-Phase Measurement Matrix

the path to the control area in the CIM model depends on the type of this energy consumer.

9.8.2. Validation Goals

The validation goals of this case study are as follows:

Applicability The Smart Grid as another example of a cyber-physical system is again an interesting industry-relevant application area. The type of transformation, model views that match model elements from multiple models and combine the information from both, is also a highly relevant area for model transformations.

Performance Because elementary changes in a smart grid environment happen very frequently, the response time from a model change to an updated view is again very important. In particular, we want to compare the response time with the reference solution in ModelJoin as well as with other solutions.

Memory Consumption Similar to the Train Benchmark, we measure the impact of incremental execution to the memory consumption of the benchmark.

Understandability As the Smart Grid case also is a case study presented at the TTC contest, we use the responses from open peer reviews and from questionnaires at the workshop to evaluate the understandability of the solutions.

Correctness The correctness indicators in this case study are purely the number of elements in the output model. Nevertheless, we intend to use these indicators as correctness indicators of our solution.

9.8.3. NMF Solution

The description of the NMF solution is based on the original solution submission to the TTC 2017 [95].

We discuss the solutions to the outage detection and the outage prevention tasks separately in Sections 9.8.3.1 and 9.8.3.2.

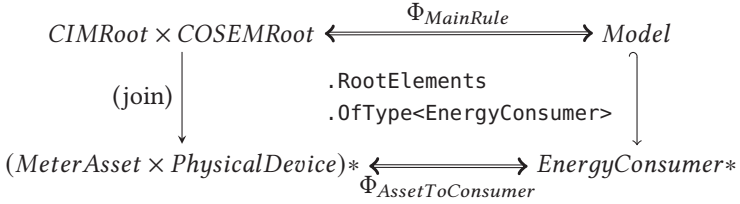


Figure 9.31.: The join in the outage detection task formulated in a synchronization block

```
1  public class MainRule : SynchronizationRule<Tuple<CIMRoot, COSEMRoot>, Model>
       {
2    public override void DeclareSynchronization() {
3      SynchronizeManyLeftToRightOnly(SyncRule<AssetToConsumer>(),
4        sg => from pd in sg.Item2.PhysicalDevice
5              join ma in sg.Item1.IDobject.OfType<IMeterAsset>()
6              on pd.ID equals ma.MRID
7              select new Tuple<IMeterAsset, IPhysicalDevice>(ma, pd),
8        target => target.RootElements.OfType<IModelElement,
               OutageDetectionJointarget.IEnergyConsumer>());
9    }
10 }
```

Listing 9.22: The implementation of the main rule for outage the outage detection task

9.8.3.1. Outage Detection

In NMF Synchronizations, the support for multiple input pattern elements is rather limited. Therefore, the easiest way to support multiple input pattern elements in NMF Synchronizations is to simply use tuples as inputs. Then, the model matching has to be adapted to match tuples instead of elements. Therefore, the main rule synchronizes a tuple of the CIM model and the COSEM model with the resulting view model.

In a synchronization block, the main join of meter assets with physical devices is depicted in Figure 9.31, where we abbreviated the join expression. The implementation of this matching is depicted in Listing 9.22.

```
public class AssetToConsumer : SynchronizationRule<Tuple<IMeterAsset,
     IPhysicalDevice>, IEnergyConsumer> {
  public override void DeclareSynchronization() {
    SynchronizeLeftToRightOnly(
      asset => Convert.ToInt32(asset.Item2.AutoConnect.Connection),
      e => e.Reachability);
    SynchronizeLeftToRightOnly(
      asset => asset.Item2.ElectricityValues.ApparentPowermL1,
      e => e.PowerA);
    SynchronizeLeftToRightOnly(
      asset => asset.Item1.ServiceDeliveryPoint.EnergyConsumer.MRID,
      e => e.ID);
    SynchronizeLeftToRightOnly(
      asset => asset.Item1.ServiceDeliveryPoint.EnergyConsumer is ConformLoad
           ?
       ((ConformLoad)asset.Item1.ServiceDeliveryPoint.EnergyConsumer)
            .LoadGroup.SubLoadArea.LoadArea.ControlArea.MRID :
       ((NonConformLoad)asset.Item1.ServiceDeliveryPoint.EnergyConsumer)
            .LoadGroup.SubLoadArea.LoadArea.ControlArea.MRID,
      e => e.ControlAreaID);
    SynchronizeLeftToRightOnly(SyncRule<LocationToLocation>(),
      asset => asset.Item1.Location, e => e.Location);
  }
}
```

Listing 9.23: Implementation of kept attributes and references in the outage detection task

Because .NET has a hard implementation of generics[104], a type filter can be easily specified by passing generic type arguments. NMF also contains an overload of the `OfType` type filter that accepts two type arguments and keeps the collection interface.

In particular, the incrementalization system NMF Expressions that is underlying NMF Synchronizations does support joins, available also through the query syntax of C#. A second synchronization rule then implements the kept attributes for every such a tuple, as depicted in Listing 9.23.

In particular, for each attribute that should be synchronized, the synchronization rule contains a synchronization block that is responsible for the synchronization of this attribute. As an example, the synchronization of the

[104] This means that the generic type arguments are still available at runtime.

reachability information converted to an integer value is depicted in Lines 3 to 5 of Listing 9.23.

Because these synchronization blocks are unidirectional, also more complex patterns such as the synchronization block in Lines 7 to 15 are possible where the selector obtains the control area differently according to the type of the energy consumer at the service delivery point.

Two further synchronization rules synchronize location and position point. They are omitted in this thesis to save space.

9.8.3.2. Outage Prevention

In the implementation of the outage prevention task, the principle approach to use tuples to synchronize multiple inputs is the very same approach as in the outage detection task. The implementation of the main rule is depicted in Listing 9.24.

In this listing, we used the alternative method chaining syntax for the join. Both syntaxes are equivalent, as the compiler converts the query syntax into the method chaining syntax.

To handle the different transformation of the various subtypes of a power system resource, we utilize the rule instantiation feature of NMF Synchronizations. With a rule instantiation, the isomorphism represented by a synchronization rule can be refined for a subset of model elements.

An example of synchronization rule instantiation for conducting equipment is depicted in Listing 9.25. This means that whenever a power system resource is a conducting equipment, also its terminals are synchronized.

9.8.4. Results

Unfortunately, only one solution other than the NMF solution was submitted to the TTC contest, created Sven Peldszus et al. [162] using eMoflon. This solution tried to use the incrementalization capabilities of eMoflon but unfortunately, the matching could not be done incrementally with eMoflon such that this solution is also working in a batch manner, i.e. the views are recomputed after each change sequence from scratch.

```
public class MainRule :
   SynchronizationRule<Tuple<CIMRoot, COSEMRoot, Substandard>, Model> {
  public override void DeclareSynchronization() {
   SynchronizeManyLeftToRightOnly(SyncRule<MMXUAssetToVoltageMeter>(),
     dr => dr.Item1.IDobject.OfType<IMeterAsset>()
           .Join(dr.Item3.LN.OfType<IMMXU>(),
                asset => asset.MRID,
                mmxu => mmxu.NamePlt.IdNs,
                (asset, mmxu) => new Tuple<IMeterAsset, IMMXU>(asset, mmxu)),
     model => model.RootElements.OfType<IModelElement, IPMUVoltageMeter>());

   SynchronizeManyLeftToRightOnly(SyncRule<DeviceAssetToPrivateMeterVoltage
        >(),
     dr => dr.Item1.IDobject.OfType<IEndDeviceAsset>()
           .Join(dr.Item2.PhysicalDevice,
                asset => asset.MRID,
                pd => pd.ID,
                (asset, pd) => new Tuple<IEndDeviceAsset, IPhysicalDevice>(
                     asset, pd)),
     model => model.RootElements.OfType<IModelElement, IPrivateMeterVoltage
          >());
  }
}
```

Listing 9.24: The implementation of the main rule in the outage prevention task

```
public class PowerSystemResource2PowerSystemResource
    : SynchronizationRule<IPowerSystemResource, IPowerSystemResource> {
  public override void DeclareSynchronization() {}
}
public class ConductingEquipment2ConductingEquipment
    : SynchronizationRule<IConductingEquipment, IConductingEquipment> {
  public override void DeclareSynchronization() {
   SynchronizeManyLeftToRightOnly(SyncRule<Terminal2Terminal>(),
      conductingEquipment => conductingEquipment.Terminals, equipment =>
           equipment.Terminals);
   MarkInstantiatingFor(SyncRule<PowerSystemResource2PowerSystemResource>());
  }
}
```

Listing 9.25: Transforming power system resources

Because there is only one other solution, there is only one peer review of the NMF solution which is why we do not discuss the results of the open peer reviews. Further, also only three responses were returned for the smart grid case at the TTC, possibly due to other parallel sessions at the STAF 2017.

The performance results for propagating 100 change sequences is depicted in Figure 9.32 for the *Outage Detection* task and in Figure 9.33 for the *Outage Prevention* task. The diagrams show the change propagation times on a logarithmic plot against the iterations for the reference solution in ModelJoin, eMoflon and the NMF solution in batch and incremental mode.

For the *Outage Prevention* task, the reference solution in ModelJoin unfortunately ran out of memory during the updates. Thus, no performance times can be depicted.

The results indicate that the NMF solution in batch mode is already the fastest among the batch implementations. Furthermore, if one switches the execution mode to incremental, then this yields another speedup of roughly more than a magnitude.

The performance curve for the incremental change propagation is more rough than the performance for the batch execution. This is because propagating the change depends much more on the actual changes than rerunning the view computation on the entire (changed) model. However, interestingly, we see some spikes in the otherwise smooth curve for the batch execution. We think that this is due to garbage collection taking place.

The memory measurement for the solutions is depicted in Figure 9.34. It shows the average working set size for the different change sequences, depending on the view task. Because the memory measurement is performed before and after each transformation step, we also see a memory consumption of the ModelJoin solution for the *OutagePrevention* case.

The result may seem surprising as the memory consumption for the incremental NMF solution is lower by multiple orders of magnitude. The reason for this effect is that although more memory may be required to store assets such as DDGs or trace entries, non-incremental batch solution have to load the models after each modification from persistent storage and discard the old memory. However, the garbage collector typically only starts to free memory as soon as the main memory gets to its capacity limits. Therefore, we see a significantly higher memory consumption for non-incremental solutions.

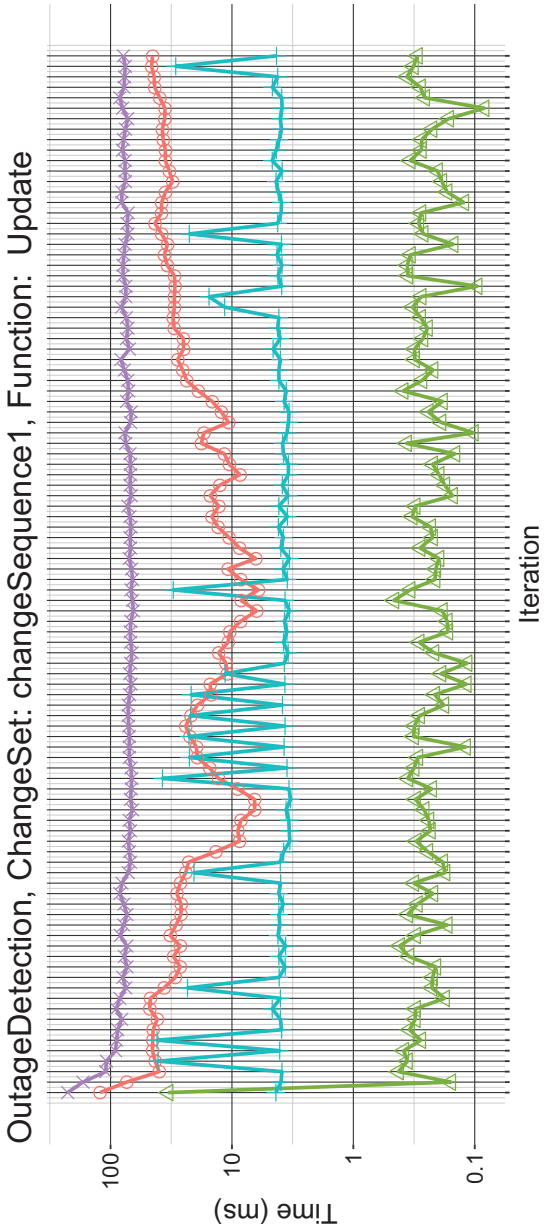


Figure 9.32.: Time to propagate change sequences of 20 elementary changes in the *Outage Detection* task

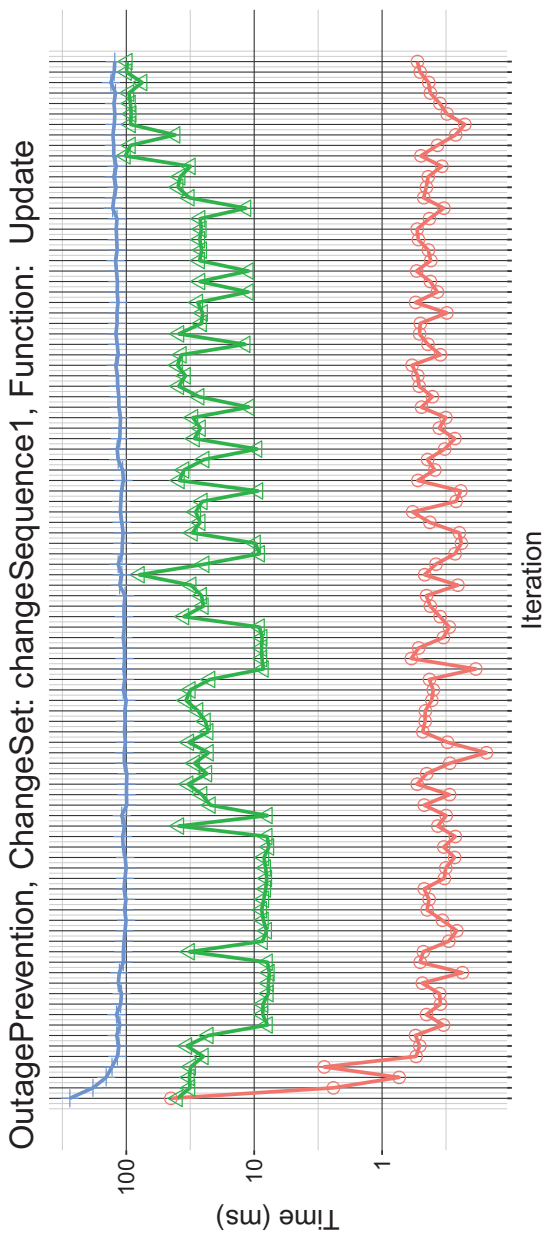
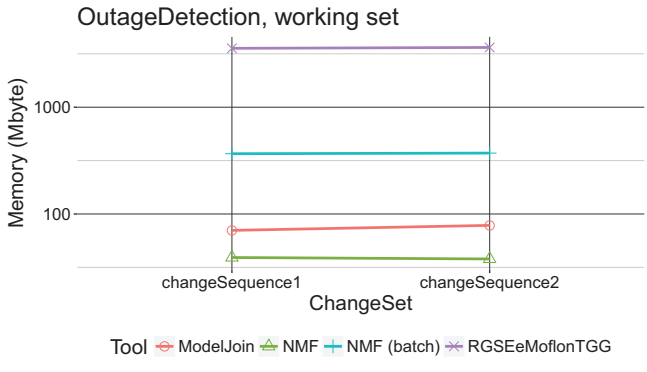
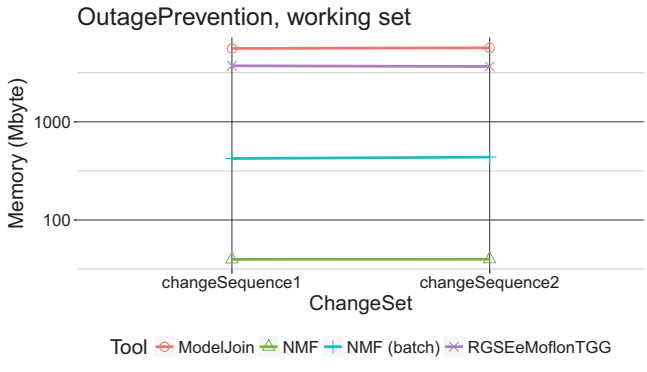


Figure 9.33.: Time to propagate change sequences of 20 elementary changes in the *Outage Prevention* task



(a) *Outage Detection* task



(b) *Outage Prevention* task

Figure 9.34.: Working set sizes solving the Smart Grid benchmark

At the TTC 2017, the NMF solution won the awards for the best overall solution and the best performance. The award for the best understandable solution went to the eMoflon solution.

9.8.5. Summary

The case study shows that model transformation using synchronization blocks can be used to join information from multiple model elements into a single model – a feature that only few model transformation languages

support. Perhaps as a consequence, there was only one submission to the corresponding TTC contest besides the NMF solution and hence the case study only offers limited insights with regard to a comparison to other tools. This solution in eMoflon did not manage to achieve an incremental execution.

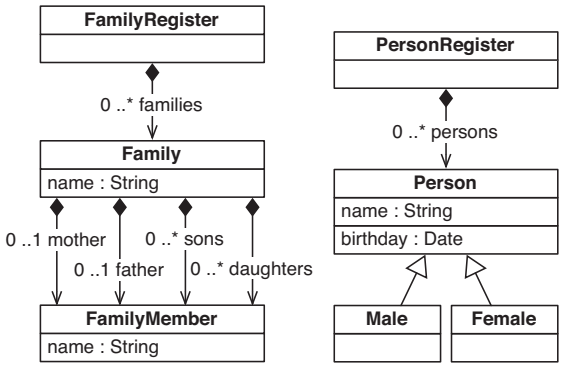
From the results, we see that an incremental execution using NMF is significantly faster than running the NMF solution in batch mode or running either of ModelJoin or eMoflon. In addition, the incremental solution also has the lowest memory consumption, indicating that although a memory overhead is required for incremental computation, the overall memory consumption may be lower because fewer models have to be created anew.

The fact that only two solutions were submitted to the Smart grid case for the TTC 2017 has the consequence that we can hardly make a statement on the understandability of the NMF solution in comparison with other tools.

9.9. Case Study: Bidirectional Transformation from Families to Persons

According to the general model theory of Stachowiak [185], models always have a purpose. However, in many practical applications, models shall be used for multiple purposes while existing metamodels should be reused, usually because valuable instrastructure is built on top of it. However, because the different purposes require different abstractions, the information concerning an entity is often split among multiple of these models, which makes a pure transformation approach infeasible. Instead, the models must be synchronized to make sure that they are consistent with regard to some correspondence rules.

The Families to Persons case at the Transformation Tool Contest (TTC) 2017 [7] demonstrates this problem in the scenario of a well known example model transformation, the Families to Persons transformation. Here, a structured model of family relations shall correspond to a flat model of persons where the information of a persons family is encoded only through that persons full name. The information contained in one model cannot be fully reconstructed using the other model. Nevertheless, there is a clear correspondence as there should be a family member for each person and vice versa.



(a) The Families metamodel **(b)** The Persons metamodel

Figure 9.35.: Source and target metamodel in the FamiliesToPersons case study [7]

At the TTC 2017, solutions to the Families to Persons case were submitted using NMF [93], UML-RSDS [134], SDMLib [222], FunnyQT [112], EVL+Strace [179] and Yage [62].

9.9.1. Benchmark Setup

In this section, we briefly introduce the Families to Persons benchmark. The Families to Persons transformation is a well known example transformation from the ATL website, already used in Section 9.6. However, in this case study, we are using a version extended by Anthony Anjorin, Thomas Buchmann and Bernhard Westfechtel, specifically designed to compare bidirectional tools [7].

The metamodels in this case study are depicted in Figures 9.35a and 9.35b. They represent similar information in different manner: Whereas the full name of a person is separated in its first name and surname in the *Families* model, this information is combined in the Person model in the format *"Surname, Firstname"*. Whereas the information on gender is explicitly modeled in the *Persons* metamodel, this information is implicitly encoded in the containment hierarchy in the *Families* metamodel. Lastly, the *Families* metamodel contains a family structure that is missing in the *Persons* model, meanwhile the birthday information is only present in the *Persons* metamodel.

Since no model can be reconstructed entirely from the other, the task for solutions is to create a bidirectional transformation between both models. Meanwhile the direction from the *Families* metamodel to the *Persons* metamodel is straight forward, the backwards transformation is indeterministic. Therefore, the behavior of the backward transformation has to be controlled by two flags `PREFER_CREATING_PARENT_TO_CHILD` and `PREFER_EXISTING_FAMILY_TO_NEW`. Although the naming conventions of these configurations suggest that they are constant, they actually may change during a synchronization.

The benchmark case study is equipped with a rich benchmark suite available online[105], where also reference implementations in eMoflon [138], BiGUL [126], Medini-QVT[106] and BXtend [31] are available.

Furthermore, the benchmark framework contains a set of test cases to compare bidirectional model transformation tools with regard to their expressiveness and correctness as well as a further set of test cases to compare solutions with respect to performance. The benchmark reports on the number of failures (unexpected test failures according to the specification of the tool) and limitations (expected failures according to the specification of the tool).

The scalability test cases work as follows:

Batch Forward: In this test case, a *Families* model with an increasing number of families is created. The benchmark then measures the time of solutions to create a corresponding *Persons* model.

Batch Backward: This test case is exactly symmetric to the *Batch Forward* test case. The benchmark creates a *Persons* model of increasing size and measures the time that tools need to create a corresponding *Families* model.

Incremental Forward: In this test case, the benchmark framework creates a *Families* model of an increasing size and let the solutions create a corresponding *Persons* model. Then, a new family member is inserted into the

[105] `https://github.com/eMoflon/benchmarx`
[106] `http://projects.ikv.de/qvt`, retrieved 27 Sep 2017

model and the benchmark measures how long solutions needs to propagate this added family member to the *Persons* model.

Incremental Backward: This test case is again exactly symmetric to the *Incremental Forward* test case.

9.9.2. Validation Goals

The goals of this case study are as follows:

Applicability While the application scenario of this case study is clearly synthetic, the case study offers to gain insights on model transformations with custom backwards transformation and an integration of configuration variables.

Performance As the goal of this case study is to compare the incremental performance of bidirectional model transformation tools, we are particularly interested in the results of the incremental scalability test cases *Incremental Forward* and *Incremental Backward*.

Understandability As for all TTC case studies, the case study offers to draw a comparison with respect to understandability to other tools.

Correctness Correctness is the main emphasis of the original benchmark by Anjorin and others. Correctness is indicated by a series of tests that solutions have to pass.

9.9.3. NMF Solution

The description of the NMF solution is based on the original solution as submitted to the TTC 2017 [93].

To solve the FamiliesToPersons case, we see two correspondences that need to be synchronized:

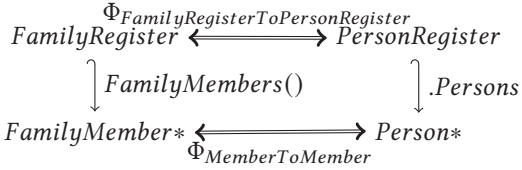


Figure 9.36.: Synchronization block to synchronize family members with person elements

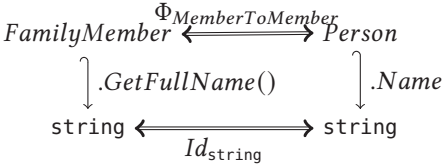


Figure 9.37.: Synchronization block to synchronize names

1. All family members contained in a family need to be synchronized with the people in the Persons model and
2. The full name of family members that consists of the name of the family and the name of the family member needs to be synchronized with the full name of the corresponding person.

Using synchronization blocks, these correspondences can be formulated in the diagrams of Figure 9.36 and Figure 9.37. The implementation in NMF Synchronizations is depicted in Listing 9.26.

While NMF is able to convert the simple member accesses for the persons into lenses, this does not hold for the helpers `FamilyMemberCollection` and `GetFullName` that we used in this implementation – these lenses are highly specific to the given scenario, indicated by the fact that their implementation depends on the flags `PREFER_CREATING_PARENT_TO_CHILD` and `PREFER_EXISTING_FAMILY_TO_NEW`. Therefore, we have to explicitly provide an implementation of Put for these two lenses.

For the `GetFullName`-method, the Put operation needs to be specified through an annotation. In addition, because NMF does not parse the contents of a method (only of lambda expressions), we need to specify an explicitly incrementalized version of the given helper method. To do this, we can reuse

```
public class FamilyRegisterToPersonRegister : SynchronizationRule<
     FamilyRegister, PersonRegister> {
  public override void DeclareSynchronization() {
    SynchronizeMany(SyncRule<MemberToMember>(),
                fam => new FamilyMemberCollection(fam),
                persons => persons.Persons);
  }
}
public class MemberToMember : SynchronizationRule<IFamilyMember, IPerson> {
  public override void DeclareSynchronization() {
    Synchronize(m => m.GetFullName(), p => p.Name);
  }
}
```

Listing 9.26: Implementation of main synchronization blocks

```
1  private static ObservingFunc<IFamilyMember, string> fullName =
2    new ObservingFunc<IFamilyMember, string>(m => m.Name == null ? null : ((
         IFamily)m.Parent).Name + ",␣" + m.Name);
3
4  [LensPut(typeof(Helpers), "SetFullName")]
5  [ObservableProxy(typeof(Helpers), "GetFullNameInc")]
6  public static string GetFullName(this IFamilyMember member) {
7    return fullName.Evaluate(member);
8  }
9  public static INotifyValue<string> GetFullNameInc(this IFamilyMember member)
       {
10   return fullName.Observe(member);
11 }
12 public static void SetFullName(this IFamilyMember member, string newName) {
13   ...
14 }
```

Listing 9.27: Implementation of the `GetFullName` lens

the implicit incrementalized lambda expression and also use that for the batch implementation to avoid code duplication. A sketched implementation is depicted in Listing 9.27.

In Listing 9.27, we create an `ObservingFunc` for the function to obtain a family members full name. This syntax allows NMF to get a model of the actual function and therefore, NMF Expressions is able to incrementalize it. In Lines 6–11, we use this object to represent the actual extension

```
1  private class FamilyMemberCollection : CustomCollection<IFamilyMember> {
2    public FamilyRegister Register { get; private set; }
3    public FamilyMemberCollection(FamilyRegister register)
4      : base(register.Families.SelectMany(fam => fam.Children.OfType<
           IFamilyMember>()))
5    { Register = register; }
6
7    public override void Add(IFamilyMember item) { ... }
8    public override bool Remove(IFamilyMember item) { ... }
9    public override void Clear() { ... }
10 }
```

Listing 9.28: Implementation of the `FamilyMemberCollection`

method and its incrementalization, which essentially forwards to the same `ObservingFunc` object. The incrementalization is connected with the original function through the annotation in line 5. Furthermore, we annotate the Put method also using an annotation in line 4 where we reference the `SetFullName` method in line 12–14. The return type `void` of this method makes it clear that this is a persistent lens. This method may contain arbitrary C# code and is called when NMF Synchronizations needs to write a value, for instance as a consequence of an update in the *Persons* model where the last name of a person changed.

In the case of `FamilyMemberCollection` which as the name implies is a collection, we only have to provide the query how the results of this collection are obtained and implement the methods `Add`, `Remove` and `Clear`. NMF Expressions is able to automatically incrementalize the query when this is necessary and uses the provided model manipulation methods in case a model element has to be added to the collection. A schematic implementation is depicted in Listing 9.28.

Again, the model manipulation methods `Add`, `Remove` and `Clear` may contain arbitrary C# code. Our implementation uses them to add the family member to the family register depending on the current configuration of the flags `PREFER_CREATING_PARENT_TO_CHILD` and `PREFER_EXISTING_FAMILY_TO_NEW`.

However, to add a family member to a family, the `Add` method has to know the family name of a person as well as its gender – information that is encoded using the containment hierarchy in the Families model and therefore

```
1  public class MemberToMale : SynchronizationRule<IFamilyMember, IMale> {
2    public override void DeclareSynchronization() {
3      MarkInstantiatingFor(SyncRule<MemberToMember>(),
4            leftPredicate: m => m.FatherInverse != null || m.SonsInverse != null)
                 ;
5    }
6    protected override IFamilyMember CreateLeftOutput(IMale input, ...) {
7      var member = base.CreateLeftOutput(input, candidates, context, out
           existing);
8      member.Extensions.Add(new TemporaryStereotype(member) {
9        IsMale = true,
10       LastName = input.Name.Substring(0, input.Name.IndexOf(','))
11     });
12     return member;
13   }
14 }
```

Listing 9.29: The `MemberToMale`-rule

unavailable before the element is added to a family. Therefore, we carry this information over from the corresponding element of the Person metamodel using a temporary stereotype: In NMF, all model elements are allowed to carry extensions. We use this to add an extension that specifies the last name and whether the given element is male. The stereotype is deleted as soon as a family member is added to a family.

Furthermore, the fact that different genders are modeled through different classes in the Persons model, the synchronization rule `MemberToMember` needs to be refined to allow NMF Synchronizations to decide whether to create a `Male` or `Female` output element. This can be done in NMF Synchronizations through an instantiating rule.

The implementation of both of these concepts is depicted in Listing 9.29.

In particular, lines 3 and 4 mark the synchronization rule `MemberToMale` as instantiating for the rule `MemberToMember` on the condition that the family member is either a father or a son of a family. Further, we override the creation of an output model element for the LHS by overriding the method `CreateLeftOutput`. This method calls the base implementation which simply uses the default constructor to create a new `FamilyMember` element. Then, it adds a case-specific extension called `TemporaryStereotype` that carries

the information on the gender (through the `IsMale` attribute) and the last name.

9.9.4. Results

Several solutions submitted to the TTC 2017 *Families to Persons* case, using the languages NMF, UML-RSDS [134], SDMLib [222], FunnyQT [112], EVL+Strace [179] and Yage [62]. In addition, the case provided reference solutions in eMoflon, MediniQVT and BXtend.

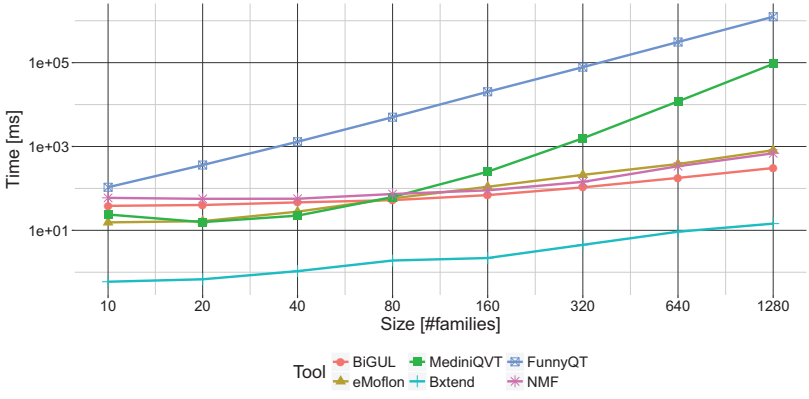
However, not all solutions have been integrated into the benchmark framework. Therefore, we only present performance results for BiGUL, eMoflon, MediniQVT, BXtend, FunnyQT and NMF.

The performance results for the scalability test cases of the benchmark framework that comes along with with case resources [7] are depicted in Figure 9.38. The graphs show the time for the benchmark solutions against the size of the model in terms of the number of families. Both axes are logarithmic.

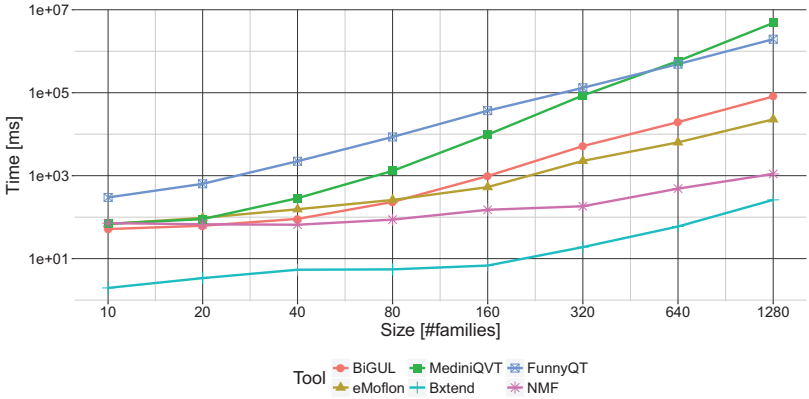
The results for the batch scenarios *Batch Forward* and *Batch Backward* depicted in Figures 9.38a and 9.38b show that the NMF solution is among the faster solutions, slightly faster than BiGUL and eMoflon and significantly faster than MediniQVT and FunnyQT. The solution is slower than BXtend. We think that this is because the NMF solution always runs in an incremental model and creates DDGs even though they are not used in the batch scenario.

The results for the incremental scenarios *Incremental Forward* and *Incremental Backward* depicted in Figures 9.39a and 9.39b show that in these scenarios, the DDGs created by NMF Synchronizations are indeed useful as they make the solution faster than any of the other solutions by multiple orders of magnitude.

However, the model sizes depicted in Figure 9.38 are still relatively small as otherwise, running the benchmark would take too much time for the slower benchmark solutions. Therefore, we repeated the incremental scenarios for the three tools that were fastest in Figure 9.38, eMoflon, BXtend and NMF. This time, we scaled up the model sizes up to more than 40,000 families in order to obtain information on the runtime for larger model sizes.



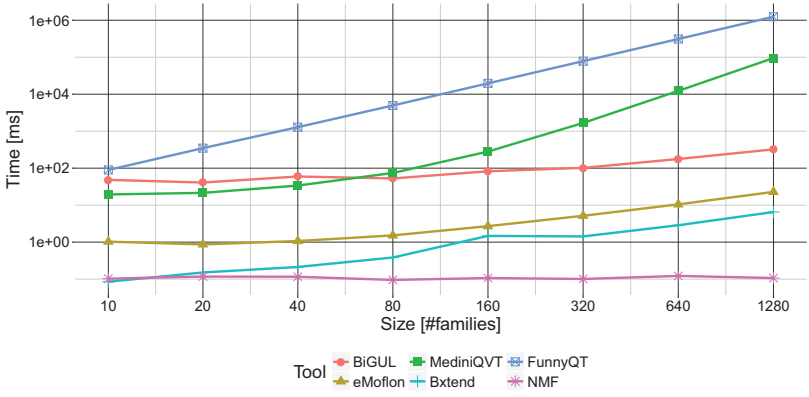
(a) Batch Forward



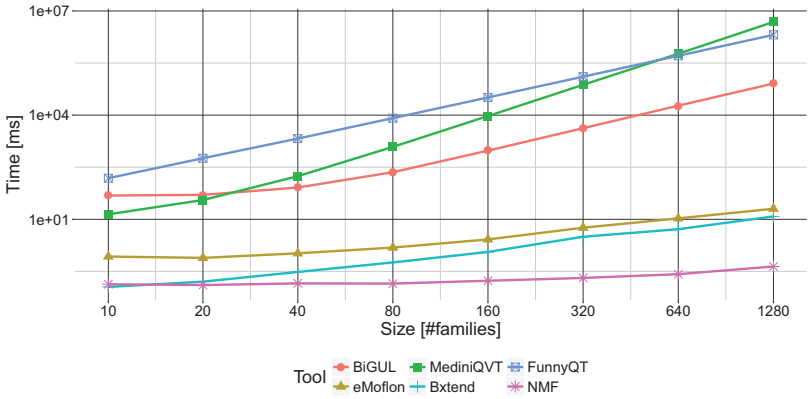
(b) Batch Backward

Figure 9.38.: Performance results for the scalability tests in the *FamiliesToPersons* case study

In the results for the *Incremental Forward* scenario, we can see that the resulting curve for the NMF solution is flat meanwhile the curve for eMoflon and BXtend is not. This indicates that unlike the other two, the NMF solution is the only fully incremental solution as the time to add a new family member does not depend on the size of the model. Furthermore, even for the smallest model with only 10 families, the incremental update is faster than recomputing the *Persons* model from scratch. For the largest size that



(a) Incremental Forward



(b) Incremental Backward

Figure 9.39.: Performance results for the scalability tests in the *FamiliesToPersons* case study (cont.)

includes a total number of more than 900,000 model elements in *Families* and *Persons* model together[107], this leads to a speedup of more than three orders of magnitude.

[107] The number is computed by the benchmark framework and also includes the number of edges between model elements.

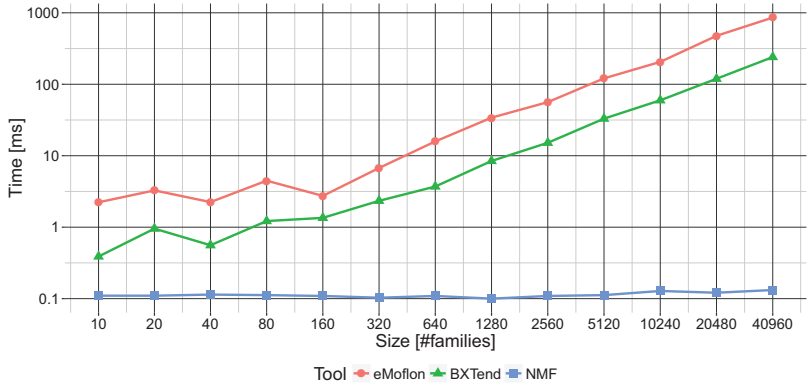


Figure 9.40.: Performance results for the scalability incremental forward tests in the *FamiliesToPersons* case study

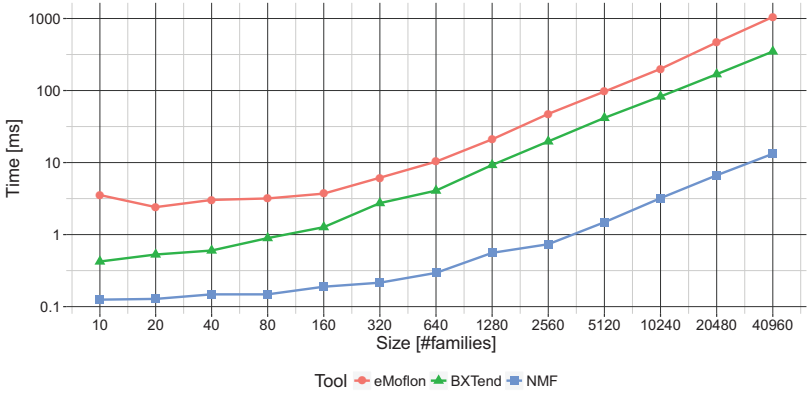


Figure 9.41.: Performance results for the scalability incremental backward tests in the *FamiliestoPersons* case study

As a confirmation of the results achieved in Section 9.5.3), the eMoflon solution does not appear to be fully incremental. It is even slower than the BXtend solution even though the latter is admittedly not incremental. The scalability seems exactly the same since the curves are parallel for the larger models, indicating a constant factor between these two solutions.

Tool	Overall Quality	Understandability	Confidence	Presentation
EVL+Strace	4.17	4	4.17	4.5
FunnyQT	3.86	3	3.86	3.83
NMF	3.38	3.13	4.13	3.33
SDMLib	4	4	4.17	4.5
UML-RSDS	3.5	4.17	4.5	3.6
Yage	2.17	2.17	4	2.5

Table 9.6.: Responses from the open peer reviews and from the live evaluation at the TTC 2017 for the *FamiliesToPersons* case study. The questions asked to evaluate the presented properties in a scale from 1 to 5.

The results for the *Incremental Backward* benchmark in the opposite direction are depicted in Figure 9.41. Unlike the incremental forward where the scalability tests inserts a new family member and the propagation simply adds the corresponding person which can be done in amortized $O(1)$, the incremental backward test has to look for a suitable family, which implies a $\Omega(n)$ complexity. Therefore, the speedups in this case do not grow equally with the size. Instead, a saturation happens at a speedup of slightly more than one order of magnitude.

For an evaluation of the understandability, we rely on the questionnaire responses collected at the TTC workshop. At the workshop, 29 responses for the evaluation questionnaire that was given to attendees were collected where each response evaluates one solution by one attendee. The responses are publicly available[108]. The results for the average evaluation of the solutions is depicted in Table 9.6.

[108] Open Peer Reviews: `https://docs.google.com/spreadsheets/d/1w3VBZJGe9nhwcrHn_RDn5YunR9ZNicJJ0t77WwlzaPc/edit?usp=sharing`
Live evaluation: `https://docs.google.com/spreadsheets/d/1hOidjd_WKVT-faNHXPVpToyRQIzxkdfrIMrFqgDtuY8/edit?usp=sharing`

```
1 context Families2Persons!FamilyMemberSourceEnd{
2   guard: not self.isRemoved() and not self.refFamilyMember2Persons.
        endTypeIsRemoved()
3   constraint nameIsModified{
4     check: not self.nameIsModified()
5     message: 'name of '+self+' is modified'
6     fix{
7       title:'Propagate the modification'
8       do{ self.namePropagates(); }
9     }}}
```

Listing 9.30: Repairing an inconsistent names in EVL+Strace

The results indicate that the NMF solution was less understandable than the solutions in EVL, SDMLib or UML-RSDS. Based on comments from the open peer reviews and from the presentation, this is mainly due to a lack of understanding of the theory of synchronization blocks. We think that this is due to the fact that there was little information on the theory of synchronization blocks available because at the time of the TTC 2017, there was no publication available on this theory to explain the concepts. Furthermore, the NMF solution, like the FunnyQT solution, could not be presented in person but only with a recorded video presentation.

On the other hand, the NMF solution was rated as being slightly more understandable than the FunnyQT solution that also provides declarative bidirectionality and quite more understandable than the Yage solution.

Interestingly, the EVL+Strace solution had a very high score for understandability and even got the audience award for the best overall quality. This is interesting mainly because this solution is neither very declarative nor bidirectional but follows a different paradigm: It keeps a trace model between the synchronized model and detects inconsistencies within the trace model.

As an example, we depicted the EVL+Strace solution to repair name inconsistencies of a family member. The constraint specifies that if the name of a person is modified (computed in the operation `nameIsModified`), then the operation `namePropagates` should be called to fix the constraint violation. Such a declaration is necessary for every possible change that could cause an inconsistency. In the example, this is necessary for the a change of the

first name of a family member, the change of the name of the family and a change of the name of the person.

Using synchronization blocks, the same semantics of three constraints is expressed in a single synchronization block with an additional lens for `GetFullName`. As a result, the EVL+Strace solution is much less concise, using 2,077 lines of code as opposed to 198 lines of code for the NMF solution.

A declarative specification always comes at the cost that developers lose their overview on the control flow. In particular, a developer quickly has an intuition how the system processes a specification such as the one in Listing 9.30 because analysis of inconsistencies and their repair operations are strictly separated. In particular, the developer has a full control on the model manipulation that is performed in the synchronization process.

This is not true for synchronization blocks specified using NMF Synchronizations: Here, the analysis of inconsistencies and the operations performed as repair operations are intertwined as both are deduced from lenses. Further, if developers are not used to the declarative support of C#, they may simply see the function and not understand that NMF Synchronizations lifts this specification to a lens and also runs an incrementalization system over it. Therefore, developers may assume that the code is only used to check consistency constraints, but not to enforce them.

The last problem is also more general: the C# language is not very common to the general TTC audience and we think that many people still compare C# closely with Java. However, this impression is wrong, at least if the C# language is used declaratively. We suspect that the understandability is easier for developers more familiar with the C# language. For users of LINQ, the idea that C# code is not executed but analyzed and then something else is executed based on a model of the code is not new: For LINQ, the code model is transformed to a database query. NMF Synchronizations uses the model to construct a synchronization block. However, if a developer does not know the declarative syntax capabilities of C#, then the NMF solution must puzzle him.

Solutions using external languages such as for example UML-RSDS do not have this problem. Similar to NMF Synchronizations, the UML-RSDS solution works by letting users specify consistency constraints, in UML-RSDS as post-conditions of a transformation but does not support incremental change propagation. However, because the language is external, developers

expect that the constraint is not only tested but that the system can deduct an execution semantics from it. For C# code, the expectation is usually that it is only executed, not as much analyzed.

There was no evaluation of the solution in eMoflon or BXtend because these solutions counted as reference solutions.

9.9.5. Summary

Even though the use case of this case study is rather artificial, the case study allows a detailed comparison, especially with regard to performance, with state-of-the-art BX tools. The comparison shows that significant advances in the incremental execution of model transformations could be achieved, meanwhile the solution allows a very flexible customization with regard to resolving indeterminism in the case description through configuration entries.

The evaluation of the understandability showed that our solution was less understandable than solutions in EVL+Strace, SDMLib or UML-RSDS. Comparing our solution with the EVL+Strace solution that received the best understandability assessments, we think that this is mainly due to an unawareness of the audience that C# allows to specify functions declaratively.

Developers have to know the details of the C# language and also details of NMF Synchronizations in order to understand the NMF solution. This is not the case for EVL+Strace where the solution is rather intuitive. In particular, we think that the implementation of NMF Synchronizations as an internal DSL has pointed the audience towards a false assumption that the functions specified in the synchronization blocks can only be executed while in truth, they are lifted to in-model lenses and incrementalized.

Therefore, the case study shows that developers that shall comprehend model transformations using NMF Synchronizations should be trained both in its underlying theoretical concepts, but also very importantly the declarative usage of the C# language that NMF Synchronizations is built upon.

9.10. Case Study: Modeling a Bike shop

Many existing approaches for Deep Modeling require entirely new concepts such as potencies [15] that are often incompatible with existing modeling standards common in model-driven engineering such as Ecore. Therefore, these languages have to create their own languages to process these models auch as DeepATL [13].

Because many of the existing approaches define their own concepts, it is hard to make a comparison. Therefore, the Bicycle Challenge in the scope of the MULTI 2017 workshop was suggested to foster this comparison and to reach a common understanding what the general concepts of these approaches are.

In this section, we take this challenge as a case study to see how the presented approaches to simplify metamodels as presented in Part IV can be applied to this modeling challenge.

9.10.1. Task

For self-containment of the thesis, we replicate the case description from the MULTI website[109] here:

> A configuration is a physical artefact that is composed of components. A component may be composed of other components or of basic parts. There is a difference between the type of a component and its instances. A component has a weight. A bicycle is built of components like frame, a handle bar, two wheels … A bicycle component is a component. A frame, a fork, a wheel, etc. are bicycle components. Frames and forks exist in various colors. Every frame has a unique serial number. Front wheel and rear wheel must have the same size. Each bicycle has a purchase price and a sales price. There are different types of bicycles for different purposes such as race, mountains, city .. A mountain bike or a city bike may have a suspension. A mountain bike make have a rear suspension. That is not the

[109] `https://www.wi-inf.uni-duisburg-essen.de/MULTI2017/#challenge`, retrieved 4 Sep 2017

case for city bikes. A racing fork does not have a suspension. It does not have a mud mount either. A racing bike is not suited for tough terrains. A racing bike is suited for races. It can be used in cities, too. Racing frames are specified by top tube, down tube, and seat tube length. A racing bike can be certified by the UCI. A racing frame is made of steel, aluminum, or carbon. A pro race bike is certified by the UCI. A pro race frame is made of aluminum or carbon. A pro racing bike has a minimum weight of 5200 gr. A carbon frame type allows for carbon or aluminum wheel types only. „Challenger A2-XL" is a pro racer for tall cyclists. The regular sales price is 4999.00. Some exemplars are sold for a lower price. It is equipped with a Rocket-A1-XL pro race frame. The Rocket-A1-XL has a weight of 920.0 gr. A sales manager may be interested in the average sales price of all examplars of a certain model. He may also be interested in the average sales price of all mountain bikes, all racing bikes etc.

To models of this domain, the following requirements were posed:

1. Knowledge about the domain, which may include particular aspects, should be represented at the highest level possible.
2. It should be possible to use the model (or parts of it) as a foundation for a software system that is suited for a wide range of general bicycle stores. At the same time, it should allow for this software to be refined into more specific systems like one for a specialized dealer of professional racing bikes.
3. It should be possible to define associations between elements on different levels. Alternatively, it can be shown that cross-levels associations are not required.
4. As a consequence of req. 3, it should be possible to specify cross-level constraints.
5. There should be mechanisms that protect the integrity of lower levels of the model from changes that occur on higher levels.
6. There should be mechanisms to preserve the model semantics and foster the synchronization of MLM-based models with code.

Solutions were demanded to fulfill at least three requirements. Lack of information or ambiguities in the description should be identified and removed by making explicit assumptions.

9.10.2. Case Analysis

From our perspective, there are very few instantiation relations between different model elements of the bicycle shop: As an example, a suspension fork is still a component, regardless of the possibility that there may be classes such as `BicycleComponent` or `Fork` in between in the inheritance hierarchy.

For concrete components such as the Rocket-A1-XL racing frame, the question whether these should inherit or instantiate `RacingFrame` is not as easy to answer: A bike shop may have several of such racing frames in stock and a question is whether they should all be modeled as instances of a Rocket-A1-XL racing frame model element. The answer that we think is appropriate here is no because there are no important properties that a frame has because it is a Rocket-A1-XL racing frame but many properties are shared. Therefore, it is better to model the Rocket-A1-XL racing frame as an *instance* of `RacingFrame` as this makes it easier to specify shared properties.

A very similar question arises for bikes themselves: The bike store has many Challenger A2-XL racing bikes in stock but should they be instances of Challenger A2-XL racing bikes? In contrast to the racing frames, the bikes have individual properties, namely the actual sales price. Even more, is it necessary that we model not only instances of bikes but also the instances of frames, forks and other components? If we do, then each instance of a Challenger A2-XL racing bike has to be connected with an instance of a Rocket-A1-XL racing frame. This means an additional overhead because the modeler has to model each and every part of each bike sold. It also has some advantages as it allows to put labels on each instance, though we hardly think that this is necessary. After all, we assume that a bike store would only sell bikes as a whole or components separately. A bike store would not disassemble a bike and sell or exchange its components. Therefore, modeling each component of a bike as a model element would unnecessarily bloat the instance models, increase the effort to create instances and cause a lot of complexity in the metamodel which is something that we try to avoid.

Further, we think that the actual sales price is something that is not at all specific to bikes, but rather is a general concept of items in stock that may be discounted and in that case are sold to a different price than the recommended sales price. Thus, we think that it is more practical to insert a new class of `StockItem` that refers to a configuration of components that is sold, but perhaps for a different price. Because the domain description only foresees a price for bikes rather than for components, we named the class `BikeInStock` and reference the model of the bike in a dedicated reference.

The relationship of a bike in stock to the model of a bike can in fact be described in a instantiation relationship. However, the properties of a bike in stock are only marginally influenced by the specific bike instance – from the domain description, only the sales price default value is different. This is something that can be conveniently handled using classic two-level metamodeling. For us, the characteristics of deep modeling are rather cases where the properties of instances are determined by properties of the model element determing their type. For example, instances of components must be connected to other instances of components according to the required interfaces of the respective component, in addition to some general properties. For the bicycle challenge, the only such case is that the default sales price of bicycles is influenced by the type of bicycle. Therefore, we do not see a reason to apply deep modeling concepts here.

9.10.3. Validation Goals

Unfortunately, the NMF solution was not submitted to the MULTI 2017 workshop due to logistics reasons and therefore, we do not have empirical data to reason on the understandability of our approach. As a consequence, we only rely on the reader to get his own impression and only use this case study to validate the applicability of our refinements and structural decomposition approach. Furthermore, we do compare our solution to the only submitted solution to the modeling challenge by Macías et al. using MultEcore [142] in terms of suitability for analysis purposes.

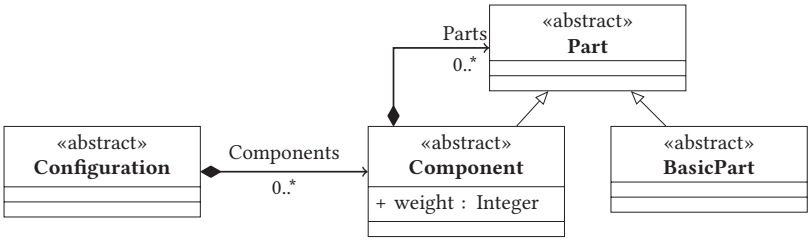


Figure 9.42.: Configurations and components in a bike shop

9.10.4. NMF Solution

In this section, we develop how to model the bike shop according to the domain description presented in the challenge description. The model, the code generated for it and a very simple client application are publicly available on GitHub[110].

Because the case description does not contain instantiation relationships between model elements (cf. Section 9.10.2), we do not use the deep modeling capabilities of NMF (cf. Chapter 8) for this case. Rather, we only use the ability of NMF to implement refinements and structural decomposition.

The very basic classes in the domain are configurations and components. They can be modeled as one would expect to model them in a classic two-level metamodel such as depicted in Figure 9.42. A `Configuration` is an abstract class that consists of components. A `Component` in turn may consist of other components or basic parts, which is why we added a common base class for these two classes.

Much more interesting from a modeling perspective is the fact that bicycles are a special form of configurations that consist of bicycle components such as a forks, frames, wheels or handle bars. More precisely, a bicycle consists of exactly one frame, exactly one fork, exactly two wheels and exactly one handle bar. Moreover, these components *are* the components that the bicycle consists of when viewed as a configuration.

[110] `https://github.com/georghinkel/BikeShop`

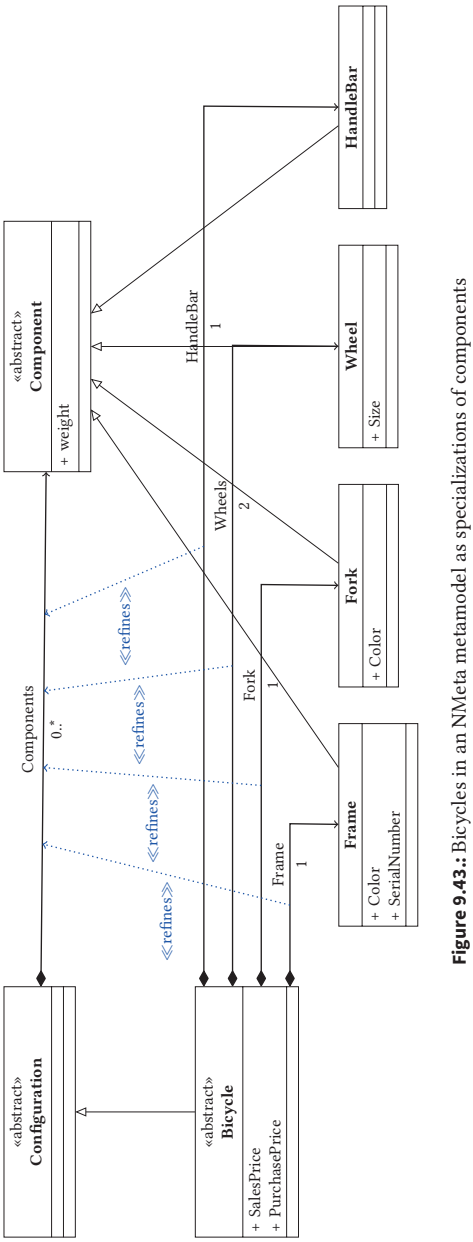


Figure 9.43.: Bicycles in an NMeta metamodel as specializations of components

Through structural decomposition, NMeta has a dedicated built-in support for this kind of relationships. The suggested model is depicted in Figure 9.43 where blue refinement arrows denote a structural decomposition. In particular, we can specify that several references of `Bicycle` to all of its components taken together refine the *Parts* reference of `Configuration`. Thus, the components of a `Bicycle` are exactly its frame, its fork, its handle bar and the wheels.

As a consequence of this assignment, the default implementation class of `Bicycle` does not inherit from the default implementation class of `Configuration` but implements the members required for the generated interface for `Configuration` directly. Whereas the default implementation class contains a field to store the parts of the configuration, the class implementing `Bicycle` assembles this collection on the fly, based on the referenced elements for each of the references that decompose the `Parts` reference in the scope of a `Bicycle`.

This interface implementation is also private such that the `Parts` reference of `Configuration` is no longer visible in the generated model API for `Bicycle`. The user shall not see this reference any more because the data of this reference is provided through the refining reference: Instead of specifying that the frame (as a component) is a part of the bike, users have to assign the frame as an instance of `Frame` to the corresponding reference directly.

We did not add a dedicated common base class `BicycleComponent` as we think that the current model is much more precise. Furthermore, we do not realize the constraint that both wheels must have the same size. Such a constraint would be easy to specify in OCL based on the `Bicycle` class, but as mentioned earlier, NMF has no support for such constraints.

Refinements can be stacked, i.e. a reference that refines another reference can be refined itself in the scope of another class. This is used for example in for the class `MountainBike`. According to the domain description, the fork of a `MountainBike` *must* be a `SuspensionFork`. Therefore, we redefine the `Fork` reference for `MountainBike` such that it is typed with a `SuspensionFork`. This is depicted in Figure 9.44.

Furthermore, mountain bikes consist of an additional component, a rear suspension. However, we already decomposed the `Parts` reference into the frame, the fork, the handle bar and the wheels. Fortunately, this is not a problem because a decomposition is always scoped. This means, the

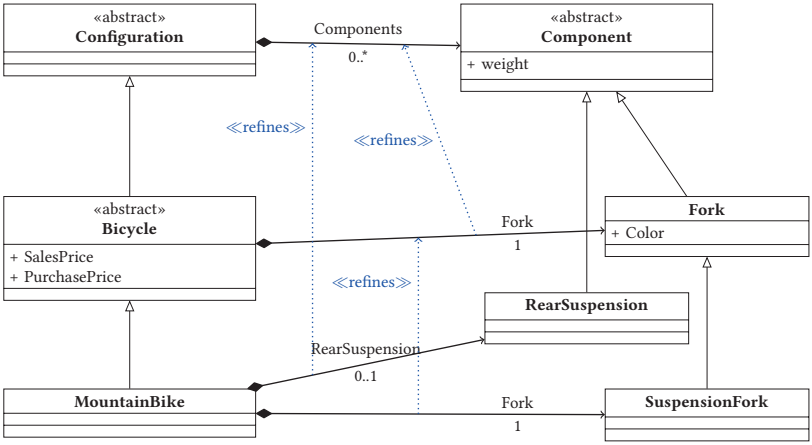


Figure 9.44.: Stacked refinements for MountainBikes

decomposition holds only as long as no other references take part in the decompositon. That means, we can extend the decomposition of the parts of a mountain bike by adding a rear suspension. This is also depicted in Figure 9.44.

Next, we need to model different types of bikes. Besides `MountainBike`, there are further classes for `CityBike`, `RaceBike` and `ProRaceBike` where `ProRaceBike` inherits from `RaceBike`. However, meanwhile a certification is optional for a `RaceBike` in general, it is mandatory for a `ProRaceBike`.

Using refinements, this relationship can be implemented by refining the `Certified` attribute of `RaceBike` with a constant attribute as depicted in Figure 9.45. This is implemented using a `AttributeConstraint` element and denoted in Figure 9.45 as an equation in the attributes compartment. Similar to other refinements, this assignment has the consequence that the generated default implementation type for `ProRaceBike` does not inherit from the default implementation class for `RaceBike` but implements its interface directly. In particular, the getter for the `Certified` attribute simply always returns true.

Similarly, the purpose of bikes is available as an attribute in the `Bicycle` class but refined in the subtypes with a constant attribute. Note that although a `RaceBike` in general is allowed to be used for cities and races, a `ProRaceBike`

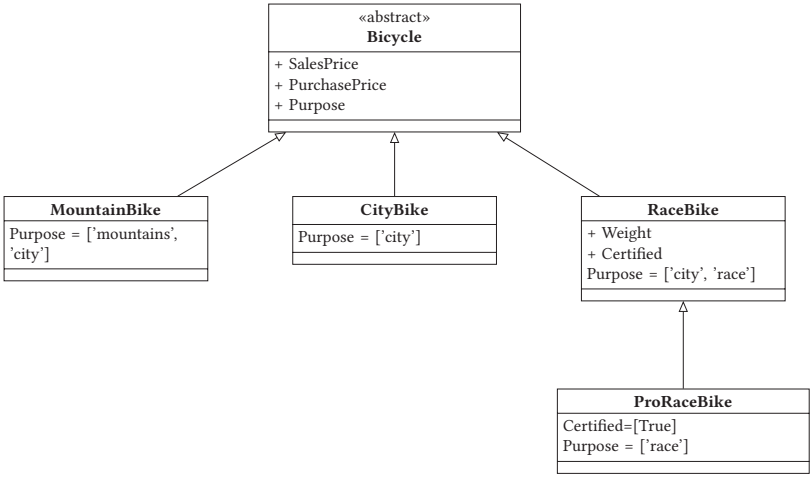


Figure 9.45.: Inheritance hierarchy of bicycle types

can only be used for races, as refinements by constant attributes or references may be overridden.

For racing frames, we would model the material as a simple enumeration because there is no additional information attached to these materials. However, NMeta does not allow to model an inheritance hierarchy for value types such as enumerations. Therefore, one would need a constraint to specify that the material of a `ProRaceFrame` must be either aluminum or carbon but such constraints are not (yet) supported in NMF. A similar statement hold for the minimum weight of 5200gr for pro racing bikes.

Instances of the class `Bicycle` represent a model of bicycles rather than a specific physical bike. To account for a different sales price of some exemplars, we need to model these exemplars also as model elements. However,

The code generator in NMF can use this model to generate classes to represent instances of this model in memory. These classes have an API such they are easy to use. For example, the code depicted in Listing 9.31 can be used to create an instance model with the example pro racer bike *Challenger A2-XL* model equipped with a *Rocket-A1-XL* pro racer frame and six exemplars of this bike where one has a discounted price.

```
var challengera2xl = new ProRaceBike
{
  RaceFrame = new ProRaceFrame
  {
    Name = "Rocket-A1-XL",
    Weight = 920.0
  },
  SalesPrice = 4999.0,
  Name = "Challenger_A2-XL"
};

var stock = new ObservableList<IBikeInStock>
{
  new BikeInStock { Model = challengera2xl },
  new BikeInStock { Model = challengera2xl },
  new BikeInStock { Model = challengera2xl },
  new BikeInStock { Model = challengera2xl },
  new BikeInStock { Model = challengera2xl },
  new BikeInStock { Model = challengera2xl, DiscountedPrice = 3999.0 }
};
```

Listing 9.31: Creating instances using the generated model API

```
var averagePrice =
  (from bike in stock
   where bike.Model is IRaceBike
   select bike.DiscountedPrice.HasValue ? bike.DiscountedPrice.Value
     : bike.Model.SalesPrice).Average();
```

Listing 9.32: Querying the average sales price for race bikes

Based on these models, one can use the C# query syntax to perform analysis such as computing the average sales price of all race bikes in stock such as in Listing 9.32. Because the discounted price is optional in the metamodel, NMF generates a nullable type for the respective property whereas the sales price is mandatory and therefore a `double` property is generated.

Also, note that the type filter uses the generated interface `IRaceBike`. This is required because the default implementation type `ProRaceBike` of a pro race bike does not inherit from the implementation type `RaceBike` in order to avoid inheriting the `Certified` property – the model API does not offer to set the `Certified` property of a `ProRaceBike` because a pro race bike is always

certified: The property is implemented in private and thus only accessible via the interface `IRaceBike`. Its getter simply returns true and the setter throws an exception if the value passed in is not true.

Akehurst and others have shown that modern C# is as declarative as OCL [5]. Indeed, NMF can support the incremental execution of such queries: With very small changes to the code and an additional import statement, the average sales price query from Listing 9.32 can be executed incrementally. This is depicted in Listing 9.33.

```
1 var averagePriceInc = Observable.Expression(() =>
2   (from bike in stock
3    where bike.Model is IRaceBike
4    select bike.DiscountedPrice.HasValue ? bike.DiscountedPrice.Value
5      : bike.Model.SalesPrice).Average());
```

Listing 9.33: Querying the average sales price for race bikes incrementally

Using a DDG that tracks model change events for all affected properties such as the model of a bike, the discounted price or the sales price of a model, we obtain updates when the average sales price changes.

9.10.5. Results

Unfortunately, the MultEcore solution by Macías and others [142] was the only solution submitted and accepted at the MULTI 2017 workshop. However, this solution is also very interesting because MultEcore [141] that was created by the same authors follows very similar goals: To combine the modeling expressiveness of Deep Modeling approaches with the mature tooling of existing two-level modeling tools. Notably, the approach of MultEcore is also very different to the approach of NMF. While in MultEcore the object and the class facet of a class are clearly separated (even in two different physical files, though they are treated as one model).

The difference becomes obvious when comparing the amount of used modeling levels. While we argued in Section 9.10.2 why the NMF solution actually does not make use of Deep Modeling features, the MultEcore solution is indeed divided into six modeling levels: At level 0, the authors see Ecore as the underlying meta-metamodel. Level 1 is the superstructure in components and parts (the solution does not make a difference between components and

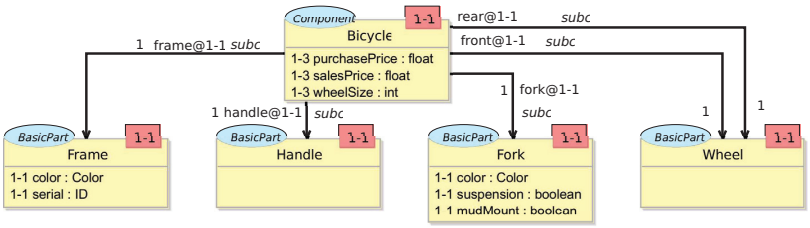


Figure 9.46.: Level 2: Bicycle in the MultEcore solution of the MULTI 2017 modeling challenge [142]

configurations). Level 2 contains Bicycles. Level 3 contains Racing Bicycles, for instance. Level 4 contains Pro Racing Bicycles and lastly, level 5 contains the example bike *Challenger A2-XL*.

As an example, the level 2 of this solution is depicted in Figure 9.46 where the class of objects is denoted in a blue oval. The red boxes depict potencies. Similarly, attributes are preceded with a potency. The potency of a class means how many levels deeper the class may be instantiated.

The diagram in Figure 9.46 matches the class diagram shown in Figure 9.43 very closely. The difference is only that what is implemented as a structural decomposition in NMF is implemented as a reference instantiation in MultEcore. Likewise, the information that a bicycle is a special type of configuration (or a special kind of component in the MultEcore solution) is represented as an inheritance relation in Figure 9.43 but as an instantiation relation in Figure 9.46.

This raises the question of what the impact of these differences are.

To answer this question, we have to go one level deeper. Therefore, we depicted the model fragment of the MultEcore solution regarding racing bikes in Figure 9.47. This level shows the assembly of racing bikes in the MultEcore solution. Similar to the level above, inheritance in the NMF solution is replaced by instantiation in the MultEcore solution.

In the Deep Modeling approach presented in Chapter 8, the most important characteristic of instantiation relationships is that instantiation is a non-transitive 'is-a'-relationship. Applied to the example in Figures 9.46 and 9.47, this means that a racing bike is not a component. One level deeper, this

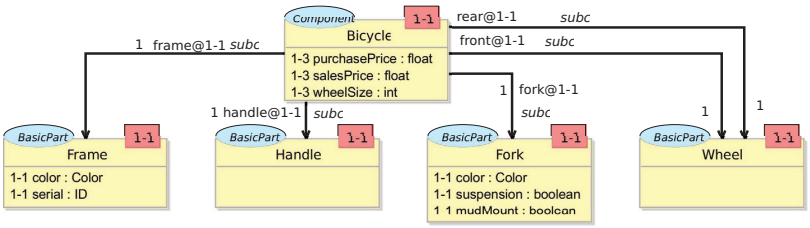


Figure 9.47.: Level 3: Racing Bicycle in the MultEcore solution of the MULTI 2017 modeling challenge [142]

similarly implies that a pro racing bike is not a bicycle – a statement that should be rejected from the domain perspective. Therefore, the usage of instantiation in MultEcore breaks the non-transitivity we expect from an instantiation relationship. However, the underlying concept of potencies does rely on this non-transitivity.

As a consequence, we have that for example the color of all racing frames must be the same, because the `color` attribute of racing frames is 1-1, i.e. the color must be specified exactly one level below. However, some instances of `Frame` are actually frames meanwhile other instances are conceptual elements such as the concept of a racing frame that suddenly must have a color set. This problem can be mitigated by making the potency a bit more flexible. As an example, this is done in the `Bicycle` class in Figure 9.46 where for example the `purchasePrice` has a potency 1-3, allowing that the purchase price does not have to be specified at the level of a `RacingBike` but multiple instances of `RacingBike` may have different purchase prices.

However, this workaround has multiple disadvantages. At first, it means that bikes do not necessarily have to have a purchase price as theoretically, the purchase price could be delegated to instances of instances of `Bicycle`, whatever that could be, semantically. But even if we take this flaw into account, we still have the problem that the required potency of attributes in `Bicycle` depends on the height of the hierarchy. In particular, if we found a further specialization of `ProRacingBike`, then would would have to change the potency of attributes in `Bicycle`. In other words, we have to foresee the classification scheme of bikes in the `Bicycle` class.

Furthermore, the usage of non-transitive relationships causes problems when it comes to model analysis: It is not possible to treat a racing bike as an instance of `Bicycle` because it is an instance of an instance, namely `RacingBike`. Therefore, when for example the average sales price has to be computed such as suggested in the original case description, then the system has to break the non-transitivity and collect all derivatives of `Bicycle` - whether these are instances of `Bicycle`, instances of instances of `Bicycle` or even instances of instances of instances, such as the *Challenger-A1-XL* pro race bike.

In the NMF solution, these problem are mitigated because there, both `Bicycle`, `RacingBike` and `ProRacingBike` are on the same modeling level, connected through inheritance relations that reflect the transitive nature of their connection. Therefore, it is indeed possible to specify an analysis very convenient using standard C# such as shown in Listings 9.32 and 9.33.

9.10.6. Summary

The Bicycle Challenge was created to compare existing multi-level or Deep Modeling tools. However, at least with NMF, we do not see a reason to use the Deep Modeling features of NMF for that case and rather solved the Bicycle model using two-level modeling, though with an additional feature of reference refinements and structural decomposition. Both of these concepts proved very useful to model the Bicycle Challenge with NMF as they help to model concepts at different levels of detail. We compared the NMF solution with a solution in MultEcore [142] that used multiple levels, but discovered severe problems of this usage that our solution is able to mitigate.

Besides the current limitations of NMF with regard to the lack of constraint support and a lack of a user interface, we think that NMF and its meta-metamodel NMeta are highly suitable to model the Bicycle Challenge. Because the meta-metamodel only introduces small extensions to classic EMOF implementations such as Ecore, we think that the metamodel is very understandable for metamodelers. Furthermore, the compatibility of the generated API with object-oriented concepts means that the models are easy to process using the generated model API.

In particular, NMF is able to execute analyses based on such a model incrementally, based on the pure batch specification.

9.11. Case Study: Incremental Analyses for Deep Architecture Description Models

In this last case study, we want to evaluate the performance of incremental model analyses in the presence of Deep Modeling when the analysis is not made obsolete by the Deep Modeling extensions. In particular, we want to compare the speedups that can be achieved through incrementalization with the speedup that can be achieved using the equivalent analysis for the two-level model. To do this, we created a small benchmark to measure the analyses presented in Section 8.4.

9.11.1. Validation Goals

The validation goals of this case study are thus as follows:

Performance In this case study, we are particularly interested in two things: The speedup that can be achieved using implicitly incremental analyses in the presence of Deep Modeling and to which that speedup is different to a two-level analysis. The important metric that we are looking at is again the average response time from a model change to an updated analysis result.

Correctness To check the correctness, we essentially compare the number of result elements for the analyses as presented in Section 8.4. These always have to match between incremental and non-incremental versions of the analysis both in their two-level or Deep Modeling versions.

9.11.2. Benchmark Setup

To evaluate the three versions of the analysis for performance, the two-level version in Listing 8.3, the Deep Modeling version in Listing 8.2 and the alternative in Listing 8.4, we created a small benchmark. This benchmark creates a model of the *MediaStore* example application used in Section 8.3.2 that is multiplied n times where n is an exponentially growing number from 1 to 64. Note that this replication of the model only affects the system assembly and the deployment, we are not replicating the repository components.

The benchmark consists of the following steps:

1. Generate a model of the *MediaStore* scaled by a given factor *n* (no time measurement). The model uses a shared component repository, but all assemblies, allocations, resource containers and resource links are replicated.
2. Initially run the given analysis on this generated model. There should be no wrongly allocated assemblies.
3. Iteratively allocate the `MySql` database assembly to an isolated component. For a given size *n*, this means to move *n* assemblies. The time measurement is done separately for each assembly. Each reallocation should cause three elements in the analysis result because there are three connections between the database adapter assembly and database assembly.
4. Iteratively allocate the `MySql` database assemblies back to the database assembly where they were before. Each reallocation should eliminate three elements in the analysis result set.
5. Iteratively remove the database resource container from all of the network links. Each such operation should again cause three new matches in the analysis result set.

Initially and after every model manipulation done in steps 3–5, we compute the number of elements in the analysis result set, i.e. how many assemblies are incorrectly connected given that there is no link to the specified resource container. The implementation of this benchmark and R scripts to generate result images are publicly available online[111].

9.11.3. Results

For each of the analyses, we run them both in batch mode and in incremental mode.

The results for the initial runs of the analyses are depicted in Figure 9.48. One can see that in the Deep Modeling analyses, the overhead to create the DDGs is smaller: They are closer to their respective batch analysis

[111] `https://github.com/georghinkel/DeepModelingDemo`

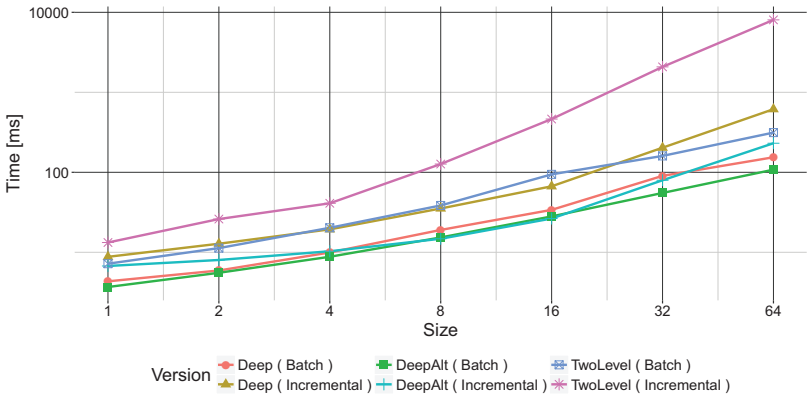


Figure 9.48.: Initial analysis time to check the correct allocation of assemblies

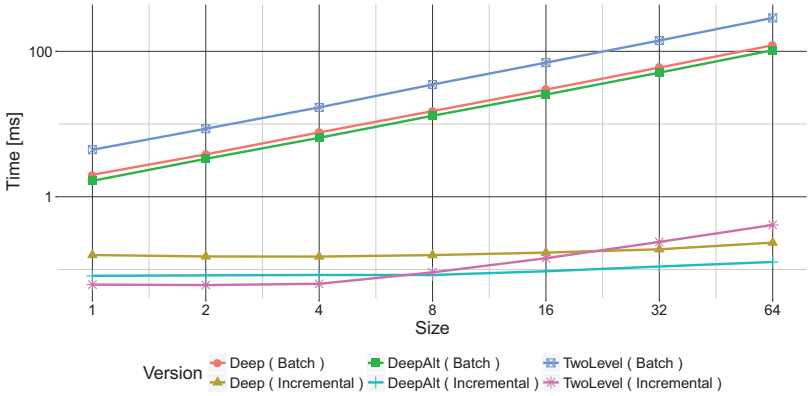


Figure 9.49.: Median response times for updates of the allocation model to updated analysis results

results. Meanwhile, creating the DDG for the two-level analysis means more overhead because the queries are larger. Furthermore, meanwhile in batch mode, a `FirstOrDefault` query simply quits after the first element, the incremental version always has to process all elements as a change could make them first. Thus, the overhead for thw two-level analysis even increases with an increasing model size.

The results for the median response times from an update operation in each of the steps to an updated analysis result are depicted in Figure 9.49. One can clearly see the difference between incremental and non-incremental executions: Even the slowest non-incremental version of the analysis is faster than the fastest non-incremental analysis by more than two orders of magnitude.

However, the results also show a difference among the incremental solutions. While the Deep Modeling versions of the analysis are slower for smaller models, the results confirm a better scalability with larger models. For size 32 (which is an architecture model with 224 assemblies), the additional overhead pays off. For size 64, the two-level version of the analysis is already significantly slower.

Lastly, the changes that we applied to the software architecture models are only at the lowest level, i.e. we were changing only deployment information. If we were to change upper levels, this would be fine for the two-level version of the analysis (though it has to account for temporarily broken validation constraints) but in the Deep Modeling model, such a change would create a need to regenerate and recompile the lowest level. In any case, this would be much slower than a propagation of a change in the two-level analysis.

9.11.4. Summary

The results of this case study are twofold: They show that at least for the model analyses depicted, there is indeed a speedup when using the Deep Modeling version(s) of the analysis, in particular also in the incremental mode, but that speedup is much smaller than the speedup that can be obtained by using incremental computation at all. Therefore, the case study indicates that whatever reasons there may be to apply Deep Modeling, the performance of incremental model analyses may not be a good one.

9.12. Threats to Validity

In this section, the validity of the results obtained in the presented validation are discussed. We separate this discussion in the internal validity in Section 9.12.1 and external validity in Section 9.12.2.

9.12.1. Internal Validity

The internal validity of the case study results greatly depends on the type of result. In particular, there is a huge difference between runtime measurements, memory measurements and questionnaire results.

Common to all case studies from the TTC, we can safely exclude an experimenters bias as the cases have been authored by other researchers (with the exception of the TTC 2017 Smart Grid case) and – more importantly – the set evaluation criteria have undergone a peer-review process. The Bike shop modeling case study is taken from the MULTI workshop. Therefore, we can also exclude a bias from the case authors.

9.12.1.1. Performance

There is a threat of confounding factors for the performance measurements. Other applications than the benchmark may be running on the machine such that the measured performance times may not be perfectly accurate.

To compensate this threat, all background services have been terminated where possible, including messengers, storage services and connection services.

However, there are also some services that are inevitable connected to the benchmarks such as the garbage collector and the just-in-time compiler. To reduce the influence of the latter two, all measurements in all case studies have been repeated at least ten times. Furthermore, most benchmarks use a warmup that eliminates the influence of the just-in-time compiler.

The benchmark framework of the TTC Train Benchmark 2015 and the Smart Grid Benchmark from the TTC 2015 do not consider such an elimination such that the just-in-time compilation does influence the results. However, this influence is only important for the smaller model sizes. For the larger model sizes, the time for the just-in-time compilation can be neglected.

Less on the measurement itself, there is also a difference of the used technology because other solutions generally use EMF instead of NMF. Therefore, differences in response times may be due to the difference of the used framework instead of difference in the used incremental tool. However, we think

that the technologies are tightly coupled to the underlying model technologies such that this comparison is actually fair. Furthermore, clearing this effect from the measurements required to reimplement the tools in another platform, an overhead which is not justified by this confounding effect.

Overall, we think that the threats to internal validity are rather small. Due to repetition of measurements, we think that the influence of garbage collection and just-in-time compilation is much smaller than the observed differences between incremental and non-incremental tools that are in the order of multiple magnitudes or roughly half an order of magnitude for the comparison with VIATRA Query or Reactive ATL.

9.12.1.2. Memory Measurements

For the memory measurements, there is a large confounding factor because we only measured the working set size. Therefore, the memory measurement is confounded by the memory consumption of the modeling framework as well as the memory consumption of any infrastructure code. Lastly, the memory consumption also depends on the memory efficiency of the underlying technology which is often different because NMF uses .NET meanwhile other solutions usually use Java.

Furthermore, the garbage collector is a very important confounding factor for the memory measurements because it makes a tremendous difference whether the memory measurement is done before or after the garbage collector frees memory for unused objects. Because it is not possible in .NET to clearly identify when garbage collection has taken place or to manually trigger it[112], the influence of garbage collection cannot be avoided.

Therefore, what one would rather want to measure is the amount of memory that is actually used by the incremental tool. However, this is not possible easily with the current architecture. Therefore, the memory measurements have a low accuracy.

[112] It is indeed possible to *suggest* the system to perform a garbage collection but it is not guaranteed when that happens.

9.12.1.3. Understandability

We list and discuss the most important threats to internal validity of the understandability below.

Confounding factors There are severe confounding factors in the data for the understandability: NMF is yet a relatively unknown approach and in general, the C# language is much less common than for example Java in the model-driven community. Therefore, many participants of the TTC are (sometimes even admittedly) not familiar with the technology, which clearly confounds the understandability results. Therefore, the results on the understandability have to be seen as a lower boundary: It is likely that participants more familiar with the technology find the approaches more understandable.

In the Families to Persons case study, another problem was that due to logistics reasons, the solution could only be presented using a video while other solutions were presented in person. We are not sure whether this had an influence.

History For the open peer reviews, it is unclear in which order the assigned solutions were reviewed. Therefore, an influence of history cannot be excluded. For the presentation at the TTC, there is a clear influence of history since the solutions are presented in sequence.

Instrument change An instrumentation effect can be excluded. Both open peer reviews and live contest questionnaire responses have been asked for by the TTC organizers, not by the authors of this thesis.

Several common forms of internal threats to validity such as differential attrition, ambiguous temporal precedence, maturation, diffusion or regression towards the mean do not apply because the understandability was only evaluated once and not over a period of time.

Overall, the results for understandability are very inaccurate and have to be taken with great care. This also explains the simultaneously very good results for understandability in the Java refactoring case study meanwhile

the results for understandability in the bidirectional Families to Persons case study are rather bad.

Mainly for the unfamiliarity of the TTC audience both with NMF and .NET in general, we think that the true understandability of the NMF solutions is better than the understandability from the questionnaire responses.

9.12.2. External Validity

Again, we split the discussion of external threats to validity on the type of validations. However, since both are hard metrics, the threats to external validity are the same for performance and memory measurements.

9.12.2.1. Performance and Memory Measurements

For performance and memory measurements, we face the problem that it is unclear to what degree the obtained results can be generalized for other applications, input model characteristics and change sequences. Further, it is unclear to what extend the opponent solutions represent the opponent tools.

Though the change sequence used in the various case studies have been generated, they depend on the selection of changes and their proportion. To mitigate this threat, we tried to cover all possible elementary changes and tried to create a realistic ratio between the different change types where this was possible. However, these ratios are not based on empirical research. Furthermore, in case of the TTC 2015 Train Benchmark and the TTC 2017 Families to Persons case study, we sticked to the original benchmark proposals (with the exception of the inhomogeneous change sequences for the evaluation of Incerator) that do only consist of homogeneous change sequences. This is required to allow a comparison to other tools.

To mitigate the threat of a lacking generalization to other contexts and other applications, we tried to include a diverse set of case studies involving different types of artifacts and different domains. We think that the presented case studies show a good variety of those and we therefore expect that the results from the case studies generalize to a wide range of applications.

For the comparison with opponent solutions, most of the solutions using other tools have been authored by the authors of the respective tools. This way, we can safely assume that these solutions are the best solutions possible using these tools. As a consequence, we expect that the comparison to these tools generalizes to other applications.

9.12.2.2. Understandability

The participants of the TTC cannot be seen as representative for the likely users of incremental model analyses and incremental model transformations. Rather, in the last years they represent the authors of solutions to the TTC cases and perhaps a few other participants of the STAF event in which the TTC is embedded. The solution authors may or may not be biased towards their own solution, even though all of them are researchers and therefore should give an unbiased opinion on all solutions.

A similar statement holds for the open peer reviews where reviewers may or may not be biased towards their own solutions. Because in the open peer reviews of the TTC, each solution is only reviewed by two reviewers, there is an influence of chance whether the reviewers are biased.

However, the selection of reviewers and the selection of participants of the solution presentations at the TTC is outside the control of the author of this thesis. Therefore, similar to the validation of correctness, the threats to validity limit the expressiveness of the validation data but as we cannot influence the validation setup, we do not have an option to make the data more conclusive.

Over all, we think that the results on the understandability are rather preliminary and should be supported by future research to gain credibility.

9.13. Validation Summary

In the following, we summarize the insights obtained from the validation of this section with regard to the validation goals as defined in Section 9.1.

Applicability The variety of case studies shows a wide applicability of the presented approaches for practical problems, particularly for cyber-physical systems that are described in software models.

For model transformations, we obtained that incremental transformations with synchronization blocks are at least as expressive as the ATL model transformation language, considerably the most common model transformation language.

Our notion of refinements and structural decomposition turned out to neatly express the Bicycle modeling challenge of the MULTI 2017 workshop, whereas we found severe problems in the models of the only actually submitted solution using MultEcore 9.13.

However, the problems presented in this chapter are rather small and the approaches of this thesis have not yet been applied in a large industrial application. Therefore, a validation on larger artifacts such as more complex analyses or model transformations is subject to future work.

Performance The response times show that in all considered cases, the response time from a model change to an updated analysis or transformation result could be reduced significantly, often even multiple orders of magnitude. The automated optimization from Chapter 5 show limited practical advances, but we think this may also be due to the smaller size of the case studies.

For the incrementalization system introduced in Chapter 4, we have seen that the annotations of explicit incrementalizations for the query operators yields a performance for incremental analyses that is able to catch up with state-of-the-art specialized incremental tools such as VIATRA Query. For all five queries of the TTC version of the Train Benchmark, NMF Expressions was faster than VIATRA Query for the medium-sized models and in the *SwitchSet* query, even faster for the largest models considered.

The performance results for the incremental model transformation approach is even better. In particular comparing our approach with bidirectional model transformation languages, our approach is able to beat other incremental model transformation tools such as primarily eMoflon by multiple orders of magnitude. However, also for unidirectional model transformation languages

such as ATL, our approach turned out to yield a great performance improvement in comparison to repeated execution of the model transformations or propagating changes using SyncATL.

For incremental analyses in the context of Deep Modeling, we were able to manifest that the usage of Deep Modeling can lead to improved response times, but these speedups are relatively small in comparison to the speedups that can be achieved using incrementalization.

Memory Consumption The results with regard to memory consumption show that the additional memory consumption required by the incremental model analyses stays at a tolerable scale and is comparable to other approaches such as VIATRA Query. In other case studies, it turns out that the working set of an incremental update propagation is even smaller than the one for a batch solution, simply because the incremental solution does not have to load and transform the same models over and over again.

Understandability As indicated in Section 9.12, the data that we have for the understandability are the least accurate and in fact, the results differ a lot. While the open peer reviews of the Java refactoring case study showed maximum points for understandability, the understandability in the Families to Persons case was evaluated much worse, even though both solutions referred to similar artifacts, namely bidirectional model synchronizations using NMF Synchronizations. Especially taking the analysis of the rather bad result in the Families to Persons case study into account, we interpret this as follows: We believe that it is very understandable *what* a model synchronization using NMF Synchronizations does but it is not very understandable *how* it does that. We assume that this problem may be mitigated if more developers are aware of the C# language features and their usage for declarative incremental and bidirectional model transformations such as executed in this thesis.

Correctness In all case studies, the indicators for correctness indicated a correct incrementalization. Thus, at least in the case studies, there were no obvious bugs.

Relaxation of Assumptions The incremental data flow transformation case presented in Section 9.4 shows that the approach can also be applied beyond its original scope, as side-effects can be tolerated if they can be controlled.

10. Experiences and Lessons Learned

The goal of this chapter is to reflect on the experiences and lessons learned from the case studies including but not limited to those presented in Chapter 9. However, the findings did not fit into the discussion of these case studies.

10.1. Whether incremental execution is beneficial depends on how an analysis is formulated

The original NMF solution of the Train Benchmark did not use the implementation of the *SemaphoreNeighbor* query as depicted in Listing 9.5 but rather used the implementation depicted in Listing 10.1. The advantage of the latter query is that it appears simpler. Furthermore, the complete pattern match could be reconstructed from the match more easily, i.e. including the elements `te1`, `te2` and `sensor1`. In the more recent implementation, these elements have to be reconstructed from a pattern match.

The old version of the analysis has a very bad performance, even for smaller models. In batch mode, the analysis is already slower than the reference Java solution (meanwhile it is faster for all the other cases), but in incremental mode, the runtime is extremely slow. For model sizes up to 8 (which corresponds to about 10,000 model elements), incremental execution was significantly slower than the batch solution and moreover showed a much worse scalability. For larger model sizes, the solution even ran out of memory.

The reason why the original solution depicted in Listing 10.1 has such a bad performance is twofold: First, the solution iterates over every pair of routes, regardless whether such a pair is a candidate for a pattern match. For the

```
1  Fix(pattern: from route1 in routes
2           from route2 in routes
3           where route2.Entry != route1.Exit
4           from sensor1 in route1.DefinedBy
5           from te1 in sensor1.Elements
6           from te2 in te1.ConnectsTo
7           where te2.Sensor == null
8               || route2.DefinedBy.Contains(te2.Sensor)
9           select new { Route = route2, Semaphore = route1.Exit },
10     action: match => match.Route.Entry = match.Semaphore);
```

Listing 10.1: The original NMF solution of the *SemaphoreNeighbor* query in the TTC Train Benchmark 2015.

larger models that contain thousands of routes, this causes a higher computational complexity. We think that this is also the reason that already the batch mode version of the solution was slower than the reference solution in Java. In the incremental execution mode, this higher computational overhead also implies a much larger DDG. This causes a higher memory overhead.

Second and also very important for the incremental execution, the changes done in the scope of the benchmark affect a much larger part of the analysis. The elementary model changes are exclusively changes to fix some of the constraint rule violations, i.e. setting a new entry for a given route r_1. For any route r_2 that exits at the old entry of r_1, this change causes the filter condition of Line 3 in Listing 10.1 to pass. As a consequence, the pair (r_1, r_2) enters the remainder of the query in Lines 4-9. This means that the DDG templates are instantiated for this pair and further elements created as a result of `SelectMany` operations such as in Lines 4-6. This increases the memory consumption. Furthermore, the overhead of instantiating a DDG template in comparison to plainly executing the analysis in batch mode costs time, though the created DDG template instantiations do not add any value because changes of the network topology that they would be reacting to simply do not occur during the benchmark.

Both problems are mitigated in the new implementation of this query as depicted in Listing 9.5, using the fact that at most one other route may be connected to a given route.

```
var parallelQuery = from route1 in routes.AsParallel()
                from route2 in routes
                where route2.Entry != route1.Exit
                from sensor1 in route1.DefinedBy
                from te1 in sensor1.Elements
                from te2 in te1.ConnectsTo
                where te2.Sensor == null
                 || route2.DefinedBy.Contains(te2.Sensor)
                select new { Route = route2, Semaphore = route1.Exit };
```

Listing 10.2: The original NMF solution of the *SemaphoreNeighbor* query in the TTC Train Benchmark 2015 run in parallel.

10.2. A good speedup in parallel execution is not an indicator for a good speedup in incremental execution

The similarity of parallelism and incrementalization in their common goal to improve the performance of a given analysis may suggest that analyses where incrementality achieves good results are the same as those where parallelism allows large speedups. This is enforced by the literature where both of these properties could be implemented using the same paradigm [33].

However, during the validation performed for this thesis, we found that this is not the case: There are analyses where parallelism achieves good results, but incrementality does not.

As an example, consider again the original implementation of the *SemaphoreNeighbor* query of the Train Benchmark as depicted in Listing 10.1. As explained in Section 10.1, the incrementalization of this query is very inefficient, especially given the change sequences that are applied in the Train Benchmark. However, it is actually straight-forward to run this query in parallel: One just has to run the outermost loop in parallel.

To achieve this, one can simply reuse the parallel LINQ implementation as provided by Microsoft and parallelize the *SemaphoreNeighbor* query as depicted in Listing 10.2. All that is necessary for the developer is to add a

call to the `AsParallel` extension method in the beginning of the query. The rest of the query stays exactly the same.

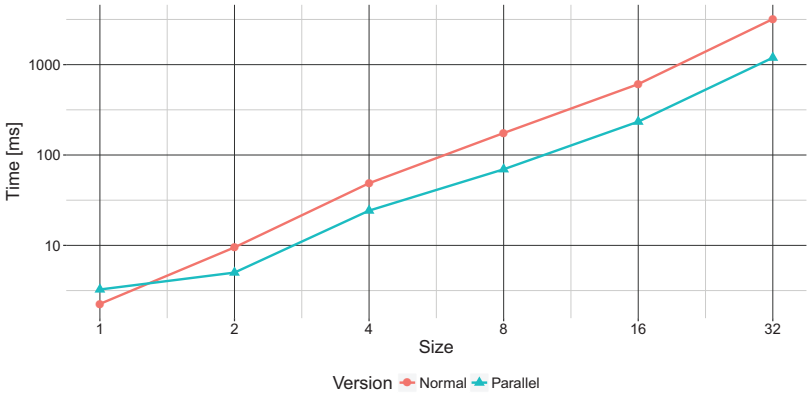


Figure 10.1.: Benchmark results for parallel execution of the original *SemaphoreNeighbor* query

For example, we depicted the times for running the queries in the parallel and non-parallel version on the railway network models of different sizes in Figure 10.1. The size axis is relative, same as in the original Train Benchmark. This means that size 1 has about 1,300 model elements, size 32 has about 50,000.

As could be expected, the parallel execution times grows with the size of the input models, very similar to the standard LINQ implementation. However, one can clearly see that the parallel version is faster than the sequential version by an almost constant factor. Only an initial overhead of the parallel query causes the speedup for smaller models to be lower. For the largest model size, the speedup of the parallel version against the sequential implementation is 2.68, a considerably good result for a machine with four cores, particularly because given the low implementation effort to support parallel execution in this case.

The reason for this is that the potential for a speedup in through parallelism lies in the fact that multiple routes can be analyzed in parallel meanwhile for the incremental evaluation, this is irrelevant because only one route is affected by a given model change at a time.

10.3. A good speedup in parallel execution is not an indicator for a good speedup in incremental execution

The previous section demonstrates a counter-example of an analysis that achieves very good results in an incremental execution but is very hard to incrementalize. One could think that at least the opposite is true, that an analysis that has a good performance in an incremental execution should also yield very good parallel execution results.

This is also not the case. An example of model analyses that achieve good incremental execution results but are hard to run in parallel manner are model transformations. As suggested in Sections 9.5 and 9.6, the incremental execution of model transformations scales very well with an increasing model size suggesting that the incremental model transformation approach of this thesis is fully incremental.

However, it turns out that model transformations are often also very hard to run in parallel. Meanwhile there are approaches that indeed try to run model transformations in parallel [202][113], there is a major obstacle: A model transformation uses the trace as a shared resource. In a parallel execution, any access to the trace must be locked. This causes a high synchronization overhead and leads to very small speedups in comparison with a sequential execution of the model transformation.

In contrast, for the incremental execution this shared trace is highly valuable as it allows to quickly find corresponding elements for a given model element that is affected by an elementary change. This allows to identify necessary corresponding changes in the target model very efficiently, leading to a highly efficient incremental execution.

[113] The proposed approach to run a ATL model transformation in parallel boils down to run the model matching phase in parallel where the matches for each rule are computed in separate threads.

10.4. The influence of the order

An important insight regarding when incrementality yields good performance results is the order of elements. As it turns out, whether or not an incremental algorithm has to respect the order of elements in an analysis has a massive influence on the performance gains that can be achieved using incremental execution. At the same time, this information is mostly unimportant for batch analysis: Meanwhile a remove operation is significantly more expensive in ordered collections[114], this information is unimportant for most query operators such as `Select` or `Where` as they preserve the order of elements by default. This behavior is natural because these operators simply iterate the source collection and perform filters or mappings on the fly.

For an incremental analysis, preserving the order of elements is much more complicated as a single change of a collection results in multiple changes of element positions.

Despite the importance in an incremental setting, it is not as easy to specify that the order of elements in a particular collection does not matter for an implicit incrementalization system. In a batch specification, developers usually iterate through the elements of a collection sequentially, thus in the order in which the elements appear in the collection. As a consequence, an incrementalization system operating only on batch instructions has no chance to detect whether the order of elements has an importance and must assume that it is, although this makes the incrementalization much more inefficient.

To avoid this, the existence of manually incrementalized functions that express in their semantics that collections they are working with is of great importance for an incrementalization system to produce efficient results. The manual incrementalization can then take this fact into account and implement the incrementalization accordingly.

[114] In an ordered collection, removing an element has complexity $O(n)$ meanwhile a hash set implementation of an unordered collection allows a remove in amortized $O(1)$.

10.5. The influence of the model and the change sequence

Whether the incrementalization of an analysis produces good results does not only depend on the characteristics of the analysis but also on the characteristics of the model and the change sequence applied to it.

An example where this becomes very obvious is the application of Incerator to the Train Benchmark. Meanwhile the characteristics of the *PosLength* query and the *SwitchSensor* query are very similar, the difference between the instruction-level strategy and the repository listening strategy produce very different results, purely because there are more segments in the model than switches[115].

This insight also confirms in Section 9.5, the model transformation from finite state machines to Petri nets. There, we also recorded the time that the incremental model transformation takes to propagate a given change. As it turns out, simple attribute changes of states or the state machine are much faster to propagate than more complex changes such as adding or removing states or transitions. This is because attribute changes are reflected in simple property changes in the target model meanwhile for a newly added state, the transformation engine has to process a rule execution, which means a slightly higher computational overhead.

10.6. Usage of Deep Modeling

One of the main rationale behind Deep Modeling is to reduce the accidental complexity of models. An important experience that was made when working with the approach from Chapter 8 is that removing accidental complexity from the metamodel does not necessarily make the metamodel more understandable. In particular, we found that the approach from Chapter 8 turns out to be surprisingly difficult to apply.

[115] The fact that the model manipulation for the *SwitchSensor* query is more complex does not matter in this case, because the model manipulation is the same regardless of the incrementalization configuration.

As a reason, we assume that this is because the amount of assumptions that instantiation causes in comparison to a reference or a containment. Referencing another model element does not imply any constraints, a containment reference only implies the constraint that there may be at most one container element at a time for a model element. Instantiation, however, causes a lot more constraints because for each attribute or reference of the type element, the instance has the possibility to assign a value (either value object in case of an attribute or another model element in case of a reference) but constrains its type and multiplicity. While these implicit constraints have several advantages over making these constraints explicit, one must be aware of them.

10.7. Conclusions

The main result of this chapter is that although an implicit incrementalization of model analyzes and transformations makes incrementalization more easy and maintains the understandability of a batch specification, developers still have to have a good understanding of how the incrementalization system works and how they need to formulate the analysis to achieve the best results.

In particular, despite the common goal of improving the scalability, the necessary characteristics that allow a good speedup are different for incrementalization and parallelism. Meanwhile parallelism achieves good results if there are less dependencies to consider, incrementalization rather benefits from a direct mapping of changes of the source model to changes of the result. In case of model transformations, such a mapping can be provided by the transformation trace, which we take as the reason that the results in this area are so good.

The chapter also shows that the characteristics of a model analysis such that good performance results can be achieved by incrementalization depends on the model, the change sequence applied to it and how easily the changes of the model can be mapped to changes in the result. In addition, constraints such as the order that are implicitly encoded in a batch specification may be toxic for the performance of incremental analyses.

Part VI.

Epilogue

11. Related Work

This chapter briefly summarizes related work in the research field of this thesis. The description is divided by research fields.

In particular, Section 11.1 reviews related research in the field of incremental computation systems. Section 11.2 reviews research in model transformations. Section 11.3 compares the meta-metamodel extensions presented in this thesis with other approaches.

11.1. Incremental Computation Systems

This section reviews the current state of the art in incremental computation with a focus on model-driven engineering. We divided the existing tools into general-purpose approaches applicable to any analysis through support of a Turing-complete language and those specific to a smaller class of analyses. We refer to the latter as specialized incremental tools. These tools do not have a restriction in the domain but in the kinds of analysis that are supported, e.g. only attribute grammars, queries or graph patterns. Further sections review the related work in reactive systems, optimization and distribution of incrementalization systems.

11.1.1. General-Purpose Incremental Computation Systems

Pugh and Teitelbaum [166] were the first to apply memoization to incremental computation. Memoization is applicable to any referential transparent function but rests on the assumption that the data structures it operates on is immutable. If this is not the case, a function result may become invalid even though the arguments did not change. This makes memoization not applicable for model analyses.

Later, Acar and others created Self-Adjusting Computation (SAC), a framework to support the development of incremental programs [1] using the then newly introduced DDGs. A good overview on SAC is provided by Acar [2]. While the framework originally supported functional languages, it has been extended to imperative languages based on C [81]. SAC operates on the batch specification of an analysis annotated with explicit incrementalization primitives. From these primitives, a DDG is deduced that keeps track of changes.

Closest to our approach in chapter 4, Tracable Data Types (TDTs) have been integrated into SAC to allow developers to supply a custom incrementalization of an algorithm in a dedicated data structure [3]. TDTs have an internal virtual clock and work by allowing developers to explicitly revoke previous operations and return the earliest inconsistent state, if any. These operations include both state management and queries, which allow different states during an analysis, but require the developer of an analysis to manage the state of the data structures on their own. As a consequence, TDTs are limited to their own data structures while our approach allows the incrementalization of higher-order methods that are independent of data structures used in predicates. Furthermore, TDTs require some boilerplate code to use them in SAC [3]. Our implementation only requires a method annotation so that the dynamic algorithm can be reused in its compiled form.

Based on SAC, Carlsson was the first to find that incrementalization can be represented as a monad [42]. However, the paper concentrates mostly on the advantages to implement such a system in Haskell rather than on the conceptual benefits of integrating manually incrementalized versions of functionality that we have proven in this thesis. In particular, as argued in this thesis, incrementality is *not* a monad (in the original meaning from category theory) because the unit transformation must not be natural – a fact that Carlsson ignores because it is not strictly required for an implementation in Haskell[116]. While Haskell makes it very easy and convenient to specify a monad, it is very hard to manually exchange the incrementalization of a given method.

Hammer and others introduced the idea of demanded computation graphs, implemented in Adapton [84, 82]. Demanded computation graphs make sure

[116] In the context of functional programming, monads are often defined without the requirement that the unit transformation must be natural.

that a change propagation is only performed if the result is actually needed, a feature that we implemented using reference counters on the DDG nodes. Similar to SAC, Adapton does not work implicitly, such that developers have to think carefully about where to insert incrementalization primitives.

SAC and Adapton both have the problem that they work explicit: The programmer must give a hint which parts of an analysis can be saved for repeated analysis runs. Furthermore, the mutation of the input must be done through a dedicated mutator component in SAC or through refreshing computation thunks in Adapton. Our approach is able to pick up change notifications from the generated model API and works entirely implicit, i.e. the programmer does not have to change the code at all. This makes it possible to use incrementalization in mainstream technologies.

Chen et al. have developed an approach to transform programs written in purely functional Standard ML into SAC [43], allowing developers to omit incrementalization primitives. Hence, the approach works implicit. However, we are not aware of any research that integrates the usage of TDTs into this framework to make them more efficient. Therefore, this technique has the problem that incrementalizations of collection operators are inefficient.

On a rather technical level, neither Adapton nor SAC are currently available to be used with MOF models. While there is no publicly available implementation of SAC, Adapton has a freely accessible and maintained implementation in Rust[117]. However, Rust has very limited support for object-oriented design. In particular, Rust only allows inheritance and dynamic binding for traits, but trait objects cannot be downcasted. However, this is a mandatory requirement for many metamodels of the case studies from Chapter 9.

Other approaches to incremental computation include entirely new programming models that allow an easy incrementalization or parallelization. An example of these approaches is revision-based computing [33]. However, here the developer has to explicitly think about where to insert such a revision concept.

[117] `http://adapton.org/`, retrieved 18 Jul 2017

11.1.2. Specialized Incremental Approaches

Incrementality is a desirable property as it promises to save computational effort when analyses are computed repeatedly. Therefore, it has been a subject of research for decades [170], prominently for example with the search for incremental compilers [174]. Common to all of these approaches is that they make advantage of abstractions they make on the analysis to perform at the cost of limited applicability. As soon as one formalism alone is no longer capable of expressing an analysis, multiple approaches must be integrated, causing a large integration overhead [24]. This is especially important in maintenance scenarios where perfective changes require to extend an analysis such that it no longer suits the given formalism.

Among the first incrementalization systems specialized on a limited class of problems is the approach by Reps [175] for attribute grammars. This approach works by using a static dependency graph for attribute evaluations for which Reps has shown that an optimal-time reevaluation strategy can be found by reevaluating the attributes in a topologically sorted order of a static dependency graph. This approach rests on the assumption that the data processed by the attribute grammar is immutable. As Reps applies this technique for parse trees, this assumption is reasonable, but it does not hold for models in general.

Willis et al. have achieved convincing results for an implicitly incremental execution of the Java Query Language [217]. In their approach, they find all the places where caches may be invalidated through aspect-oriented programming. As a consequence, all these places must be known at compile-time. Thus, model manipulation and analysis are tightly coupled and cannot be separated into different modules.

On the .NET platform, a range of non-academic projects aimed to provide change notifications for queries, sometimes with an incremental execution. Among the rather mature are Continuous LINQ [107], Bindable LINQ[118] and Obtics[119]. However, the example query we present in Section 1.6 only works with Bindable LINQ where the runtimes are far worse than rerunning the query for each elementary change because the approach requires a compilation for each model change.

[118] `http://bindablelinq.codeplex.com/`, retrieved 2 Aug 2017
[119] `http://obtics.codeplex.com/`, retrieved 2 Aug 2017

A similar problem to the incremental queries appears in relational databases when maintaining materialized views [29]. An overview on the research can be found in [79]. These approaches have also been applied to object-oriented databases, as for example in [132]. Some approaches like notably LINQ or SQuOpt [71] have ported this database technology to object-oriented languages, in case of SQuOpt Scala.

A popular approach to specify queries, especially in graph transformation, are Graph Patterns. Bergmann et al. have created IncQuery, an incremental pattern matching system for Graph Patterns [26, 25]. This approach uses a Rete network [63], a static dependency graph, whose nodes are primitive filter conditions or joins of partial pattern matches. Each node represents a set of (partial) pattern matches. This approach can support mutable models because the notification API of models can be used to determine when matches must be revoked or new matches arise. Unlike NMF Expressions that requires a *dynamic* dependency graph, this means that queries as in the Train Benchmark can be incrementalized using only a *static* dependency graph.

IncQuery was integrated to EMF models as VIATRA Query [205]. This system was also used to evaluate queries [172] and certain OCL expressions [23]. We evaluated NMF Expressions against VIATRA Query in Section 9.3.

An incremental execution of OCL has also been subject of research for incremental constraint checking [40, 173, 204]. These approaches are either limited to boolean-valued constraints or limited to static analysis. The latter is the same as the repository change incrementalization strategy in Chapter 5.

Lastly, in the field of algorithmics, a whole class of algorithms is dedicated to process incremental changes, dynamic algorithms. Prominent examples from this field include the dynamic spanning forests by Holm et al. for connectivity analysis [109] and King and Sagerts approach for dynamic transitive closures [125]. These algorithms often have a sub-optimal runtime in absence of changes but can update their data structures according to changes. Our approach does not compete with these dynamic algorithms but provides a way how they can be used to specify incremental analyses using a batch specification. An overview of dynamic algorithms, particularly for dynamic graph algorithms was provided by Ramalingam and Reps [169].

A generic theory of changes applicable in a wider range of applications was developed in the scope of the SCuOpt project by Cai et al. [41]. While this approach has a wider applicability than just a single class of analyses, the authors do not yet have a concept how to mix several of such analysis kinds. Thus, it serves as a foundation for the development of specialized incremental tools. In the paper, Cai et al. applied the approach to a map-reduce technique, which is a small subset of incremental queries.

11.1.3. Reactive Programming

A very related paradigm to incremental computation is reactive programming where the goal is to get notifications for changes. An overview of 15 languages for reactive programming was created by Bainomugisha [18]. According to the taxonomy suggested in this survey, NMF Expressions is based on events with a push-based evaluation model and implicit lifting.

Our approach can be interpreted as a way with a formal foundation how implicit lifting can be overridden to gain performance. Most approaches in reactive programming circumvent this problem as they apply explicit lifting [18]. Even if the lifting works implicit, the approaches do not incrementalize methods using their structure but rather execute a given predicate as is.

In reactive programming, the developer has to explicitly declare signals to which the analysis should react. Our approach makes this implicit as the incremental algorithm automatically attaches to the model as required. Notable exceptions are FrTime [47] and FlapJax [148].

Particularly on the .NET platform, Reactive Extensions (Rx) have been introduced to support reactive programming [146]. Similar to our approach Rx uses the query syntax to combine several streams of data.

Reactive programming approaches are built upon an important assumption, namely that signals do not change once they are processed. That is, they operate on an (potentially infinite) sequence of immutable data. This is a contrast to model analysis where the model usually has an approximately constant size, but is mutable.

11.1.4. Scenario-specific optimization

We are not aware of any other incrementalization approach that tries to systematically optimize the performance for a given scenario.

11.1.5. Distributed Incremental Tools

The only incremental computation tool that we are aware of is the distributed version of EMF-IncQuery: IncQuery-D [195]. This systems works by distributing the partial matchers typically for a RETE-network onto multiple machines. The distribution is therefore static (a Rete-network is a static dependency graph) in the sense that it depends on the pattern to be matched, but not on the data. Our distribution mechanism is potentially more fine-grained as it allows a distribution based on the actual data but the finer granularity also causes additional overhead. The fine-grained distribution in combination with the Orleans auto-scaling abilities may cause a better elasticity. However, we did not do a detailed performance comparison with IncQuery-D.

11.2. Model Transformations

The thesis touches a multitude of model transformation topics. In the following subsections, related work in different aspects of model transformations are discussed.

11.2.1. Incremental Model Transformations

There are several different approaches for model synchronizations. For a range of approaches, Kusel and others have created a survey [133].

TGGs are a mature approach for incremental model synchronizations. TGGs are a graphical, declarative, and bidirectional approach, based on graph transformations [182]. Leblebici et al. created a survey of using TGGs for incremental model synchronizations [139]. A detailed comparison between NMF Synchronizations and the TGG implementation eMoflon is made in Section 6.4. A performance comparison is done in Sections 9.5, 9.8 and 9.9.

Jouault and Tisi [122] introduced Reactive ATL, an approach for incremental model synchronization based on the ATL language that works online (the model elements must be kept in memory when changes are made to them). To accomplish this, they only support a subset of the ATL language and make changes to the ATL compiler. They also implemented a prototype of their incremental execution algorithm for ATL, but since it only supports a small subset of the ATL language, the usage of their prototype is limited. They do not support imperative statements, helpers, lazy rules, queries, the refining mode and variables in rules. This restricts the supported ATL transformations considerably, especially since helpers are widely used in ATL transformations.

SyncATL, a similar approach of Xiong et al. [219] also introduces incremental model synchronizations for the ATL language. This is achieved by extending the ATL Virtual Machine (VM). Their prototype works offline and therefore operates differently. To identify the differences in two versions of a model, both have to be compared first. Afterwards, the changes can be propagated. The exact subset of the ATL language supported by their approach is unclear. After analyzing the example transformations that can be used by their approach, it seems as if they at least support matched rules, helpers and different OCL constructs. However, SyncATL statically relies on identifiers to match model elements, whereas our approach is able to use custom developer-supplied heuristics as well.

Varró and others created VIATRA, which in its latest form is a reactive transformation platform built on incremental queries [206]. With this platform, developers can declaratively specify graph patterns that can be used in incremental model transformations [171], but in the latest version of VIATRA the actual transformation has to be specified by the developer. Therefore, we rather see VIATRA, in particular VIATRA Query [205] as an alternative to NMF Expressions, not to NMF Synchronizations that combines the latter with declarative model transformation concepts.

Beaudoux and others presented the Active Operations Framework (AOF) [20], which is slowly getting applied to model transformations to make them incremental [120]. The transformation language proposed in the latter paper is actually very similar to NMF Synchronizations, both in terms of the syntax and in terms of concepts. Therefore, we think that the mapping presented in this paper can also be applied to make ATL incremental through AOF.

Lauder et al. provided an incremental synchronization algorithm that statically analyzes rules to determine the influence range while retaining formal properties [136]. In contrast, we are using a *dynamic* dependency graph such that we have a much more precise understanding which model changes influence a transformation rule.

11.2.2. Bidirectional Model transformation languages

A good overview on bidirectional model transformation languages, including a classification scheme, was created by Steven [190] or Hidaka et al. [90]. In this classification of the latter, our approach operates on the technical space of models (MDE) and consists of both forward- and backward-functional correspondences with a Turing-complete (through extensions), bidirectional specification. It reacts on live delta-based changes[120] and all operations are supported through change translation, though the enforcement of these translated changes are only checked dynamically. The explicit trace is only available in memory and we leave it to the user to persist it, if necessary.

Among the existing bidirectional model transformation languages is the standardized QVT Relations language [168], though Stevens has identified some semantic issues with it [191]. To the best of our knowledge, there is also no approach that executes QVT-R transformations incrementally. Nevertheless, Section 9.9 contains a performance comparison with the Medini implementation of QVT-R.

The most prominent incremental and bidirectional transformation languages may be Triple Graph Grammars, originally introduced by Schürr [182]. Multiple tools implemented this paradigm [91] and Giese et al. even implemented them incrementally [75]. Triple Graph Grammars are usually specified graphically or through external languages. As we have shown, we can express some TGG rules in a single line of code, which we believe is not easily possible for TGGs. Furthermore, our approach is extensible with user-supplied lenses as executed in Section 9.9.

Kramer has presented a series of languages for consistency preservation [128] in the context of the Vitruvius framework [35]. From these, the

[120] Although our formalization uses state-based lenses for simplicity, the implementation actually works delta-based, which is important for multi-valued synchronization blocks.

Mapping language is closest to NMF Synchronizations as it also defines correspondences between elements that can be enforced automatically in both directions. Similar to our approach, it specifies how the properties of model elements should be synchronized based on isomorphisms and lenses. These rules may relate to base isomorphisms. However, the synchronization properties of this approach are not formally proven and it is not clear how this formalism can be implemented in an internal language.

The internal transformation language FunnyQT [110] has a bidirectional execution mode that uses the *core.logic* library that implements relational programming in Clojure. A performance comparison with FunnyQT is available in Section 9.9.

Lastly, Wider has created a bidirectional model transformation language as an internal DSL in Scala [216] that is based on lenses, similar to our approach. However, the approach uses pure lenses that are conceptually limited to tree-based structures.

Other approaches to bidirectional transformation include putback-based systems such as BiGUL [126]. Here, the forward transformation is automatically reconstructed from the backward transformation with the rationale that the latter may require more attention. In comparison to these approaches, our approach is completely symmetric and therefore also supports the synchronization of heterogeneous models. BiGUL is also implemented as an internal language in Agda and Haskell. However, these languages are no mainstream languages and therefore, the language adoption problem cannot be tackled by these languages. Section 9.9 contains a performance comparison with BiGUL.

11.2.3. Lenses

Based on the original approach by Foster et al. [65], a multitude of lens variants have been proposed such as delta-lenses [58], symmetric lenses [59] or edit-lenses [108]. An overview and a great comparison can be found by Johnson and Rosebrugh [119]. The main difference from these approaches to our notion of lenses is the different scope of application. While to the best of our knowledge, all other applications of lenses so far have applied them on a model space where an object is an entire model, we apply lenses at a mutable object space where an object is a set of object identities that share a

common global state. Essentially, we apply lenses not between models but within models. They are used as a form of model navigation that remembers where it came from such that changes can be persisted.

Because of the different application area, the problems we face are different. The objects we are working with are much simpler (model element identities instead of models), but the shared global state causes us some problems which is why we have to suitably restrict the formalism and weaken the PutGet law. As a consequence, the compositionality of lenses breaks. To solve this problem, we do not strive to see an entire model synchronization as a big lens but rather use several small lenses and combine them to obtain a model synchronization.

We believe that the idea we employed in Chapter 6, to use lenses as a generalization of model references, may also be beneficial for other bidirectional approaches such as TGGs. In essence, synchronization blocks are simply very simple TGG rules and the complexity comes in with more sophisticated lenses. The same lenses could also be made available to generalize the definition of TGG rules in that direction. However, we leave a detailed analysis up to future work.

11.2.4. Model Transformation Languages as Internal Languages

Some experiences exist with creating model transformation languages as internal languages like RubyTL [51], ScalaMTL [70], FunnyQT [110] or SDMLib[121]. The goals to use an existing language as host language are diverse and range from an easier implementation [19], reuse of the static type system [70], inherited tool support [92], reusing the expression evaluation, easier integration into the host language up to mitigated language adoption [104]. A detailed analysis on how an internal model transformation language should be designed to inherit tool support is discussed in previous work [101]. The degree in which these goals can be met depends very much on the selected host language, as e.g. tool support can only be inherited if some tool support exists but a concise syntax can usually only be achieved with host languages having a rather flexible syntax.

[121] `http://sdmlib.org/`, retrieved 27 Sep 2017

An example of the former is SDMLib [221], which provides an internal DSL embedded in Java and uses the Fujaba [154] tool internally. Transformations are specified in a fluent method chaining syntax operating on generated code for each metamodel. As the language essentially builds up a model for Fujaba, it could inherit change propagation support from Fujaba. Here, we think that our solution is more easy to use since we are working directly on the abstract syntax tree, which is transparent for the developer.

An example of an internal language that is capable of change propagation is VIATRA Query [205], which is implemented as an internal DSL in Xtend. More precisely, VIATRA Query is a tool for incremental graph pattern matching. However, transformations written in this way cannot be inversed. Furthermore, Xtend is not as popular as for example C# such that the language adoption problem remains.

11.2.5. Higher-order Transformations

Greenyer and Kindler presented an approach, in which concepts of QVT and TGGs are compared and a part of the QVT-Core is mapped to TGGs [76]. The objective of their work is firstly to identify essential concepts of a declarative model transformation by finding common concepts of QVT-Core and TGGs. Secondly, they want to show the similarities of both by mapping concepts of QVT-Core to TGGs, so that a transformation specified in QVT-Core can be transformed to TGG rules and executed by a TGG transformation engine. Thirdly, they intend to discuss how both technologies can benefit from the concepts of each other by analyzing their differences.

Richa et al.[176] transform ATL transformations into the Algebraic Graph Transformation (AGT) framework. The focus is on the declarative features of the ATL language and the HOT is used to make an easier theoretical analysis of ATL transformations possible [176]. They implemented the HOT in a Java-based tool called ATLAnalyser.

Büttner et al.[32] presented a HOT that is used for a verification of ATL transformations. The HOT is implemented as an ATL transformation that takes the ATL transformation which should be verified and transforms it into a transformation model. A transformation model is a specific kind of trace model and is used as a starting position for the verification [32]. Again the HOT only supports the declarative subset of the ATL language.

A further example is the work from Troya and Vallecillo [203] that defines a HOT in the ATL language which transforms an ATL transformation model into a model of the Maude language. Among others, the objective is the formal analysis of ATL programs.

Not exactly a HOT, but at least a slight transformation of semantics of ATL transformations was also presented by Wagelaar and others where they run ATL in the EMFTVM instead of the default VM [212] with slight adaptions in the semantics of rule inheritance, but no support for incrementality.

None of the HOTs that were mentioned in this section has the same goal as ATL2NMFs. They all have the purpose of analyzing or verifying the source transformation in contrast to ATL2NMFs that has the purpose of introducing the functionality of incremental model synchronizations for its input transformation. To the best of our knowledge, no previous work proposed a transformation into synchronization blocks.

11.3. Metamodeling Improvements

There are three fields of metamodel improvements in the literature: Other aproaches to define refinements and structural decomposition as well as conceptually different approaches to realize instantiation relationships between model elements. For the latter, Atkinson and others introduced the classification in level-adjuvant and level-blind languages [12]. While the name of this classification seems biased, it it useful to categorize the approaches. Igamberdiev and others created a survey of Deep Modeling approaches [117].

11.3.1. Refinements

The idea to use refinements for modeling is not new as in particular Back and von Wright have written a whole book on refinement calculus with a strong mathematical foundation based on lattices and set theory [17]. A usage in a model-driven context has been proposed by Varró and Patarisza in 2003 [208] or by Pons [165]. In contrast to our approach, they break with existing modeling paradigms. Furthermore, they do not seem to enforce the refinements through the type system.

The specifications of UML and CMOF also know refinements, as redefinitions and subsets. However, the actual semantics of these constructs is not detailed in the specification [151].

In particular, UML defines three methods to refine associations: redefinition, specialization and subsetting, though as mentioned, the semantics and especially their interplay are not clearly defined. In particular, these definitions have some correctness problems in connection with other constraints such as multiplicity constraints as shown by Maraee and Balaban [144, 143]. To some degree, our approach tackles the necessity of the three of those: Our implementation of refinements matches redefinition quite closely, but we show that through the interplay of refinements of the same features, we can support many more modeling scenarios. We also consider the interconnection of refinements with multiple inheritance.

There have been a couple of works to define the semantics of UML association refinements through OCL constraints [155, 48, 80]. Closest to our approach, Nieto et al. [155] propose a semantics for association redefinition and use a similar notation. However, they also implement refinements through a constraint of the more general reference and therefore do not inherit type-system guarantees same as we do. Furthermore, their approach is limited in that only a single reference may refine another reference.

Further, tools that use OCL to check the validity of models are usually dynamic in the sense that they represent models and metamodels in memory. Our approach uses a generative approach where code is generated from a metamodel to represent models in memory. A generative approach usually has a faster model API at the cost of higher maintenance efforts.

We are not aware of any solutions that considered UML redefinition in combination with diamond-shaped inheritance and how these situations can be resolved.

11.3.2. Level-adjuvant languages

Level-adjuvant approaches typically use a level-agnostic meta-metamodel [14] describing the model structure. Many of these approaches are much more mature than ours and already provide rich tool support [11, 54]. However, we believe the case of composite components as in PCM is a case that inherently

asks for support for level-crossing references as we presented in Chapter 8 which is typically not supported by level-adjuvant languages.

On the other hand, Rossini et al. have created a comparison where they compare the level-adjuvant MetaDepth language with strict two-level modeling for cloud-based applications [177]. We believe that their solution using potencies is much simpler than a solution using structural decomposition could be, mainly because the potencies allow attributes clear when designing the first level to span multiple modeling levels more easily. Thus, there seems to be a trade-off decision between our approach and level-adjuvant languages when to apply which strategy.

11.3.3. Level-blind languages

In contrast to level-adjuvant languages, level-blind languages have no explicit notion of levels. Rather, they are helping constructs as the result of stratification. As our approach shows, this still allows to cross the boundaries of modeling levels. Atkinson et al. have presented some paradoxa for level-blind languages [12] for which they regarded these approaches to be logically inconsistent but because our approach requires an explicit mapping from a clabject to the platform type system, paradox situations like the quoted phenomena of being his own baby simply do not apply to our approach.

Our approach is not the first one to be level-less. Henderson, Clark and Gonzalez-Perez have been working on an approach involving basically only objects and slots [88, 46]. However, this approach is not compatible to existing two-level metamodels such as Ecore and less convenient in terms of a generated API.

11.3.4. Deep Modeling compatible with EMOF

We supported Deep Modeling directly through a small set of non-invasive extensions of the EMOF modeling standard. However, there have been other approaches with the common goal to reuse tooling for EMOF metamodels in the past, typically the tooling provided in the EMF ecosystem.

The earliest we are aware of are Kainz and others [123] who propose a model transformation from a model to a metamodel. A similar approach is used by

Macías and others [141] that generate the metamodel from a higher level instance model.

Kimura and others [124] have a slightly different approach as they essentially designed an object-slot-value metamodel very similar to the approach from Hendersen, Clark and Gonzales-Perez [88, 46].

11.3.5. Aspect-oriented modeling

Refinements are only one possibility to simplify the modeling of recurring patterns. Another possibility is to model the pattern once and very explicit, including possible constraints that have to be implemented, and weave this pattern implementation into a concrete use case using aspect-oriented modeling techniques. An example is CORE[122]. However, once applied, concerns have a fixed level of abstraction, while refinements allow to model multiple levels of abstraction concurrently. In our scenario, this is important in the case of bought components where their inner structure is not known.

[122] Concern-Oriented REuse [181]

12. Limitations and Future Work

In this chapter, important limitations of the presented approaches are listed that will constitute future work. For this, the remainder of this chapter mimics the contents of the thesis and each section discusses the limitations and future work based on a given chapter.

12.1. Incremental Model Analyses

As explained in Section 10.1, there are many methods, particularly in queries, that are very efficient to process in a batch implementation, but very inefficient in an incremental setting. However, because the initialization time of an incremental analysis is rather unimportant in many applications, this opens space for automatic query optimization. In particular, we want to review how the research results from the automatic optimization of database queries can be used for the optimization of incremental queries.

Further, the validation of the incremental model analysis approach presented in this thesis is currently done using rather simple analyses. This makes it unclear what the maximum complexity of an analysis is such that incrementalization yields efficient results. Therefore, we want to apply the incrementalization system to larger case studies. The results that were achieved so far look promising such that also more complex analyses may profit from implicit incrementalization techniques.

So far, there is also hardly a validation against Self-adjusting Computation or the Adapton system. This is partially due to the sparse availability of these tools and fact that the most recent implementation of Adapton is based on Rust that makes it hard to implement model-driven benchmarks for its intentional lack of inheritance concepts. Here, a comparison is left open for future work.

Lastly, meanwhile our evaluation of the distributed computing approach has let us gain a fair understanding of the performance characteristics of this approach, it also suggested that the Train Benchmark was not a good example of when to apply this approach, particularly because the analyses and their predicates were rather simple. Therefore, we want to evaluate this approach also in a larger case study and compare it with IncQuery-D.

The distributed version of incremental computation currently also suffers from the fact that the models themselves cannot be distributed but must be present in all silos. This also limits the maximum model size that can be processed as each copy of the model has to fit into each nodes main memory. However, distributing a model in NMF is complicated at the moment as NMF has no support for a proxy mechanism such as present in EMF. Such a proxy mechanism is also subject of future work.

12.2. Contraction of Dependency Graphs

The evaluation results performed in Section 9.3.5 only give a first (promising) impression on what one is able to achieve using the automated optimization of incremental queries, particularly once again because the query predicates are rather simple. An evaluation of the dependency graph contraction in a larger case study is left as future work.

A larger case study then likely raises the need for an improvement of the genetic optimization strategy suggested in Section 5.5 and better heuristics to prune the search space. The proposed genetic algorithm rather is a proof of concept. Perhaps also other optimization techniques such as a Simplex algorithm achieve better results.

12.3. Incremental Model Transformations

Currently, we assume in our implementation that a correspondence between model elements once established will not change during the lifecycle of both objects. This has an impact mainly on synchronization rule instantiation in the presence of filter conditions.

For example, if in the *Families2Persons* case study the gender of an object should change, for example because in the family model, a member is moved from a *father* reference to a *mother* reference, this is currently not supported by NMF Synchronizations. The implementation actually does allow filters to be set on synchronization rule instantiations, but the current engine implementation throws an exception as soon as any change would cause the correspondence relation to break or silently ignores the error. This issue can of course be worked around by simply creating separate synchronization rules but the lack of a generalized isomorphism may hamper the conciseness in other parts of the model synchronization.

Though the implementation currently does not support these cases, this is only a technical problem. The formalization is fine with this: The synchronization blocks of the broken correspondence have to stop pushing incremental updates and the new correspondence link has to be created.

To solve the technical problem, we plan to extend NMF with a generic replace operator. This operator will also replace the `Delete` method as a deletion simply is the same as a replacement by `null`. It is further required for proxy elements such that the proxies can easily be replaced by the actual model elements.

12.4. Meta-metamodel extensions and Deep Modeling

The meta-metamodel extensions proposed in this thesis are only validated using a comparison to existing modeling technologies but it yet unclear how well users understand and accept these new metamodeling capabilities. Furthermore, besides that these extensions make several model analyses obsolete, it is interesting to see whether and how these extensions have an influence on model analyses that are still required.

To reason on the acceptance of the modeling extension, we are planning to create a modeling tool that allows to create models graphically. So far, all models using these modeling extensions have been created programatically. Such an editor would of course also allow to create metamodels without the proposed meta-metamodel extensions, opening an opportunity to compare

metamodels created by developers with or without these extensions. However, also a comparison with other modeling tools would be interesting.

After all, the EMOF standard only is a post-priori standardization of the MOF subset implemented in EMF.

13. Conclusion

This thesis presents several approaches to reduce the time from a model change to an updated model analysis or transformation results with a minimal set of changes necessary for the developer. Rather, an incremental evaluation strategy is deduced automatically from the batch specification of the analysis or from the declarative specification of the model transformation. For the first time, this incremental derivation process is available in a mainstream general-purpose programming language where good tool support for the developer is available. Furthermore, the incrementalization process is formally described in terms of category and this formalization allows to prove important correctness properties.

The presented approaches optimize incremental execution using a dynamic dependency graph in multiple ways: For the first time, the incrementalization system is able to reuse explicit knowledge how commonly used functions can be executed incrementally, it can contract the dynamic dependency graph in the presence of complex domain logic and automatically decide for which parts of an analysis a fine-grained incrementalization yields the best performance.

Applied to a query framework, the explicit incrementalization shows very good results in multiple case studies where the incremental execution is often an order of magnitude faster than other incremental tools or multiple orders of magnitude faster than repetitive batch execution. Furthermore, the results indicate that our implicit approach reusing the C# query syntax for developers to specify queries is very understandable.

The presented model transformation approach reuses this incrementalization system to enable fully incremental model transformations. The correctness and hippocraticness of the inconsistency repair operator used in these model transformations is formally proven using the newly introduced algebraic construct of synchronization blocks. As model transformations based on the declarative part of the ATL transformation language – considerably the most

common model transformation language in the community [203] – can be transformed into unidirectional synchronization blocks, the transformation language presented in the thesis is also widely applicable. In a community benchmark for bidirectional model transformation languages, we could confirm the applicability and achieved the best incremental performance by multiple orders of magnitude. Meanwhile, the implementation of this language as an internal language in C# also means that a very good tool support is available.

Further, the evaluation shows that the model transformation approach is very concise. However, the analysis of the understandability suggests that the comprehension of model transformations created with NMF Synchronizations requires an understanding of the declarative usage of the C# programming language and the underlying theory of synchronization blocks.

Taken together, the presented approaches are able to execute a wide range of model processing operations incrementally without or with very few changes by the developers to the batch specification. However, this incrementalization still has to remain in the borders of the metamodel.

To shift these borders without the need of radically new meta-metamodels, we proposed several meta-metamodel extensions. These extensions make multiple classes of model analyses obsolete because the analysis result can be guaranteed by the underlying platform type system. In other cases, we could show that much less changes have to be considered to update a given analysis when models contain instantiation relationships between model elements. Because the presented Deep Modeling approach integrates seamlessly with traditional two-level modeling as it only requires a couple of non-invasive modeling extensions, we are able to reuse our results on incremental model analyses and transformations. This is a unique feature compared with any other Deep Modeling approach.

Taken together, all these approaches make it possible to update analysis and transformation results very efficiently once the input model of these artifacts changes. In many cases, in particular in model transformations, the time to obtain these updated results depends only on the size of the change rather than the size of the model and is therefore much faster than repetitive execution. Meanwhile, the understandability of these artifacts is not affected as the incrementalization works implicitly.

Bibliography

[1] Umut A Acar. "Self-Adjusting Computation". PhD thesis. Princeton University, 2005.

[2] Umut A Acar. "Self-adjusting computation:(an overview)". In: *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*. ACM. 2009, pp. 1–6.

[3] Umut A Acar, Guy Blelloch, Ruy Ley-Wild, Kanat Tangwongsan, and Duru Turkoglu. "Traceable data types for self-adjusting computation". In: *ACM Sigplan Notices*. Vol. 45. 6. ACM. 2010, pp. 483–496.

[4] Gul A Agha. *Actors: A model of concurrent computation in distributed systems*. Tech. rep. DTIC Document, 1985.

[5] David H Akehurst, W Gareth J Howells, Markus Scheidgen, and Klaus D McDonald-Maier. "C# 3.0 makes OCL redundant". In: *Electronic Communications of the EASST* 9 (2008).

[6] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560. URL: http://doi.acm.org/10.1145/1465482.1465560.

[7] Anthony Anjorin, Thomas Buchmann, and Bernhard Westfechtel. "The Families to Persons Case". In: *Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) federation of conferences*. Ed. by Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava. CEUR Workshop Proceedings. Marburg, Germany: CEUR-WS.org, July 2017.

[8] Anthony Anjorin, Marius Lauder, Sven Patzina, and Andy Schürr. "eMoflon: Leveraging EMF and Professional CASE Tools". In: *3. Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe2011)*. Lecture Notes in Informatics. 2011.

[9] DLMS User Association. “Excerpt from companion specification for energy metering COSEM interface classes and OBIS identification system”. In: (Sept. 2014).

[10] Colin Atkinson. “Meta-modelling for distributed object environments”. In: *Enterprise Distributed Object Computing Workshop [1997]. EDOC’97. Proceedings. First International*. IEEE. 1997, pp. 90–101.

[11] Colin Atkinson and Ralph Gerbig. “Melanie: multi-level modeling and ontology engineering environment”. In: *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards*. ACM. 2012, p. 7.

[12] Colin Atkinson, Ralph Gerbig, and Thomas Kühne. “Comparing Multi-Level Modeling Approaches”. In: *MULTI 2014–Multi-Level Modelling Workshop Proceedings* (2014), p. 53.

[13] Colin Atkinson, Ralph Gerbig, and Christian Tunjic. “Towards multi-level aware model transformations”. In: *Theory and Practice of Model Transformations*. Springer Berlin Heidelberg, 2012, pp. 208–223.

[14] Colin Atkinson and Thomas Kühne. “Meta-level independent modelling”. In: *International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming* (2000), pp. 12–16.

[15] Colin Atkinson and Thomas Kühne. “The essence of multilevel metamodeling”. In: *≪ UML ≫ 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. Springer Berlin Heidelberg, 2001, pp. 19–33.

[16] *ATL User Guide*. `http://wiki.eclipse.org/ATL/User_Guide_-_Introduction`. [Last accessed on 2016-07-05].

[17] Ralph-Johan Back and Joakim Von Wright. *Refinement calculus: a systematic introduction*. springer Heidelberg, 1998.

[18] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. “A Survey on Reactive Programming”. In: *ACM Comput. Surv.* 45.4 (Aug. 2013), 52:1–52:34. ISSN: 0360-0300. DOI: `10.1145/2501654.2501666`. URL: `http://doi.acm.org/10.1145/2501654.2501666`.

[19] Howard Barringer and Klaus Havelund. *TraceContract: A Scala DSL for trace analysis*. Springer, 2011.

[20] Olivier Beaudoux, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. "Active Operations on Collections". In: *Model Driven Engineering Languages and Systems: 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*. Ed. by Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 91–105. ISBN: 978-3-642-16145-2. DOI: 10.1007/978-3-642-16145-2_7. URL: http://dx.doi.org/10.1007/978-3-642-16145-2_7.

[21] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software* 82 (2009), pp. 3–22. DOI: 10.1016/j.jss.2008.03.066.

[22] Mordechai Ben-Menachem and Garry S Marliss. *Software Quality: Producing Practical, Consistent Software*. International Thomson Computer Press NY, USA, 1997.

[23] Gábor Bergmann. "Translating OCL to graph patterns". In: *Model-Driven Engineering Languages and Systems*. Springer, 2014, pp. 670–686.

[24] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. "Viatra 3: A Reactive Model Transformation Platform". In: *Theory and Practice of Model Transformations*. Springer, 2015, pp. 101–110.

[25] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. "Incremental evaluation of model queries over EMF models". In: *Model Driven Engineering Languages and Systems*. Springer, 2010, pp. 76–90.

[26] Gábor Bergmann, András Ökrös, István Ráth, Dániel Varró, and Gergely Varró. "Incremental pattern matching in the VIATRA model transformation system". In: *Proceedings of the third international workshop on Graph and model transformations*. ACM. 2008, pp. 25–32.

[27] Alexis Tobias Bernhard. "Deep Modeling für Palladio". Bachelor Thesis. Karlsruhe Institute of Technology, 2016.

[28] Philip A. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. *Orleans: Distributed Virtual Actors for Programmability and Scalability*. Tech. rep. MSR-TR-2014-41. Mar. 2014. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=210931.

[29] Jose A Blakeley, Per-Ake Larson, and Frank Wm Tompa. "Efficiently updating materialized views". In: *ACM SIGMOD Record*. Vol. 15. 2. ACM. 1986, pp. 61–71.

[30] Franz Brosch, Heiko Koziolek, Barbora Buhnova, and Ralf Reussner. "Architecture-based Reliability Prediction with the Palladio Component Model". In: *IEEE Transactions on Software Engineering* 38.6 (Nov. 2012), pp. 1319–1339. ISSN: 0098-5589. DOI: 10.1109/TSE.2011.94.

[31] Thomas Buchmann and Sandra Greiner. "Handcrafting a Triple Graph Transformation System to Realize Round-trip Engineering Between UML Class Models and Java Source Code". In: *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 2: ICSOFT-PT, Lisbon, Portugal, July 24 - 26, 2016*. 2016, pp. 27–38. DOI: 10.5220/0005957100270038. URL: http://dx.doi.org/10.5220/0005957100270038.

[32] Fabian Buettner, Marina Egea, Jordi Cabot, and Martin Gogolla. "Verification of ATL Transformations Using Transformation Models and Model Finders". In: *ICFEM 2012: 14th International Conference on Formal Engineering Methods*. Kyoto, Japan, Nov. 2012.

[33] Sebastian Burckhardt, Daan Leijen, Caitlin Sadowski, Jaeheon Yi, and Thomas Ball. "Two for the price of one: a model for parallel and incremental computation". In: *ACM SIGPLAN Notices*. Vol. 46. 10. ACM. 2011, pp. 427–444.

[34] E. Burger and B. Gruschko. "A Change Metamodel for the Evolution of MOF-based Metamodels". In: *Proceedings of Modellierung*. Vol. 161. 2010, pp. 285–300.

[35] Erik Burger. "Flexible Views for View-based Model-driven Development". PhD thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology, July 2014. ISBN: 978-3-7315-0276-0. DOI: 10.5445/KSP/1000043437. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000043437.

[36] Erik Burger, Jörg Henß, Martin Küster, Steffen Kruse, and Lucia Happe. "View-Based Model-Driven Software Development with ModelJoin". In: *Software & Systems Modeling* 15.2 (2014). Ed. by Robert France and Bernhard Rumpe, pp. 472–496. ISSN: 1619-1374. DOI: 10.1007/s10270-014-0413-5.

[37] Erik Burger, Victoria Mittelbach, and Anne Koziolek. "View-based and Model-driven Outage Management for the Smart Grid". In: *Proceedings of the 11th Workshop on Models@run.time co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016)*. Saint Malo, France: CEUR Workshop Proceedings, Oct. 2016. URL: http://ceur-ws.org/Vol-1742.

[38] Sergey Bykov, Alan Geller, Gabriel Kliot, James Larus, Ravi Pandya, and Jorgen Thelin. *Orleans: A Framework for Cloud Computing*. Tech. rep. MSR-TR-2010-159. Nov. 2010. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=141999.

[39] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. "Orleans: Cloud Computing for Everyone". In: *Proceedings of the 2Nd ACM Symposium on Cloud Computing*. SOCC '11. Cascais, Portugal: ACM, 2011, 16:1–16:14. ISBN: 978-1-4503-0976-9. DOI: 10.1145/2038916.2038932. URL: http://doi.acm.org/10.1145/2038916.2038932.

[40] Jordi Cabot and Ernest Teniente. "Incremental integrity checking of UML/OCL conceptual schemas". In: *Journal of Systems and Software* 82.9 (2009), pp. 1459–1478.

[41] Yufei Cai, Paolo G Giarrusso, Tillmann Rendel, and Klaus Ostermann. "A Theory of Changes for Higher-Order Languages". In: *ACM SIGPLAN Notices*. Vol. 49. 6. ACM. 2014, pp. 145–155.

[42] Magnus Carlsson. "Monads for incremental computing". In: *ACM SIGPLAN Notices* 37.9 (2002), pp. 26–35.

[43] Yan Chen, Joshua Dunfield, Matthew A Hammer, and Umut A Acar. "Implicit self-adjusting computation for purely functional programs". In: *Journal of Functional Programming* 24.01 (2014), pp. 56–112.

[44] BettyH.C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, HolgerM. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, HausiA. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. "Software Engineering for Self-Adaptive Systems: A Research

Roadmap". English. In: *Software Engineering for Self-Adaptive Systems*. Ed. by BettyH.C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee. Vol. 5525. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 1–26. DOI: 10.1007/978-3-642-02161-9_1.

[45] Kiyoung Choi, Sun Young Hwang, and Tom Blank. "Incremental-in-time algorithm for digital simulation". In: *Proceedings of the 25th ACM/IEEE Design Automation Conference*. IEEE Computer Society Press. 1988, pp. 501–505.

[46] Tony Clark, Cesar Gonzalez-Perez, and Brian Henderson-Sellers. "A Foundation for Multi-Level Modelling". In: *MULTI 2014–Multi-Level Modelling Workshop Proceedings*. 2014, p. 43.

[47] Gregory H Cooper and Shriram Krishnamurthi. "Embedding dynamic dataflow in a call-by-value language". In: *ESOP*. Vol. 3924. Springer. 2006, pp. 294–308.

[48] Dolors Costal, Cristina Gómez, and Giancarlo Guizzardi. "Formal Semantics and Ontological Analysis for Understanding Subsetting, Specialization and Redefinition of Associations in UML". In: *Conceptual Modeling – ER 2011: 30th International Conference, ER 2011, Brussels, Belgium, October 31 - November 3, 2011. Proceedings*. Ed. by Manfred Jeusfeld, Lois Delcambre, and Tok-Wang Ling. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 189–203. ISBN: 978-3-642-24606-7. DOI: 10.1007/978-3-642-24606-7_15. URL: http://dx.doi.org/10.1007/978-3-642-24606-7_15.

[49] Roy L Crole. *Categories for types*. Cambridge University Press, 1993.

[50] Jesús Sánchez Cuadrado, Jesús García Molina, and Marcos Menárguez Tortosa. "RubyTL: A Practical, Extensible Transformation Language". In: *Proceedings of the Second European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA '06)*. Ed. by Arend Rensink and Jos Warmer. Vol. 4066. Lecture Notes in Computer Science. Springer, 2006, pp. 158–172. ISBN: 3-540-35909-5.

[51] Jesús Sánchez Cuadrado, Jesús García Molina, and Marcos Menarguez Tortosa. "Rubytl: A practical, extensible transformation language". In: *Model Driven Architecture–Foundations and Applications*. Springer. 2006, pp. 158–172.

[52] Krzysztof Cwalina and Brad Abrams. *Framework design guidelines: conventions, idioms, and patterns for reusable. net libraries*. Pearson Education, 2008.

[53] Krzysztof Czarnecki and Simon Helsen. "Classification of Model Transformation Approaches". In: *OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture*. Last retrieved 2008-01-06. Oct. 2003. URL: http://www.softmetaware.com/oopsla2003/czarnecki.pdf.

[54] Juan De Lara and Esther Guerra. "Deep meta-modelling with metadepth". In: *Objects, Models, Components, Patterns*. Springer, 2010, pp. 1–20.

[55] Rogério De Lemos, Holger Giese, Hausi A Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M Villegas, Thomas Vogel, et al. "Software engineering for self-adaptive systems: A second research roadmap". In: *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 1–32.

[56] Zinovy Diskin. "Algebraic Models for Bidirectional Model Synchronization". In: *Model Driven Engineering Languages and Systems*. Ed. by Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter. Vol. 5301. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, pp. 21–36. ISBN: 978-3-540-87874-2.

[57] Zinovy Diskin. "Model Synchronization: Mappings, Tiles, and Categories". In: *Generative and Transformational Techniques in Software Engineering III*. Ed. by João Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva. Vol. 6491. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, pp. 92–165. ISBN: 978-3-642-18022-4.

[58] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. "From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case". In: *Journal of Object Technology* 10 (2011), 6:1–25. ISSN: 1660-1769. DOI: 10.5381/jot.2011.10.1.a6. URL: http://www.jot.fm/contents/issue_2011_01/article6.html.

[59] Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann, and Fernando Orejas. "From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case". English. In: *Model Driven Engineering Languages and Systems*. Ed. by Jon Whittle, Tony Clark, and Thomas Kühne. Vol. 6981. Lecture Notes

in Computer Science. Springer Berlin Heidelberg, 2011, pp. 304–318. ISBN: 978-3-642-24484-1. DOI: 10.1007/978-3-642-24485-8_22. URL: http://dx.doi.org/10.1007/978-3-642-24485-8_22.

[60] Jonathan Edwards. "Coherent Reaction". In: *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. OOPSLA '09. Orlando, Florida, USA: ACM, 2009, pp. 925–932. ISBN: 978-1-60558-768-4. DOI: 10.1145/1639950.1640058. URL: http://doi.acm.org/10.1145/1639950.1640058.

[61] Sven Efftinge and Markus Völter. "oAW xText: A framework for textual DSLs". In: *Eclipsecon Summit Europe 2006*. Nov. 2006. URL: http://eclipsecon.org/summiteurope2006/presentations/ESE2006-EclipseModelingSymposium12_xTextFramework.pdf.

[62] Christoph Eickhoff, Simon-Lennert Räsch, and Philipp Kolodziej. "Yage Solution to the Family to Persons Case of the TTC'17". In: *Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) federation of conferences*. Ed. by Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava. CEUR Workshop Proceedings. Marburg, Germany: CEUR-WS.org, July 2017.

[63] Charles L Forgy. "Rete: A fast algorithm for the many pattern/many object pattern match problem". In: *Artificial intelligence* 19.1 (1982), pp. 17–37.

[64] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. "Combinators for bi-directional tree transformations: a linguistic approach to the view update problem". In: *SIGPLAN Not.* 40.1 (Jan. 2005), pp. 233–246. ISSN: 0362-1340. DOI: 10.1145/1047659.1040325.

[65] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. "Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29.3 (May 2007). ISSN: 0164-0925. DOI: 10.1145/1232420.1232424. URL: http://doi.acm.org/10.1145/1232420.1232424.

[66] Martin Fowler. *Domain-specific languages*. Addison-Wesley Professional, 2010.

[67] Martin Fowler and Rebecca Parsons. *Domain Specific Languages*. 1st. Addison-Wesley, Reading, MA, USA, 2010. ISBN: 9780321712943.

[68] Gael Fraiteur. "User-friendly aspects with compile-time imperative semantics in .NET: an overview of PostSharp". In: *Seventh International Conference on Aspect-Oriented Software Development (AOSD)*. 2008.

[69] Antonio Garcia-Dominguez. *TTC'16 Live Contest Case Study – Execution of dataflow-based model transformations*. `https://github.com/TransformationToolContest/ttc2016-live/blob/master/description/description.pdf`. 2016.

[70] Lars George, Arif Wider, and Markus Scheidgen. "Type-Safe model transformation languages as internal DSLs in scala". In: *Theory and Practice of Model Transformations*. Springer, 2012, pp. 160–175.

[71] Paolo G Giarrusso, Klaus Ostermann, Michael Eichberg, Ralf Mitschke, Tillmann Rendel, and Christian Kästner. "Reify your collection queries for modularity and speed!" In: *Proceedings of the 12th annual international conference on Aspect-oriented software development*. ACM. 2013, pp. 1–12.

[72] Holger Giese and Stephan Hildebrandt. *Efficient model synchronization of large-scale models*. Tech. rep. 28. Hasso-Pleitner-Institut für Softwaresystemtechnik, 2009.

[73] Holger Giese and Robert Wagner. "From model transformation to incremental bidirectional model synchronization". In: *Software and Systems Modeling* 8 (1 2009), pp. 21–43. ISSN: 1619-1366.

[74] Holger Giese and Robert Wagner. "Incremental Model Synchronization with Triple Graph Grammars". In: *Model Driven Engineering Languages and Systems*. Ed. by Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio. Vol. 4199. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, pp. 543–557. ISBN: 978-3-540-45772-5.

[75] Holger Giese and Robert Wagner. "Incremental model synchronization with triple graph grammars". In: *Model Driven Engineering Languages and Systems*. Springer, 2006, pp. 543–557.

[76] Joel Greenyer and Ekkart Kindler. "Reconciling TGGs with QVT". In: *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2007, September 30 - October 5, 2007, Nashville, USA, LNCS*. Ed. by G. Engels, B. Opdyke, D.C. Schmidt, and F. Weil. Vol. 4735. Lecture Notes in Computer Science. Springer Verlag Berlin/Heidelberg, 2007, pp. 16–30.

[77] Torsten Grust. *Monad comprehensions: a versatile representation for queries*. Springer, 2004.

[78] Olaf Gunkel, Matthias Schmidt, and Albert Zündorf. "The SDMLib Solution to the Java Refactoring Case for TTC2015". In: *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L'Aquila, Italy, July 24, 2015*. 2015, pp. 123–127. URL: http://ceur-ws.org/Vol-1524/paper17.pdf.

[79] Ashish Gupta, Inderpal Singh Mumick, et al. "Maintenance of materialized views: Problems, techniques, and applications". In: *IEEE Data Eng. Bull.* 18.2 (1995), pp. 3–18.

[80] Lars Hamann and Martin Gogolla. "Endogenous Metamodeling Semantics for Structural UML 2 Concepts". In: *Model-Driven Engineering Languages and Systems: 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 – October 4, 2013. Proceedings*. Ed. by Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 488–504. ISBN: 978-3-642-41533-3. DOI: 10.1007/978-3-642-41533-3_30. URL: http://dx.doi.org/10.1007/978-3-642-41533-3_30.

[81] Matthew A Hammer, Umut A Acar, and Yan Chen. "CEAL: a C-based language for self-adjusting computation". In: *ACM Sigplan Notices*. Vol. 44. 6. ACM. 2009, pp. 25–37.

[82] Matthew A Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S Foster, Michael Hicks, and David Van Horn. "Incremental computation with names". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM. 2015, pp. 748–766.

[83] Matthew A Hammer, Georg Neis, Yan Chen, and Umut A Acar. "Self-adjusting stack machines". In: *ACM SIGPLAN Notices*. Vol. 46. 10. ACM. 2011, pp. 753–772.

[84] Matthew A Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S Foster. "Adapton: Composable, demand-driven incremental computation". In: *ACM SIGPLAN Notices*. Vol. 49. 6. ACM. 2014, pp. 156–166.

[85] David Hearnden, Michael Lawley, and Kerry Raymond. "Incremental model transformation for the evolution of model-driven systems". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2006, pp. 321–335.

[86] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. *Jamopp: The java model parser and printer*. Techn. Univ., Fakultät Informatik, 2009.

[87] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. *Jamopp: The java model parser and printer*. Tech. rep. 2009.

[88] Brian Henderson-Sellers, Tony Clark, and Cesar Gonzalez-Perez. "On the search for a level-agnostic modelling language". In: *Advanced Information Systems Engineering*. Springer. 2013, pp. 240–255.

[89] Brian Henderson-Sellers and Cesar Gonzalez-Perez. "Connecting Powertypes and Stereotypes." In: *Journal of Object Technology* 4.7 (2005), pp. 83–96.

[90] Soichiro Hidaka, Massimo Tisi, Jordi Cabot, and Zhenjiang Hu. "Feature-based classification of bidirectional transformation approaches". In: *Software & Systems Modeling* (2015), pp. 1–22. ISSN: 1619-1374. DOI: 10.1007/s10270-014-0450-0. URL: http://dx.doi.org/10.1007/s10270-014-0450-0.

[91] Stephan Hildebrandt, Leen Lambers, Holger Giese, Jan Rieke, Joel Greenyer, Wilhelm Schäfer, Marius Lauder, Anthony Anjorin, and Andy Schürr. "A survey of triple graph grammar tools". In: *Electronic Communications of the EASST* 57 (2013).

[92] Georg Hinkel. "An approach to maintainable model transformations using an internal DSL". MA thesis. Karlsruhe Institute of Technology, Oct. 2013.

[93] Georg Hinkel. "An NMF solution to the Families to Persons case at the TTC 2017". In: *Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) federation of conferences*. Ed. by Antonio Garcia-Dominguez,

Georg Hinkel, and Filip Krikava. CEUR Workshop Proceedings. Marburg, Germany: CEUR-WS.org, July 2017.

[94] Georg Hinkel. "An NMF Solution to the Java Refactoring Case". In: *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences*. Ed. by Louis Rose, Tassilo Horn, and Filip Krikava. Vol. 1524. CEUR Workshop Proceedings. L'Aquila, Italy: CEUR-WS.org, July 2015, pp. 95–99.

[95] Georg Hinkel. "An NMF solution to the Smart Grid Case at the TTC 2017". In: *Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) federation of conferences*. Ed. by Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava. CEUR Workshop Proceedings. Marburg, Germany: CEUR-WS.org, July 2017.

[96] Georg Hinkel. *NMF: A Modeling Framework for the .NET Platform*. Tech. rep. Karlsruhe: Karlsruhe Institute of Technology, 2016. URL: http://nbn-resolving.org/urn:nbn:de:swb:90-537082.

[97] Georg Hinkel. "The TTC 2017 Outage System Case for Incremental Model Views". In: *Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) federation of conferences*. Ed. by Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava. CEUR Workshop Proceedings. Marburg, Germany: CEUR-WS.org, July 2017.

[98] Georg Hinkel and Erik Burger. "Change Propagation and Bidirectionality in Internal Transformation DSLs". In: *Software & Systems Modeling* 18.1 (2019), pp. 249–278. ISSN: 1619-1374. DOI: 10.1007/s10270-017-0617-6. URL: http://rdcu.be/u9PT.

[99] Georg Hinkel, Kiana Busch, and Robert Heinrich. "Refinements and Structural Decompositions in Generated Code". In: *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*. Funchal, Portugal, Jan. 2018. URL: http://sdqweb.ipd.kit.edu/publications/pdfs/hinkel2018b.pdf.

[100] Georg Hinkel and Thomas Goldschmidt. "Tool Support for Model Transformations: On Solutions using Internal Languages". In: *Modellierung 2016*. Karlsruhe, Germany, Mar. 2016.

[101] Georg Hinkel, Thomas Goldschmidt, Erik Burger, and Ralf Reussner. "Using Internal Domain-Specific Languages to inherit Tool Support and Modularity for Model Transformations". In: *Software & Systems Modeling* 18.1 (2019), pp. 129–155. ISSN: 1619-1374. DOI: 10.1007/s10270-017-0578-9. URL: http://rdcu.be/oTED.

[102] Georg Hinkel, Thomas Goldschmidt, and Lucia Happe. "An NMF solution for the Flowgraphs case at the TTC 2013". In: *Proceedings Sixth Transformation Tool Contest, TTC 2013, Budapest, Hungary, 19-20 June, 2013.* 2013, pp. 37–42. DOI: 10.4204/EPTCS.135.5. URL: https://doi.org/10.4204/EPTCS.135.5.

[103] Georg Hinkel, Thomas Goldschmidt, and Lucia Happe. "An NMF solution for the Petri Nets to State Charts case study at the TTC 2013". In: *Proceedings Sixth Transformation Tool Contest, TTC 2013, Budapest, Hungary, 19-20 June, 2013.* 2013, pp. 95–100. DOI: 10.4204/EPTCS.135.12. URL: https://doi.org/10.4204/EPTCS.135.12.

[104] Georg Hinkel, Henning Groenda, Lorenzo Vannucci, Oliver Denninger, Nino Cauli, and Stefan Ulbrich. "A Domain-Specific Language (DSL) for Integrating Neuronal Networks in Robot Control". In: *2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*. 2015.

[105] Georg Hinkel and Lucia Happe. "An NMF Solution to the TTC Train Benchmark Case". In: *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences*. Ed. by Louis Rose, Tassilo Horn, and Filip Krikava. Vol. 1524. CEUR Workshop Proceedings. L'Aquila, Italy: CEUR-WS.org, July 2015, pp. 142–146.

[106] Georg Hinkel and Lucia Happe. "Using component frameworks for model transformations by an internal DSL". In: *Proceedings of the 1st International Workshop on Model-Driven Engineering for Component-Based Software Systems co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014)*. Vol. 1281. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pp. 6–15.

[107] Kevin Hoffman. "Continuous LINQ". In: *Dr Dobbs Journal* 33.2 (2008), pp. 55–58.

[108] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. "Edit Lenses". In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. Philadelphia, PA, USA: ACM, 2012, pp. 495–508. ISBN: 978-1-4503-1083-3. DOI: 10.1145/2103656.2103715. URL: http://doi.acm.org/10.1145/2103656.2103715.

[109] Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity". In: *Journal of the ACM (JACM)* 48.4 (2001), pp. 723–760.

[110] Tassilo Horn. "Model Querying with FunnyQT". In: *Theory and Practice of Model Transformations*. Springer, 2013, pp. 56–57.

[111] Tassilo Horn. "Solving the Petri-Nets to Statecharts Transformation Case with FunnyQT". In: *EPTCS* (2013).

[112] Tassilo Horn. "Solving the TTC Families to Persons Case with FunnyQT". In: *Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) federation of conferences*. Ed. by Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava. CEUR Workshop Proceedings. Marburg, Germany: CEUR-WS.org, July 2017.

[113] Tassilo Horn. "Solving the TTC Java Refactoring Case with FunnyQT". In: *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L'Aquila, Italy, July 24, 2015*. 2015, pp. 83–87. URL: http://ceur-ws.org/Vol-1524/paper14.pdf.

[114] Tassilo Horn. "Solving the TTC Train Benchmark Case with FunnyQT". In: *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L'Aquila, Italy, July 24, 2015*. 2015, pp. 147–151. URL: http://ceur-ws.org/Vol-1524/paper15.pdf.

[115] "IEC 61850 Communication networks and systems for power utility automation". In: (July 2015).

[116] "IEC 61970 Energy management system application program interface (EMS-API) - Part 301 Common Information Model (CIM) Base". In: (Aug. 2011).

[117] Muzaffar Igamberdiev, Georg Grossmann, and Markus Stumptner. "A Feature-based Categorization of Multi-Level Modeling Approaches and Tools." In: *Proceedings of the 3rd International Workshop on Multi-Level Modelling co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2016), Saint-Malo, France, October 4, 2016.* 2016, pp. 45–55.

[118] Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N. Smith. "Property Models: From Incidental Algorithms to Reusable Components". In: *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. GPCE '08. Nashville, TN, USA: ACM, 2008, pp. 89–98. ISBN: 978-1-60558-267-2. DOI: 10.1145/1449913.1449927. URL: http://doi.acm.org/10.1145/1449913.1449927.

[119] Michael Johnson and Robert D. Rosebrugh. "Unifying Set-Based, Delta-Based and Edit-Based Lenses". In: *Proceedings of the 5th International Workshop on Bidirectional Transformations, Bx 2016, co-located with The European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 8, 2016.* 2016, pp. 1–13. URL: http://ceur-ws.org/Vol-1571/paper_13.pdf.

[120] Frédéric Jouault, Olivier Beaudoux, and ESEO Groupe. "Efficient OCL-based Incremental Transformations". In: *16th International Workshop in OCL and Textual Modeling*. 2016.

[121] Frédéric Jouault and Ivan Kurtev. "Transforming models with ATL". In: *Satellite Events at the MoDELS 2005 Conference*. Springer. 2006, pp. 128–138.

[122] Frédéric Jouault and Massimo Tisi. "Towards Incremental Execution of ATL Transformations". In: *Theory and Practice of Model Transformations: ICMT 2010*. Springer International Publishing, 2010, pp. 123–137.

[123] Gerd Kainz, Christian Buckl, Stephan Sommer, and Alois Knoll. "Model-to-Metamodel Transformation for the Development of Component-Based Systems". In: *Model Driven Engineering Languages and Systems: 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part II*. Ed. by Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 391–405. ISBN: 978-3-642-16129-2. DOI: 10.1007/978-3-642-16129-2_28.

[124] Kosaku Kimura, Yoshihide Nomura, Yuka Tanaka, Hidetoshi Kurihara, and Rieko Yamamoto. "Practical Multi-level Modeling on MOF-compliant Modeling Frameworks." In: *Proceedings of the 2nd International Workshop on Multi-Level Modelling co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2015), Ottawa, Canada, 2015.* 2015, pp. 43–52.

[125] Valerie King and Garry Sagert. "A fully dynamic algorithm for maintaining the transitive closure". In: *Proceedings of the thirty-first annual ACM symposium on Theory of computing.* ACM. 1999, pp. 492–498.

[126] Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. "BiGUL: a formally verified core language for putback-based bidirectional programming". In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016.* 2016, pp. 61–72. DOI: 10.1145/2847538.2847544. URL: http://doi.acm.org/10.1145/2847538.2847544.

[127] Max E. Kramer, Georg Hinkel, Heiko Klare, Michael Langhammer, and Erik Burger. "A Controlled Experiment Template for Evaluating the Understandability of Model Transformation Languages". In: *Proceedings of the Second International Workshop on Human Factors in Modeling co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016).* Vol. 1805. CEUR Workshop Proceedings. Saint Malo, France: CEUR-WS.org, Oct. 2016, pp. 11–18. URL: https://publikationen.bibliothek.kit.edu/1000069163.

[128] Max Emanuel Kramer. "Specification Languages for Preserving Consistency between Models of Different Languages". PhD thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology (KIT), 2017. 278 pp. DOI: 10.5445/IR/1000069284. URL: http://nbn-resolving.org/urn:nbn:de:swb:90-692845.

[129] F. Krikava, P. Collet, and R. France. "SIGMA: Scala Internal Domain-Specific Languages for Model Manipulations". In: *Proceedings of 17th International Conference on Model-Driven Engineering Languages and Systems.* Vol. 8767. 2014. DOI: 10.1007/978-3-319-11653-2. URL: http://link.springer.com/10.1007/978-3-319-11653-2.

[130] Filip Krikava. "Solving the TTC'15 Train Benchmark Case Study with SIGMA". In: *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015)*

federation of conferences, L'Aquila, Italy, July 24, 2015. 2015, pp. 167–175. URL: http://ceur-ws.org/Vol-1524/paper22.pdf.

[131] Géza Kulcsár, Sven Peldszus, and Malte Lochau. "Object-oriented Refactoring of Java Programs using Graph Transformation". In: *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L'Aquila, Italy, July 24, 2015*. 2015, pp. 53–82. URL: http://ceur-ws.org/Vol-1524/paper3.pdf.

[132] Harumi A Kuno and Elke A Rundensteiner. "Using object-oriented principles to optimize update propagation to materialized views". In: *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*. IEEE. 1996, pp. 310–317.

[133] Angelika Kusel, Juergen Etzlstorfer, Elisabeth Kapsammer, Philip Langer, Werner Retschitzegger, Johannes Schoenboeck, Wieland Schwinger, and Manuel Wimmer. "A Survey on Incremental Model Transformation Approaches". In: *ME 2013–Models and Evolution Workshop Proceedings*. Citeseer. 2013, p. 4.

[134] Kevin Lano and Shekoufeh Kolahdouz Rahimi. "Families to Persons Case with UML-RSDS". In: *Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) federation of conferences*. Ed. by Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava. CEUR Workshop Proceedings. Marburg, Germany: CEUR-WS.org, July 2017.

[135] Juan De Lara, Esther Guerra, and Jesús Sánchez Cuadrado. "When and How to Use Multilevel Modelling". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24.2 (2014), p. 12.

[136] Marius Lauder, Anthony Anjorin, Gergely Varró, and Andy Schürr. "Efficient Model Synchronization with Precedence Triple Graph Grammars". In: *Graph Transformations*. Ed. by Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg. Vol. 7562. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 401–415. ISBN: 978-3-642-33653-9. DOI: 10.1007/978-3-642-33654-6_27. URL: http://dx.doi.org/10.1007/978-3-642-33654-6_27.

[137] F William Lawvere and Robert Rosebrugh. *Sets for mathematics*. Cambridge University Press, 2003.

[138] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. "Developing eMoflon with eMoflon". In: *Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings*. 2014, pp. 138–145. DOI: 10.1007/978-3-319-08789-4_10. URL: http://dx.doi.org/10.1007/978-3-319-08789-4_10.

[139] Erhan Leblebici, Anthony Anjorin, Andy Schürr, Stephan Hildebrandt, Jan Rieke, and Joel Greenyer. "A Comparison of Incremental Triple Graph Grammar Tools". In: *Electronic Communications of the EASST* 67 (2014).

[140] Barbara Liskov. "Keynote Address - Data Abstraction and Hierarchy". In: *SIGPLAN Not.* 23.5 (Jan. 1987), pp. 17–34. ISSN: 0362-1340. DOI: 10.1145/62139.62141. URL: http://doi.acm.org/10.1145/62139.62141.

[141] Fernando Macías, Adrian Rutle, and Volker Stolz. "MultEcore: Combining the Best of Fixed-Level and Multilevel Metamodelling." In: *Proceedings of the 3rd International Workshop on Multi-Level Modelling co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2016), Saint-Malo, France, October 4, 2016*. 2016, pp. 66–75.

[142] Fernando Macías, Adrian Rutle, and Volker Stolz. "Multilevel Modeling with MultEcore – A Contribution to the MULTI 2017 Challenge". In: *Proceedings of the 4th International Workshop on Multi-Level Modelling co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2017), Austin, TX, USA, 2017*. 2017.

[143] Azzam Maraee and Mira Balaban. "Inter-association Constraints in UML2: Comparative Analysis, Usage Recommendations, and Modeling Guidelines". In: *Model Driven Engineering Languages and Systems: 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30–October 5, 2012. Proceedings*. Ed. by Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 302–318. ISBN: 978-3-642-33666-9. DOI: 10.1007/978-3-642-33666-9_20. URL: http://dx.doi.org/10.1007/978-3-642-33666-9_20.

[144] Azzam Maraee and Mira Balaban. "On the Interaction of Inter-relationship Constraints". In: *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*. MoDeVVa. Wellington, New Zealand: ACM, 2011, 3:1–3:8. ISBN: 978-1-4503-0914-1. DOI: 10.1145/2095654.2095658. URL: http://doi.acm.org/10.1145/2095654.2095658.

[145] Object Management Group (OMG). *Model Driven Architecture - Specifications*. 2006. URL: http://www.omg.org/mda/specs.htm.

[146] Erik Meijer. "Reactive Extensions (Rx): Curing Your Asynchronous Programming Blues". In: *ACM SIGPLAN Commercial Users of Functional Programming*. CUFP '10. Baltimore, Maryland: ACM, 2010, 11:1–11:1. ISBN: 978-1-4503-0516-7. DOI: 10.1145/1900160.1900173. URL: http://doi.acm.org/10.1145/1900160.1900173.

[147] Erik Meijer, Brian Beckman, and Gavin Bierman. "LINQ: Reconciling Object, Relations and XML in the .NET Framework". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, IL, USA: ACM, 2006, pp. 706–706. ISBN: 1-59593-434-0. DOI: 10.1145/1142473.1142552. URL: http://doi.acm.org/10.1145/1142473.1142552.

[148] Leo A Meyerovich, Arjun Guha, Jacob Baskin, Gregory H Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. "Flapjax: a programming language for Ajax applications". In: *ACM SIGPLAN Notices*. Vol. 44. 10. ACM. 2009, pp. 1–20.

[149] Leo A Meyerovich and Ariel S Rabkin. "Empirical analysis of programming language adoption". In: *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. ACM. 2013, pp. 1–18.

[150] Victoria Mittelbach. "Model-driven Consistency Preservation in Cyber-Physical Systems". MA thesis. Karlsruhe Institute of Technology (KIT), Germany.

[151] Object Management Group (OMG). *MOF 2.5 Core Specification (formal/2015-06-05)*. June 2015. URL: http://www.omg.org/spec/MOF/2.5.

[152] Parastoo Mohagheghi, Miguel A Fernandez, Juan A Martell, Mathias Fritzsche, and Wasif Gilani. "MDE adoption in industry: challenges and success criteria". In: *Models in Software Engineering*. Springer, 2009, pp. 54–59.

[153] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, and Miguel A Fernandez. "An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases". In: *Empirical Software Engineering* 18.1 (2013), pp. 89–116.

[154] Ulrich Nickel, Jörg Niere, and Albert Zündorf. "The FUJABA Environment". In: *Proceedings of the 22Nd International Conference on Software Engineering*. ICSE '00. Limerick, Ireland: ACM, 2000, pp. 742–745. ISBN: 1-58113-206-9. DOI: 10.1145/337180.337620. URL: http://doi.acm.org/10.1145/337180.337620.

[155] Pilar Nieto, Dolors Costal, and Cristina Gómez. "Enhancing the semantics of UML association redefinition". In: *Data & Knowledge Engineering* 70.2 (2011), pp. 182–207. ISSN: 0169-023X. DOI: http://doi.org/10.1016/j.datak.2010.10.007. URL: http://www.sciencedirect.com/science/article/pii/S0169023X10001278.

[156] Object Management Group (OMG). *Unified Modeling Language (UML) – Version 2.5 (formal/2015-03-01)*. Mar. 2015. URL: http://www.omg.org/spec/UML/2.5/PDF.

[157] Object Management Group (OMG). *Object Constraint Language, v2.0 (formal/06-05-01)*. 2006. URL: http://www.omg.org/cgi-bin/doc?formal/2006-05-01.

[158] James J Odell. "Power types". In: *Journal of Object-Oriented Programming* 7.2 (1994), p. 8.

[159] Babajide J Ogunyomi. "Incremental Model-to-Text Transformation". PhD thesis. University of York, 2016.

[160] Gérard Paligot, Nicolas Petitprez, and Martin Monperrus. "Refactoring Java Programs using Spoon". In: *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L'Aquila, Italy, July 24, 2015*. 2015, pp. 88–94. URL: http://ceur-ws.org/Vol-1524/paper11.pdf.

[161] Nicolas Pätzold. "Mapping ATL to NMF Synchronizations". Master Thesis. Karlsruhe Institute of Technology, 2017.

[162] Sven Peldszus, Jens Bürger, and Daniel Strüber. "Detecting and Preventing Power Outages in a Smart Grid using eMoflon". In: *Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) federation of conferences*. Ed. by Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava. CEUR Workshop Proceedings. Marburg, Germany: CEUR-WS.org, July 2017.

[163] Sven Peldszus, Géza Kulcsár, and Malte Lochau. "A Solution to the Java Refactoring Case Study using eMoflon". In: *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L'Aquila, Italy, July 24, 2015.* 2015, pp. 118–122. URL: http://ceur-ws.org/Vol-1524/paper20.pdf.

[164] Alfred Pohl. "A review of wireless SAW sensors". In: *IEEE transactions on ultrasonics, ferroelectrics, and frequency control* 47.2 (2000), pp. 317–332.

[165] Claudia Pons. "Heuristics on the definition of UML refinement patterns". In: *SOFSEM 2006: Theory and Practice of Computer Science*. Springer, 2006, pp. 461–470.

[166] William Pugh and Tim Teitelbaum. "Incremental computation via function caching". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1989, pp. 315–328.

[167] Liu Qianqian, Xiangjun Zeng, Ma Xue, and Li Xiang. "A new smart distribution grid fault self-healing system based on traveling-wave". In: *Industry Applications Society Annual Meeting, 2013 IEEE*. Oct. 2013, pp. 1–6. DOI: 10.1109/IAS.2013.6682600.

[168] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (ptc/07-07-07)*. 2007. URL: http://www.omg.org/docs/ptc/07-07-07.pdf.

[169] G. Ramalingam and Thomas Reps. "On the computational complexity of dynamic graph problems". In: *Theoretical Computer Science* 158.1 (1996), pp. 233–277. ISSN: 0304-3975. DOI: http://dx.doi.org/10.1016/0304-3975(95)00079-8.

[170] Ganesan Ramalingam and Thomas Reps. “A categorized bibliography on incremental computation”. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1993, pp. 502–510.

[171] István Ráth, Gábor Bergmann, András Ökrös, and Dániel Varró. “Live model transformations driven by incremental pattern matching”. In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2008, pp. 107–121.

[172] István Ráth, Ábel Hegedüs, and Dániel Varró. “Derived features for EMF by integrating advanced model queries”. In: *Modelling Foundations and Applications*. Springer, 2012, pp. 102–117.

[173] Alexander Reder and Alexander Egyed. “Incremental consistency checking for complex design rules and larger model changes”. In: *Model Driven Engineering Languages and Systems*. Springer, 2012, pp. 202–218.

[174] Steven P. Reiss. “An Approach to Incremental Compilation”. In: *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*. SIGPLAN ’84. Montreal, Canada: ACM, 1984, pp. 144–156. ISBN: 0-89791-139-3. DOI: 10.1145/502874.502889. URL: http://doi.acm.org/10.1145/502874.502889.

[175] Thomas Reps. “Optimal-time Incremental Semantic Analysis for Syntax-directed Editors”. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’82. Albuquerque, New Mexico: ACM, 1982, pp. 169–176. ISBN: 0-89791-065-6. DOI: 10.1145/582153.582172. URL: http://doi.acm.org/10.1145/582153.582172.

[176] Elie Richa, Etienne Border, and Laurent Pautet. “Translating ATL Model Transformations to Algebraic Graph Transformations”. In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2015, pp. 183–198.

[177] Alessandro Rossini, Juan Lara, Esther Guerra, and Nikolay Nikolov. “A Comparison of Two-Level and Multi-level Modelling for Cloud-Based Applications”. In: *Modelling Foundations and Applications: 11th European Conference, ECMFA 2015*. Ed. by Gabriele Taentzer and Francis Bordeleau. Springer, 2015, pp. 18–32. ISBN: 978-3-319-21151-0. DOI: 10.1007/978-3-319-21151-0_2.

[178] Arturo Salz and Mark Horowitz. "IRSIM: An incremental MOS switch-level simulator". In: *Design Automation, 1989. 26th Conference on*. IEEE. 1989, pp. 173–178.

[179] Leila Samimi-Dehkordi, Bahman Zamani, and Shekoufeh Kolahdouz Rahimi. "Solving the Families to Persons Case using EVL+Strace". In: *Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) federation of conferences*. Ed. by Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava. CEUR Workshop Proceedings. Marburg, Germany: CEUR-WS.org, July 2017.

[180] Mohamed Sarwat, Sameh Elnikety, Yuxiong He, and Gabriel Kliot. "Horton: Online query execution engine for large distributed graphs". In: *Proceedings - International Conference on Data Engineering* (2012), pp. 1289–1292. ISSN: 10844627. DOI: 10.1109/ICDE.2012.129.

[181] Matthias Schöttle and Jörg Kienzle. "Concern-Oriented Interfaces for Model-Based Reuse of APIs". In: *Proceedings of the 18th International Conference on Model-Driven Engineering Languages and Systems - MODELS 2015*. ACM, Sept. 2015, pp. 286–291.

[182] Andy Schürr. "Specification of graph translators with triple graph grammars". In: *Graph-Theoretic Concepts in Computer Science*. Springer. 1995, pp. 151–163.

[183] Stephan Seifermann, Emre Taspolatoglu, Robert Heinrich, and Ralf Reussner. "Challenges in Secure Software Evolution - The Role of Software Architecture". In: *3rd Collaborative Workshop on Evolution and Maintenance of Long-Living Software Systems*. Softwaretechnik-Trends Band 36 Heft 1. 2016, pp. 8–11.

[184] Shane Sendall and Wojtek Kozaczynski. *Model transformation the heart and soul of model-driven software development*. Tech. rep. 2003.

[185] Herbert Stachowiak. *Allgemeine Modelltheorie*. Wien: Springer Verlag, 1973. ISBN: 3-211-81106-0.

[186] Thomas Stahl and Markus Völter. *Model-driven software development : technology, engineering, management*. Ed. by Jorn Bettin, Krzysztof Czarnecki, and Bettina von Stockfleth. Chichester: John Wiley & Sons, 2006. ISBN: 9780470025703.

[187] Miroslaw Staron. “Adopting Model Driven Software Development in Industry - A Case Study at Two Companies”. In: *MoDELS*. 2006, pp. 57–72.

[188] Miroslaw Staron. “Adopting model driven software development in industry–a case study at two companies”. In: *Model Driven Engineering Languages and Systems*. Springer, 2006, pp. 57–72.

[189] Dániel Stein, Gábor Szárnyas, and István Ráth. “Java Refactoring Case: A VIATRA Solution”. In: *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L’Aquila, Italy, July 24, 2015*. 2015, pp. 100–117. URL: http://ceur-ws.org/Vol-1524/paper21.pdf.

[190] Perdita Stevens. “A Landscape of Bidirectional Model Transformations”. English. In: *Generative and Transformational Techniques in Software Engineering II*. Ed. by Ralf Lämmel, Joost Visser, and João Saraiva. Vol. 5235. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 408–424. ISBN: 978-3-540-88642-6. DOI: 10.1007/978-3-540-88643-3_10. URL: http://dx.doi.org/10.1007/978-3-540-88643-3_10.

[191] Perdita Stevens. “Bidirectional model transformations in QVT: semantic issues and open questions”. In: *Software & Systems Modeling* 9.1 (2010), pp. 7–20. ISSN: 1619-1366. DOI: 10.1007/s10270-008-0109-9. URL: http://dx.doi.org/10.1007/s10270-008-0109-9.

[192] Christian Stier, Anne Koziolek, Henning Groenda, and Ralf Reussner. “Model-Based Energy Efficiency Analysis of Software Architectures”. In: *Proceedings of the 9th European Conference on Software Architecture (ECSA ’15)*. Lecture Notes in Computer Science. Dubrovnik, Croatia: Springer, 2015.

[193] Jeff Sutherland. “Business objects in corporate information systems”. In: *ACM Computing Surveys (CSUR)* 27.2 (1995), pp. 274–276.

[194] Gábor Szárnyas, Márton Búr, and István Ráth. “Train Benchmark Case: an EMF-INCQUERY Solution”. In: *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L’Aquila, Italy, July 24, 2015*. 2015, pp. 157–166. URL: http://ceur-ws.org/Vol-1524/paper19.pdf.

[195] Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. "IncQuery-D: A Distributed Incremental Model Query Framework in the Cloud". In: *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*. 2014, pp. 653–669. DOI: 10.1007/978-3-319-11653-2_40. URL: http://dx.doi.org/10.1007/978-3-319-11653-2_40.

[196] Gábor Szárnyas, Benedek Izsó, István Ráth, and Dániel Varró. "The Train Benchmark: cross-technology performance evaluation of continuous model queries". In: *Software & Systems Modeling* (Jan. 2017). ISSN: 1619-1374. DOI: 10.1007/s10270-016-0571-8. URL: https://doi.org/10.1007/s10270-016-0571-8.

[197] Gábor Szárnyas, Oszkár Semeráth, István Ráth, and Dániel Varró. "The TTC 2015 Train Benchmark Case for Incremental Model Validation". In: *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L'Aquila, Italy, July 24, 2015*. 2015, pp. 129–141. URL: http://ceur-ws.org/Vol-1524/paper2.pdf.

[198] Michael Szvetits and Uwe Zdun. "Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime". English. In: *Software & Systems Modeling* (2013), pp. 1–39. ISSN: 1619-1366. DOI: 10.1007/s10270-013-0394-9. URL: http://dx.doi.org/10.1007/s10270-013-0394-9.

[199] Robert Tarjan. "Depth-first search and linear graph algorithms". In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.

[200] Robert Endre Tarjan. "A class of algorithms which require nonlinear time to maintain disjoint sets". In: *Journal of Computer and System Sciences* 18.2 (1979), pp. 110–127. ISSN: 0022-0000.

[201] Robert Endre Tarjan. "Efficiency of a good but not linear set union algorithm". In: *Journal of the ACM (JACM)* 22.2 (1975), pp. 215–225.

[202] Massimo Tisi, Salvador Martinez, and Hassene Choura. "Parallel Execution of ATL Transformation Rules". In: *Model-Driven Engineering Languages and Systems*. Springer, 2013, pp. 656–672.

[203] Javier Troya and Antonio Vallecillo. "A Rewriting Logic Semantics for ATL". In: *Journal of Object Technology* 10.5 (2011), pp. 1–29. DOI: \url{10.5381/jot.2011.10.1.a5}.

[204] Axel Uhl, Thomas Goldschmidt, and Manuel Holzleitner. "Using an OCL impact analysis algorithm for view-based textual modelling". In: *Electronic Communications of the EASST* 44 (2011).

[205] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedús, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. "EMF-IncQuery: An integrated development environment for live model queries". In: *Science of Computer Programming* 98, Part 1 (2015), pp. 80–99. ISSN: 0167-6423. DOI: http://dx.doi.org/10.1016/j.scico.2014.01.004. URL: http://www.sciencedirect.com/science/article/pii/S0167642314000082.

[206] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. "Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework". In: *Software & Systems Modeling* 15.3 (2016), pp. 609–629. ISSN: 1619-1374. DOI: 10.1007/s10270-016-0530-4. URL: http://dx.doi.org/10.1007/s10270-016-0530-4.

[207] Dániel Varró and András Pataricza. "Generic and meta-transformations for model transformation engineering". In: *« UML» 2004-The Unified Modeling Language. Modelling Languages and Applications*. Springer, 2004, pp. 290–304.

[208] Dániel Varró and András Pataricza. "VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML (The Mathematics of Metamodeling is Metamodeling Mathematics)". In: *Software and Systems Modeling* 2.3 (2003), pp. 187–210.

[209] Vladimir Viyovic, Milan Maksimovic, and Branko Perisic. "Sirius: A rapid development of DSM graphical editor". In: *Intelligent Engineering Systems (INES), 2014 18th International Conference on*. IEEE. 2014, pp. 233–238.

[210] Markus Voelter. "Language and IDE Modularization and Composition with MPS". English. In: *Generative and Transformational Techniques in Software Engineering IV*. Ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Vol. 7680. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 383–430. ISBN: 978-3-642-35991-0. DOI: 10.1007/978-3-642-35992-7_11. URL: http://dx.doi.org/10.1007/978-3-642-35992-7_11.

[211] Dennis Wagelaar. "The ATL/EMFTVM Solution to the Train Benchmark Case". In: *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L'Aquila, Italy, July 24, 2015.* 2015, pp. 152–156. URL: http://ceur-ws.org/Vol-1524/paper16.pdf.

[212] Dennis Wagelaar, Massimo Tisi, Jordi Cabot, and Frédéric Jouault. "Towards a General Composition Semantics for Rule-Based Model Transformation". In: *Model Driven Engineering Languages and Systems: 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings.* Ed. by Jon Whittle, Tony Clark, and Thomas Kühne. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 623–637. ISBN: 978-3-642-24485-8. DOI: 10.1007/978-3-642-24485-8_46. URL: http://dx.doi.org/10.1007/978-3-642-24485-8_46.

[213] Dennis Wagelaar, Ragnhild Van Der Straeten, and Dirk Deridder. "Module Superimposition: A Composition Technique for Rule-based Model Transformation Languages". English. In: *Software and Systems Modeling (SoSyM)* 9 (3 2010), pp. 285–309. ISSN: 1619-1366. DOI: 10.1007/s10270-009-0134-3.

[214] Benjamin Wanner. "Incrementality in the Cloud: A Distributed Incremental Evaluation System Based on Virtual Actors". Master Thesis. Karlsruhe Institute of Technology, 2016.

[215] Alexander Wert, Jens Happe, and Lucia Happe. "Supporting swift reaction: automatically uncovering performance problems by systematic experiments". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 552–561. ISBN: 978-1-4673-3076-3. URL: http://dl.acm.org/citation.cfm?id=2486788.2486861.

[216] Arif Wider. "Implementing a Bidirectional Model Transformation Language as an Internal DSL in Scala". In: *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference.* Vol. 1133. CEUR Workshop Proceedings. CEUR, 2014, pp. 63–70. URL: http://ceur-ws.org/Vol-1133/paper-10.pdf.

[217] Darren Willis, David J Pearce, and James Noble. "Caching and incrementalisation in the Java Query Language". In: *ACM Sigplan Notices* 43.10 (2008), pp. 1–18.

[218] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. "Fact or fiction–reuse in rule-based model-to-model transformation languages". In: *Theory and Practice of Model Transformations*. Springer, 2012, pp. 280–295.

[219] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. "Towards Automatic Model Synchronization from Model Transformations". In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ASE '07. Atlanta, Georgia, USA: ACM, 2007, pp. 164–173. ISBN: 978-1-59593-882-4.

[220] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 XMI Mapping Specification, v2.1 (formal/05-09-01)*. 2006. URL: `http://www.omg.org/cgi-bin/apps/doc?formal/05-09-01.pdf`.

[221] Albert Zündorf, Tobias George, Stefan Lindel, and Ulrich Norbisrath. "Story Driven Modeling Libary (SDMLib): an Inline DSL for modeling and model transformations, the Petrinet-Statechart case". In: *EPTCS* (2013).

[222] Albert Zündorf and Alexander Weidt. "The SDMLib Solution to the TTC 2017 Families 2 Persons Case". In: *Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) federation of conferences*. Ed. by Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava. CEUR Workshop Proceedings. Marburg, Germany: CEUR-WS.org, July 2017.

Appendix

Here, we list the questionnaires used in the case studies from the TTC to obtain the scores of submitted solutions. All questionnaires have been reformatted to fit the format of this thesis, but the questions have not been changed[123]. The questionnaires depiected here also contain the original description texts.

For the TTC 2015, the open peer review questionnaires are depicted, meanwhile for the TTC 2017, we depicted the Live Contest Questionnaire. As a reason, the live questionnaire at the TTC 2015 only consisted of a single question how the participants liked the tool, but the results of that questionnaire are not publicly available. For the TTC 2017, the questionnaire used for the open peer reviews is a superset of the questionnaire for the live reviews and contains additional questions that are not evaluated in the scope of this thesis.

Transformation Tool Contest 2015 Java Refactoring Open Peer Review Questionnaire

Reviewed solution

. . . . . .

[123] The open peer review questionnaires are not anonymous and contain a question for the author of the feedback. This question is omitted in the remainder as it is only relevant for organizational purposes.

Solution Details

Only the first two questions are mandatory. The detailed questions thereafter serve as further feedback in case the answer was accessible to the reviewer.

Does the solution fit to the challenge requirements? *

- ◯ Yes. It can be validated using ARTE.
- ◯ Yes, but it has to be validated manually.
- ◯ No, the solution structure is not compatible with the case.

Which version of the case has been solved? *

- ◯ Basic challenge
- ◯ Extended challenge
- ◯ None of these

Does the solution generate a Program Graph according to the case description?

Yes ◯ ◯ No

Are the refactoring changes synchronized in an incremental fashion?

Yes ◯ ◯ No

Does the solution generate refactored Java code?

Yes ◯ ◯ No

Solution Quality Aspects

Only the first question (overall reviewer score from 1 to 15) is mandatory. The detailed points thereafter are not necessary for submitting a review and serve merely as additional feedback for the developer.

Reviewer Score

. . . . . .

Comprehensibility

The question if a solution works with an understandable mechanism is of high importance, especially in the scope of the Transformation Tool Contest where such a comprehensible solution facilitates discussion and contributes to a profitable event.

lowest ○ ○ ○ ○ ○ highest

Readability

In contrast to comprehensibility, this aspect refers to the outer appearance of the tool - whether it has a nice and/or user-friendly interface, can be easily operated, maybe even with custom-tailored commands or a DSL, ...

lowest ○ ○ ○ ○ ○ highest

Communication with the user

Although related to readability, this aspect refers to the quality, informativeness and level of detailedness of the actual messages given to the user. In other words: Am I as user informed that everything went smoothly? In case of some failure or malfunction, am I thoroughly informed what actually went wrong?

lowest ○ ○ ○ ○ ○ highest

Conciseness (if applicable) How many LOC does the solution have?

. . . . . .

Extensions

Extension score (if applicable, 1-15)

. . . . . .

Extension Score motivation If there have been some extension points awarded, please give a short textual motivation of your score.

. . . . . .

. . . . . .

Further comments Here, you can give us feedback about your answers and/or the evaluation procedure.

. . . . . .

. . . . . .

Transformation Tool Contest 2015 Train Benchmark Open Peer Review Questionnaire

Reviewed solution

. . . . . .

Completeness & Correctness

The goal of the correctness check is to determine if the different model query and transformation tasks are correctly and fully implemented in the submitted solutions. Each task and extension task is scored independently 0–3 points by the following rules:

- **0 points:** The task is not solved.
- **1–2 points:** The task is partially solved, the solution provides the subset or the superset of the expected results.
- **3 points:** The task is completely and correctly solved.

Minus 1 point if only the query is implemented, but the transformation is not.

Task 1: PosLength

. . . . . .

Task 2: SwitchSensor

. . . . . .

Task 3: SwitchSet

. . . . . .

Extension Task 1: RouteSensor

. . . . . .

Extension Task 2: SemaphoreNeighbor

. . . . . .

Conciseness

The validation rules are frequently changed and extended, therefore it is important to be able to define queries and transformations in a concise manner. These properties are scored based on the following rules:

- **0 points:** The task is not solved.
- **1 point:** The task is solved, but the solution is not significantly more concise than it would be in a general-purpose imperative language (e.g. Java), or the task is partially solved and the result set needs additional processing.
- **2 points:** The task is solved, the query and the transformation is defined in a declarative, visual or other query language, but the specification is hard to formulate.
- **3 points:** The solution is compact, the query and the transformation are defined in a concise manner.

Minus 1 point if only either the query or the transformation is implemented.

Task 1: PosLength

. . . . . .

Task 2: SwitchSensor

. . . . . .

Task 3: SwitchSet

. . . . . .

Extension Task 1: RouteSensor

. . . . . .

Extension Task 2: SemaphoreNeighbor

. . . . . .

Readability

The readability and descriptiveness of each query and transformation is scored with respect to a model validation use case. The score represents how well model queries are used as model constraints, and how well repair operations can be expressed by model transformations. The score is given based on the following rules:

- **0 points:** The task is not solved.
- **1 point:** The task is solved, but the solution is not significantly more readable than it would be in a general-purpose imperative language (e.g. Java), or the task is just partially solved. For example, a typical EMF validator should get 1 point.
- **2 points:** The task is solved, the query and the transformation follows the description of the constraint and repair rule, but it is difficult to comprehend the meaning of the solution. For example, a foreign key constraint checked by a query formulated in SQL should get 2 points.
- **3 points:** The solution could be presented in the documentation of the modeling domain, and it is easier to comprehend than a textual description in natural language. For example, a solution similar to the graphical notation used in this paper should get 3 points.

Minus 1 point if the language is only able to express either the constraint (e.g. OCL) or the repair operation.

Task 1: PosLength

. . . . . .

Task 2: SwitchSensor

. . . . . .

Task 3: SwitchSet

. . . . . .

Extension Task 1: RouteSensor

. . . . . .

Extension Task 2: SemaphoreNeighbor

. . . . . .

Usability

SHARE image

Did the SHARE solution work as expected?

- ◯ Yes
- ◯ No
- ◯ Other

Running the solution

How easy did you find to run the solution? Did the documentation cover the required steps?

. . . . . .

. . . . . .

Setting up the solution

If you set up the solution on a local system, please share your experience.

. . . . . .

. . . . . .

Additional Feedback

You may provide some additional feedback on the reviewed solution that did not fit into the questions above.

Strong points

Highlight some strong points of the solution.

. . . . . .

. . . . . .

Weaknesses

Highlight some weaknesses of the solution (if applicable).

. . . . . .

. . . . . .

Further comments

You may provide further comments for the authors to improve their tool or solution.

. . . . . .

. . . . . .

TTC 2017 Smart Grid Live Evaluation Questionnaire

The goal of this questionnaire is to evaluate quality attributes of the presented solutions that are not well captured by metrics. The results of this questionnaire is intended to be used for further research.

Which solution are you reviewing?

. . . . . .

How do you rate the overall quality of the solution?

Very poor ○ ○ ○ ○ ○ Very good

How do you rate the understandability of the Outage Detection task?

The solution is not understandable at all ○ ○ ○ ○ ○ The solution is very understandable

How do you rate the understandability of the Outage Prevention task?

The solution is not understandable ○ ○ ○ ○ ○ The solution is very understandable

How confident are you about your evaluation?

Not confident ○ ○ ○ ○ ○ Expert

How did you like the solution presentation?

very poor ○ ○ ○ ○ ○ very good

What did you like about the solution?

. . . . . .

What did you dislike about the solution?

. . . . . .

Do you have other comments about the solution?

. . . . . .

Thanks for your feedback!

TTC 2017 Families to Persons Live Evaluation Questionnaire

The goal of this questionnaire is to evaluate quality attributes of the presented solutions that are not well captured by metrics. The results of this questionnaire is intended to be used for further research.

Which solution are you reviewing?

. . . . . .

How do you rate the overall quality of the solution?

very poor ○ ○ ○ ○ ○ very good

How do you rate the understandability of the solution?

not suitable ○ ○ ○ ○ ○ very suitable

How confident are you about your evaluation?

Not confident ○ ○ ○ ○ ○ Expert

How did you like the solution presentation?

very poor ○ ○ ○ ○ ○ very good

What did you like about the solution?

. . . . . .

. . . . . .

What did you dislike about the solution?

. . . . . .

. . . . . .

Do you have other comments about the solution?

. . . . . .

. . . . . .

Thanks for your feedback!

Acronyms

ADL Architecture Description Language.

API Application Programming Interface.

AST Abstract Syntax Tree.

ATL Atlas Transformation Language.

CMOF Complete MOF.

DDG Dynamic Dependency Graph.

DSL Domain Specific Language.

EMF Eclipse Modeling Framework.

EMOF Essential Meta Object Facility.

HOT Higher-order Transformation.

IDE Integrated Development Environment.

LHS Left Hand Side.

LINQ Language Integrated Query.

LSP Liskov' Substitution Principle.

M2M Model-to-Model.

M2T Model-to-Text.

MDE Model-driven engineering.

MOF Meta Object Facility.

MTC Mutable Type Category.

MTL Model transformation language.

NMF .NET Modeling Framework.

NTL NMF Transformations Language.

OCL Object Constraint Language.

OMG Object Management Group.

PCM Palladio Component Model.

PG Program Graph.

QVT Query View Transformation.

RHS Right Hand Side.

SQO Standard Query Operator.

TGG Triple Graph Grammar.

TTC Transformation Tool Contest.

UML Universal Modeling Language.

URI Unique Resource Identifier.

XMI XML Metadata Interchange.

Glossary

NMF Expressions NMF Expressions is the implementation of an incrementalization system used in NMF (cf. Chapter 4).. 79, 91, 118, 242, 245, 246, 250, 261, 262, 275, 290, 292, 293, 296, 328, 340, 341, 374, 391, 392, 394

NMF Synchronizations NMF Synchronizations is the implementation of synchronization blocks used in NMF (cf. Chapter III).. xxv, 164, 166, 167, 170–172, 233, 277–283, 285, 286, 290, 291, 293–296, 298, 299, 301–304, 306, 307, 311–313, 315, 316, 327–329, 339, 341–343, 349, 350, 375, 393, 394, 396, 405, 408

NMF Transformations NMF Transformations is the model transformation framework that is part of NMF [92].. 159, 164, 167

eMoflon eMoflon is an implementation of TGGs [8].. 164, 166–168, 170, 171, 278, 279, 281, 282, 308, 315, 317, 329, 331, 334, 335, 337, 343, 344, 346, 350, 374, 393

API The Application Programming Interface (API) of a software artifact is the set of publicly visible methods through which clients can use and extend that artifact.. 37, 87, 90, 92, 95, 98, 108, 154, 218, 224, 226–228, 230, 357, 360, 364, 389, 391, 400, 401

AST The Abstract Syntax Tree (AST) of a method is the representation of the methods source code in a tree structure.. 123

ATL The Atlas Transformation Language is a model transformation language initially developed by Frederic Jouault [121].. 163, 233, 236, 240, 266, 268, 273, 276, 282–291, 294–299, 301, 303–305, 307, 336, 370, 374, 375, 394, 398, 399

CMOF The Complete Meta-Object Facility (CMOF) refers to the MOF standard [151] in its complete form, as opposed to EMOF that refers to a reduced subset.. 16, 17, 28, 32, 176, 400

DDG A Dynamic Dependency Graph (DDG), also known as Incremental Dependency Graph, is a graph that tracks the dependencies of a given computation in order to decide when the computation must be invalidated.. 6, 14, 91–100, 108, 110, 118, 119, 122, 123, 125–127, 129, 201, 228, 229, 250, 252, 253, 261, 263, 266, 273, 274, 331, 343, 361, 366, 367, 378, 388, 389

DeepPCM DeepPCM is an adaption of PCM that follows deep modeling principles described in Chapter 8.. xix, xxv, 204, 214–217, 219–226, 229

DSL A Domain-specific Language (DSL, [66]) is a small language usually developed for a purpose.. 8, 9, 29, 93, 133, 152, 165, 171, 350, 396, 398

Ecore Ecore is the meta-metamodel used in EMF. It is an implementation of EMOF.. 8, 31, 32, 139, 176, 184, 186, 201, 270, 351

EMF The Eclipse Modeling Framework (EMF) is a widely used framework in the model-driven community to represent models in memory.. 28, 31, 187, 369, 391, 401, 404, 406

EMFTVM The EMF Transformation VM (EMFTVM) is a virtual machine specifically developed to provide a uniform and fast platform for model transformations using EMF to access models [212]. 301, 304, 307

EMOF The Essential Meta Object Facility (EMOF) standard is a subset of the MOF standard implemented by Ecore.. 5, 8, 16, 17, 28, 73, 176, 194, 203, 364, 401, 406

FlowM2M A model transformation language based on explicit data flow. The language was specifically developed for the TTC 2016 Live case by Antonio Garcia-Dominguez and others.. xix, 266–270

HOT A Higher-order Transformation (HOT) is a model transformation that either takes model transformations as input, produces model transformations as output or a combination of both [207].. 283, 398, 399

IDE An Integrated Development Environment (IDE) is a program that provides rich tools to develop software.. 96

Incerator Incerator is a command-line tool as part of NMF that allows to automatically optimize the granularity of the incrementalization for a given model analysis, input model, change sequence and the target machine (cf. Chapter 5).. 110, 127–129, 241, 242, 253, 255, 256, 258, 261, 266, 372, 383

JaMoPP The Java Printer and Parser (JaMoPP) is a framework to parse Java code in an EMF model and on the contrary print this model back to Java code [87]. 311–314, 316

LINQ Language Integrated Query (LINQ) is a framework on the .NET platform that allows users to specify queries directly in C# and execute them on arbitrary targets (as long as there is a provider for this target). This can include object sources, but also relational databases. As LINQ sticks to the SQO, several languages offer dedicated support.. 246, 349, 379, 380

LSP The Liskov' Substitution Principle (LSP) is a principle from behavioral subtyping introduced by Barbara Liskov [140]. It states that any property provable about a type must also be provable on each of its subtypes.. 183

MDE Model-driven Engineering (MDE) is an engineering approach that puts formal models described by metamodels at the center of any development.. 27, 177, 300

MOF The Meta Object Facility is a standardized modeling foundation.. 8, 16, 28, 59, 389, 406

MTC A Mutable Type Category (MTC) is a type system formalization based on category theory. It is introduced in Section 3.. 58, 59, 61–63, 65, 69, 72, 74, 80, 82, 83, 85, 111, 180

MTL A Model Transformation Language (MTL) is a DSL specialized on the specification of model transformations.. 29

NMeta NMeta is the meta-metamodel used in NMF.. xviii, 27, 31, 32, 139, 177, 184–187, 189, 192, 194–197, 213, 217, 219, 270, 311, 357, 359, 364

NMF The .NET Modeling Framework (NMF, [96]) is an open-source modeling framework implemented on the .NET platform to which the implementations of this dissertations contribute to.. xxv, xxvi, 17, 22, 31, 32, 92, 96, 99, 110, 127, 129, 154, 155, 187, 189, 224, 227, 239,

240, 242–245, 247–253, 255, 263, 265–267, 270–275, 277, 278, 287, 289, 291, 293, 296, 299, 308, 310, 311, 315–317, 326, 328, 329, 331, 334–336, 338–340, 342–344, 347–350, 354, 355, 359–362, 364, 369–372, 377–379, 404, 405

NTL The model transformation language used in NMF [92]. Unlike NMF Synchronizations, it is only used for unidirectional, non-incremental transformations. The largest NTL transformation that exist is the code generator transformation used to generate model code.. 31, 156, 157, 160, 161, 163, 277–281, 313

OCL The Object Constraint Language (OCL) is a standard by the OMG originally built to express constraints. However, it has also been used for model transformation, either through ATL [121] or directly.. 7, 28, 43, 176, 178, 184, 186, 195, 197, 201, 206, 223, 225, 284, 292, 293, 361, 391, 400

OMG The Object Management Group (OMG) is a standardization entity.. 28

PCM The Palladio Component Model (PCM, [21]) is an Architecture Description Language (ADL) used to predict several quality attributes of a software system at design time.. 21, 40–43, 203–206, 208, 209, 214, 215, 217, 218, 222–225, 227, 230

SQO The Standard Query Operators (SQOs) are a set of standardized API methods for monads to which several languages on the .NET platform offer dedicated syntax support. An example is the language C# where the syntax support is called query syntax. The most prominent SQO methods are the higher-order methods `select` and `where` that usually perform mapping and filter tasks.. 37, 38, 70, 79, 90, 99, 103, 104, 107, 241, 245, 291

TGG Triple Graph Grammars (TGGs) are a formal approach to describe bidirectional model transformations. Several transformation languages implement TGGs and are often graphical.. 7, 163, 164, 166–168, 170, 172, 282, 393, 395, 397, 398

TTC The Transformation Tool Contest (TTC)[124] is an academic contest with the goal to compare model transformation tools through solutions to individual cases.. 20, 22, 156, 158, 235, 236, 238–240, 242, 247, 253, 266,

[124] `http://www.transformation-tool-contest.eu`

269, 275, 308, 310, 315, 317, 326, 329, 331, 334–336, 338, 343, 347–349, 369, 371–374, 378, 379, 437

UML The Universal Modeling Language (UML) is a standard by the OMG to define the syntax, notation and semantics of modeling elements to describe software systems. For the notation, UML defines a set of 14 diagrams.. 32, 176, 186, 194, 196, 199, 212, 214, 234, 400

URI A Unique Resource Identifier (URI) is an identifier for any kind of resource independent of its physical location. The format of a URI is standardized by the W3C.. 31, 32, 218

VIATRA Query VIATRA Query [205] (formerly named EMF-IncQuery) is a framework for incremental graph pattern matching based on the incremental pattern matching algorithm by Bergmann [26].. 240–242, 247–253, 265, 370, 374, 375, 391, 398

XMI The XML Metadata Interchange (XMI) OMG standard defines how MOF models should be serialized to XML [220].. 28, 31, 32

The Karlsruhe Series on Software Design and Quality

ISSN 1867-0067

Band 1 Steffen Becker
Coupled Model Transformations for QoS Enabled Component-Based Software Design.
ISBN 978-3-86644-271-9

Band 2 Heiko Koziolek
Parameter Dependencies for Reusable Performance Specifications of Software Components.
ISBN 978-3-86644-272-6

Band 3 Jens Happe
Predicting Software Performance in Symmetric Multi-core and Multiprocessor Environments.
ISBN 978-3-86644-381-5

Band 4 Klaus Krogmann
Reconstruction of Software Component Architectures and Behaviour Models using Static and Dynamic Analysis.
ISBN 978-3-86644-804-9

Band 5 Michael Kuperberg
Quantifying and Predicting the Influence of Execution Platform on Software Component Performance.
ISBN 978-3-86644-741-7

Band 6 Thomas Goldschmidt
View-Based Textual Modelling.
ISBN 978-3-86644-642-7

Band 7 Anne Koziolek
Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes.
ISBN 978-3-86644-973-2

Band 8 Lucia Happe
Configurable Software Performance Completions through Higher-Order Model Transformations.
ISBN 978-3-86644-990-9

Band 9 Franz Brosch
Integrated Software Architecture-Based Reliability Prediction for IT Systems.
ISBN 978-3-86644-859-9

Band 10 Christoph Rathfelder
Modelling Event-Based Interactions in Component-Based Architectures for Quantitative System Evaluation.
ISBN 978-3-86644-969-5

Band 11 Henning Groenda
Certifying Software Component Performance Specifications.
ISBN 978-3-7315-0080-3

Band 12 Dennis Westermann
Deriving Goal-oriented Performance Models by Systematic Experimentation.
ISBN 978-3-7315-0165-7

Band 13 Michael Hauck
Automated Experiments for Deriving Performance-relevant Properties of Software Execution Environments.
ISBN 978-3-7315-0138-1

Band 14 Zoya Durdik
Architectural Design Decision Documentation through Reuse of Design Patterns.
ISBN 978-3-7315-0292-0

Band 15 Erik Burger
Flexible Views for View-based Model-driven Development.
ISBN 978-3-7315-0276-0

Band 16 **Benjamin Klatt**
Consolidation of Customized Product Copies into Software Product Lines.
ISBN 978-3-7315-0368-2

Band 17 **Andreas Rentschler**
Model Transformation Languages with Modular Information Hiding.
ISBN 978-3-7315-0346-0

Band 18 **Omar-Qais Noorshams**
Modeling and Prediction of I/O Performance in Virtualized Environments.
ISBN 978-3-7315-0359-0

Band 19 **Johannes Josef Stammel**
Architekturbasierte Bewertung und Planung von Änderungsanfragen.
ISBN 978-3-7315-0524-2

Band 20 **Alexander Wert**
Performance Problem Diagnostics by Systematic Experimentation.
ISBN 978-3-7315-0677-5

Band 21 **Christoph Heger**
An Approach for Guiding Developers to Performance and Scalability Solutions.
ISBN 978-3-7315-0698-0

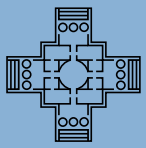
Band 22 **Fouad ben Nasr Omri**
Weighted Statistical Testing based on Active Learning and Formal Verification Techniques for Software Reliability Assessment.
ISBN 978-3-7315-0472-6

Band 23 **Michael Langhammer**
Automated Coevolution of Source Code and Software Architecture Models.
ISBN 978-3-7315-0783-3

Band 24 **Max Emanuel Kramer**
Specification Languages for Preserving Consistency between Models of Different Languages.
ISBN 978-3-7315-0784-0

Band 25 **Sebastian Michael Lehrig**
Efficiently Conducting Quality-of-Service Analyses by Templating Architectural Knowledge.
ISBN 978-3-7315-0756-7

Band 26 **Georg Hinkel**
Implicit Incremental Model Analyses and Transformations.
ISBN 978-3-7315-0763-5



The Karlsruhe Series on Software Design and Quality

Edited by Prof. Dr. Ralf Reussner

When models of a system change, any analyses based on these models have to be invalidated and thus have to be reevaluated again in order for the results to stay meaningful. In many cases, the time to get updated analysis results is critical. However, as most often only small parts of the model change, large parts of this reevaluation could be saved by using previous results but such an incremental execution is barely done in practice, as it is non-trivial and error-prone.

This work proposes an approach that allows implicit, formally justified, automatically tuned incrementalization of model analyses and a new formalism how this incrementalization system can be used to empower incremental, uni- or bidirectional model transformations in a hippocratic manner. The advantages of these approaches regarding performance are validated using seven case studies.

An extension of the modeling language can help further simplifying model analyses, thereby also improving their performance characteristics.

ISSN 1867-0067
ISBN 978-3-7315-0763-5

Gedruckt auf FSC-zertifiziertem Papier

ISBN 978-3-7315-0763-5
9 783731 507635 >