Incremental Model Synchronization

by

Ali Razavi Nematollahi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2012

© Ali Razavi Nematollahi 2012
I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.
Abstract

Changing artifacts is intrinsic to the development and maintenance of software projects. The changes made to one artifact, however, do not come about in isolation. Software models are often vastly entangled. As such, a minuscule modification in one ripples inconsistency through several others. The primary goal of the this thesis is to investigate techniques and processes for the synchronization of artifacts in model driven development environments in which projects comprise manifold interdependent models, each being a live document that is continuously altered and evolved. The co-evolution of these artifacts demands an efficient mechanism to keep them consistent in such dynamic environments. To achieve this consistency, we intend to explore methods and algorithms for impact analysis and the propagation of modifications across heterogenous interdependent models. In particular, we consider large scale models that are generated from other models by complex artifact generators. After creation, both the generated artifacts, and also the ones they are generated from, are subject to evolutionary changes throughout which their mutual consistency should be maintained. In such situations, the model transformation is the primary benchmark of consistency rules between source and target models. But the rules are often implanted inside the implementation of artifact generators and hence unavailable. Trivially, the artifacts can be synchronized by regeneration. More often than not however, regeneration of such artifacts from scratch tends to be unwieldy due to their massive size. This thesis is a summary of research on effective change management methodologies in the context of model driven development. In particular, it presents two methods of incrementally synchronizing software models related by existing model transformations, so that the synchronization time is proportional to the magnitude of change and not to the size of models. The first approach treats model transformations as black-boxes and adds to it incremental synchronization by a technique called conceptualization. The black-box is distinguished from other undertakings in that it does not require the extraction, re-engineering and re-implementation of consistency rules embedded inside transformations. The second approach is a white-box approach that uses static analysis to automatically transform the source code of the transformation into an incremental one. In particular it uses partial evaluation to derive a specialized, incremental transformation from the existing one. These two approaches are complementary and together support a comprehensive range of model transformations.
Acknowledgements

I would like to acknowledge my profound, but hopefully not prolix, gratitude to many individuals whose support and advice were greatly conducive to this project.

Kostas Kontogiannis, my PhD supervisor, has provided me with his gracious support for over eight years, not just for overcoming academic difficulties, but in matters that shape one's life and character. I truly feel indebted to him. Ladan Tahvildari, my PhD co-supervisor, has helped me on numerous occasions academically and otherwise by graciously sharing her invaluable personal experiences with me. I would like to express my sincere gratitude to professors Rudolph Seviora, Paulo Alencar and Derek Rayside for agreeing to serve on my PhD committee and for taking the time to read my proposal and this dissertation. Special thanks goes to professor Eleni Stroulia from the university of Alberta, for agreeing to be my external examiner.

Funding for this research was in part provided by IBM Center for Advanced Studies. As a IBM CAS PhD Fellow, I had the privilege to attend three summers in IBM Canada Lab which was a fulfilling experience. I would like to thank Chris Brealey for seeding the initial idea of the project and providing insightful feedback and valuable collaboration along the way.

I would like to express my sincere gratitude to Professor Ali Safavi, whom I have been fortunate enough to have as a roll model since my early childhood. I very much appreciate the father-like advices and emotional supports he has given me time and again throughout these years. Finally, nothing can synchronize a prose attempting to describe my love and admiration for my parents with the real sentiment, depth of which goes far beyond the expressive power of words.
Dedication

Dedicated to my parents.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>xiii</td>
</tr>
<tr>
<td>Glossary</td>
<td>xiv</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Problem Description</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Summary of Contributions</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Thesis Outline</td>
<td>7</td>
</tr>
<tr>
<td>2 Related Work</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Perspective on Model Change</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Change Propagation</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Metamodel Evolution</td>
<td>10</td>
</tr>
<tr>
<td>2.3.1 Model Transformations</td>
<td>10</td>
</tr>
<tr>
<td>2.3.2 Model Consistency and Dependency</td>
<td>13</td>
</tr>
<tr>
<td>2.4 Model Refactoring</td>
<td>15</td>
</tr>
<tr>
<td>2.5 Partial Evaluation</td>
<td>15</td>
</tr>
<tr>
<td>3 Characterization of Change</td>
<td>17</td>
</tr>
<tr>
<td>3.1 Model Elements, Models and Metamodels</td>
<td>17</td>
</tr>
</tbody>
</table>
3.1.1 Equivalence of two model elements ........................................... 22
3.2 Atomic Model Manipulation Operators ........................................... 25
3.3 Change Factorization and Model Edit Distance .............................. 27
   3.3.1 Edit Script Normalization ............................................... 33
   3.3.2 Impact set of a Change .................................................. 34
   3.3.3 Model Transformations .................................................... 41
   3.3.4 Classification of Transformations ...................................... 41
   3.3.5 Traces and Dependencies across Transformations ................. 47
   3.3.6 Semantics of Synchronization for Generated Artifacts ............ 48
   3.3.7 Bi-directional synchronization ......................................... 48
   3.3.8 Incremental synchronization ............................................ 49

4 Incremental Synchronization of Black-box Transformations .............. 50
   4.1 Architecture Overview ..................................................... 51
      4.1.1 Overall Process ....................................................... 53
      4.1.2 Conceptualization Phase ............................................. 56
      4.1.3 Shadow Phase .......................................................... 57
      4.1.4 De-Shadow .............................................................. 61
      4.1.5 Synchronization Process .............................................. 62
   4.2 Insertion and Deletion ....................................................... 67
      4.2.1 Propagation of Insertion Induced Changes ....................... 67
      4.2.2 Propagation of Deletion .............................................. 76
      4.2.3 Dependency Inference ............................................... 76
   4.3 Complete Picture for The Running Example ................................ 78
   4.4 Properties of the Black-box Synchronizer ................................ 81
      4.4.1 Termination ............................................................ 81
4.4.2 Transformation Chains ............................................ 82
4.4.3 Soundness ............................................................. 86
4.5 Complexity .............................................................. 87
4.6 Experiments and Evaluation ......................................... 88
  4.6.1 Experiment Setup .................................................. 88
  4.6.2 Results ............................................................... 89
4.7 Application and Integration ........................................ 91

5 White-Box Incrementalization of Transformations .................. 96
  5.1 Introduction to Partial Evaluation ................................. 97
  5.2 Introduction to Query View Transformation Operational Mappings .... 101
    5.2.1 Model Transformation with QVT Operational Mappings ........ 102
  5.3 Syntax of Essential QVT-OM .................................... 106
    5.3.1 Abstract Syntax ................................................. 106
  5.4 Overall Process and System Architecture .......................... 107
    5.4.1 Partial Evaluation of Model Transformations .................. 111
  5.5 Partial Evaluation ................................................... 114
    5.5.1 Offline Binding Time Analysis Rules .......................... 115
    5.5.2 Online Binding Time Analysis Rules ........................... 118
    5.5.3 Constant Propagation and Static Expression Evaluation ......... 119
    5.5.4 Loops ............................................................. 119
    5.5.5 Conditional Expressions ...................................... 121
    5.5.6 Symbolic Expression Simplification ............................ 122
    5.5.7 Reduce Rules ................................................... 123
    5.5.8 Contexts and Function Calls ................................... 124
    5.5.9 Reducing Iterate Expressions ................................ 127
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5.10 Mixed Reduction of Partially Static Collections</td>
<td>128</td>
</tr>
<tr>
<td>5.5.11 Mix Rules</td>
<td>131</td>
</tr>
<tr>
<td>5.5.12 Partial Evaluation of Mapping Operations</td>
<td>135</td>
</tr>
<tr>
<td>5.5.13 Polyvariant Mix Rules</td>
<td>140</td>
</tr>
<tr>
<td>5.5.14 Polyvariant Reduction of Iterate Expressions</td>
<td>141</td>
</tr>
<tr>
<td>5.6 A Full Example</td>
<td>143</td>
</tr>
<tr>
<td>5.7 Experiments and Discussion</td>
<td>147</td>
</tr>
<tr>
<td><strong>6 Conclusion</strong></td>
<td>151</td>
</tr>
<tr>
<td>6.1 Future Work</td>
<td>152</td>
</tr>
<tr>
<td>6.1.1 Information Content of Models and Model Transformations</td>
<td>152</td>
</tr>
<tr>
<td>6.1.2 Improved Conceptualization Schemes</td>
<td>153</td>
</tr>
<tr>
<td>6.1.3 Automatic Generation of Abstractors/DeAbstractors</td>
<td>153</td>
</tr>
<tr>
<td>6.1.4 Embedded Rule Language for Composite Concepts</td>
<td>154</td>
</tr>
<tr>
<td>6.1.5 Fine-Grained Version Management Using Concepts</td>
<td>154</td>
</tr>
<tr>
<td>6.1.6 Enhancements to QvtMix</td>
<td>154</td>
</tr>
<tr>
<td>6.1.7 Self Applicability of QvtMix</td>
<td>155</td>
</tr>
<tr>
<td><strong>Appendix</strong></td>
<td>156</td>
</tr>
<tr>
<td>A Haskell Implementation of Change and Sync</td>
<td>156</td>
</tr>
<tr>
<td>B Abstract Syntax of QVT Operational Mappings</td>
<td>170</td>
</tr>
<tr>
<td>C Big-Step Operational Semantics of QVT Operational Mappings</td>
<td>174</td>
</tr>
<tr>
<td>C.1 Big-Step Operational Semantics</td>
<td>174</td>
</tr>
<tr>
<td>D QvtMix Implementation</td>
<td>182</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Problem Description ................................................................. 4

3.1 Sample Model Abstracting Java Code ........................................ 18

3.2 Runtime and Space Complexity of Dynamic Programming Change Factorization Algorithm. X-Axis is $n$, the size of the randomly generated model, and Y-Axis measures the value of $O(d)$ as represented in Theorem 3.3.1 with a constant of 3, the actual size of the table is represented by $|d|$, the product of the size of two models is represented by $|s||t|$ for comparison. $T$ is the runtime of the algorithm which is normalized and projected for comparison with the space complexity ........................................ 33

4.1 Architecture of the Synchronization Framework (arrows denote dataflow) .... 52
4.2 Generating WSDL from Java Classes .......................................... 53
4.3 Simplified metamodel for Java Implementation of a Web Service .......... 53
4.4 Simplified metamodel of WSDL 2.0 ............................................. 54
4.5 Abstract Models of Java and WSDL .......................................... 55
4.6 Conceptualization of Java and WSDL models ............................... 58
4.7 Creating Shadow for Java model ................................................. 60
4.8 Transformation of Java Shadow to obtain WSDL Shadow ................. 61
4.9 Synchronization by Shadow .......................................................... 63
4.10 Injecting $\mu$-templates .......................................................... 70
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.2</td>
<td>Simplified Metamodel of Imperative OCL</td>
<td>172</td>
</tr>
<tr>
<td>B.3</td>
<td>Simplified Metamodel of OCL</td>
<td>173</td>
</tr>
</tbody>
</table>
Glossary

.: therefore.

\((a_1, \ldots, a_n)\) n-tuple consisting of elements \(a_1, \ldots, a_n\).

\(\langle e_1, \ldots, e_n\rangle\) list (ordered sequence) of elements \(e_1, \ldots, e_n\).

\(|A|\) size of a set or list.

\(\pi_i(S)\) projection of the \(i\)th element of a tuple or list.

\(\{m_1, \ldots, m_n\}\) set of elements \(m_1, \ldots, m_n\).

\(\emptyset\) empty set or list.

\(a \in S\) set or list membership.

\(2^A\) power-set (set of all subsets) of \(A\).

\(A \times B\) cartesian product of sets \(A\) and \(B\).

\(A \cup B\) union of sets \(A\) and \(B\).

\(S \psi T\) catenation of lists \(S\) and \(T\).

\(f : A \rightarrow B\) function \(f\) with domain \(A\) and co-domain \(B\).

\(f \circ g\) function composition \(f \circ g(x) = f(g(x))\).

\(f ; g\) function composition \((f ; g)(x) = g(f(x))\).

\(\perp\) undefined value.
□ place holder for any value.

\(O(g(n))\) big-O, asymptotic upper bound growth of \(g(n)\).

\(\mathbb{N}\) Natural numbers (positive integers).

\(\Sigma\) Alphabet of symbols.

\(\Sigma^*\) Transitive closure of alphabet \(\Sigma\). Set of all possible permutations.

\(\mathcal{L}\) language \(\mathcal{L} \subseteq \Sigma^*\) (c.f. \(\Sigma^*\)).

\(\mathcal{L}(\text{MM})\) language generated by meta-model \(\text{MM}\), set of all instances.

\((\text{Sig}_C, \text{Sig}_A, \text{Sig}_R)\) metamodel with signature functions for containers, attributes and references, respectively.

\((\mathcal{C}, A, R, T)\) model element of type \(T\) with container list \(\mathcal{C}\), attribute list \(A\) and reference list \(R\).

\(v : T\) value \(v\) has type \(T\).

\(\mathcal{T}\) set of all types.

\(\mathcal{M}\) set of all models.

\(\mathcal{C}\) set of all containers.

\(\mathcal{C}\) list of containers of a model element (c.f. \(C_i\)).

\(C_i\) the \(i_{th}\) container.

\(m/i\) the \(i_{th}\) container of model \(m\).

\(c.j\) the \(j_{th}\) element of container \(c\).

\(m@k\) the \(k_{th}\) attribute of model \(m\) (or attribute named \(k\)).

\(\mathcal{C}(m), A(m), R(m), T(m)\) containers, attributes, references and type of model \(m\), respectively.

\(\text{Addr}_M(m)\) Address of element \(m\) in model \(M\).
∪ ℂ union of all containers in ℂ.

≥ ℂ all elements contained by ℂ.

m = n equivalence of models m and n.

m ≃^{M,N} n congruence of model elements m and n residing respectively in models M and N.

m ≃ n weak equivalence of m and n.

♦(m, i, v) update the value of the i_th attribute of model element m to v.

♦(m@a → v) update the value of attribute a of model element m to v.

♦⊥ Reset operator, changing all values of its argument model to ⊥.

▲(m, i, T) insert new element of type T into the i_th container of model element m.

▲(M/α) insert into container address M/α.

▼(m, i) delete the last element of the i_th container of model element m.

▼(M/α) delete the last element of the container at address α.

J^\Delta_M impact of change Δ on model M.

m ↦^δ n m morphed into n by change δ.

δ ↦^T δ’ change δ translated to δ’ by transformation T.

Clone(M) duplicate model M (and all its children).

ℂ̂P concept pool.

a ↣ c attribute a points to concept c.

μ micro template.

[[prog]]ₐ[iₚ] = out program prog, interpreted in language ℒ, produces out from in.

[[x]] x quoted as expression.
Exp  set of all expressions.

Env  set of all environments.

τ  binding-time environment.

α  annotation environment.

Γ  type environment.

σ  variable store.

σ[a → v] new store mapping a to v and every other x to σ(x).

S  static binding-time.

D  dynamic binding-time.

M  maybe static binding-time.

VAR  variable annotation.

VAR⁺  insertion change annotation.

VAR⁻  deletion change annotation.

VAR♦  update change annotation.

FIXED  Invariant annotation.

t₁ △ t₂ binding-time conjunction.

t₁ ⊕ t₂ binding-time disjunction.

B[.]  online BTA function.

E[.]  eval function.

SE[.]  side-effect eval.

R[.]  reduction.

M[.]  mix reduction.
\(\mathcal{PM}[[.]]\) poly-variant mix reduction.

\(e \, \triangleright \, e'\) merge of expressions \(e\) and \(e'\).

**Context** set of all contexts.

\(\Psi\) memoization table.

\(\Psi\) space of memoization tables.

\(\zeta\) helper environment, keeping the description of all helper functions.

\(\xi\) mapping environment, keeping the description of all mapping functions.

\(\sigma; e \downarrow \nu\) large-step reduction; expression \(e\) reduces to value \(\nu\) in environment \(\sigma\), which will remain unchanged.

\(\langle \sigma, e \rangle \downarrow \langle \sigma', \nu \rangle\) large-step reduction; expression \(e\) reduces to value \(\nu\) in environment \(\sigma\), during which it will be mapped to \(\sigma'\).

\(-- \, don't \, care.\)

\(\emptyset\) looping flag.

\(\downarrow \) break flag.

\(\downarrow\) proceed flag.

\(\uparrow\) return flag.
Chapter 1

Introduction

The principal aspiration of Model Driven Development (MDD) is to raise the level of abstraction in software development. Models are denoted using diagrams and graphical notations. As such, they potentially are able to convey requirement and design information more succinctly in comparison with textual formats. MDD aims to let the stakeholders of all levels concentrate on solving the problem at hand, rather than overcoming the difficulties imposed by low-level intricacies of computation [44, 6, 68, 55]. Compared with code-centric approaches, the model driven paradigm of software development is shown to significantly boost productivity in spite of its relative immaturity (for a comprehensive empirical evaluation of UML effectiveness see [23]). In its envisioned usage, MDD deems requirement, architecture and design documents as live and evolving artifacts [38]. This comes in contrast with the traditional software development lifecycles during which such models are often inanimate documents.

Model driven projects, compared with code-centric ones, encompass more diverse types of artifacts. The heterogeneity of MDD environments stems from several key characteristics of this approach towards software development. First, modeling facilities, such as UML, offer an assortment of means for specifying orthogonal aspects of a system. UML version 2.0, for example, provides 13 different types of diagrams [56], each dedicated to model a single facet of the system. In particular, UML Class diagrams describe the structural decomposition of classes and their static relationships with each other. Other examples are Sequence diagrams whose primary intent is to model dynamic interactions between objects, and Deployment diagrams that describe the logistics and distribution of software
components. These diagrams share both latent and obvious cross dependencies. For example, the objects modeled in a sequence diagram have the signature of their classes defined in a corresponding class diagram. Therefore, altering the name of a class, for example, requires the sequence diagram and other related models to be updated accordingly.

Second, MDD advocates the encapsulation of domain knowledge in the form of domain specific languages (DSL). The syntax and semantics of these languages are usually specified as models themselves which in part are assets to projects that take advantage of them. In addition, there are domain specific models specified in these languages. Because the DSLs being used for modeling are often developed in-house, they are prone to change frequently. Changing a language compels its instances to comply with its new syntax or semantics, thus they should also be modified accordingly.

The third reason for the diversity of artifacts is the support for multiple platforms needed in many recent projects. Software systems are becoming more distributed which means there are more than one architecture involved in most projects. Usually a variety of platforms with different operating systems and middle-ware are deployed. Each platform, to play its particular role in the operation of the system, depends on various platform specific documents, such as deployment descriptors, schemas etc., for proper configuration and deployment. MDD, because of its promises to abstract the complexity of various platforms, is often touted as the paradigm of choice for developing such projects. Consequently, the platform specific documents are also treated as models, hence making the projects even more multifarious.

Throughout their life-cycle, software artifacts are iteratively changed. Being logical interdependent entities, changes made to one model would impact other models. Modern development environments strive to provide facilities for transparent and effective propagation of changes across the workspace. Being diverse however, means that the task of synchronizing models is significantly more challenging in MDD environments in which manifold of models are subject to miscellaneous transformations. To ensure the homeostasis of systems, their models ought to remain consistent according to an assortment of rules imposed by languages, grammars, meta-models, constraints and transformations.

A development environment should be responsive and interactive. They are meant to help the developers achieve their goals without impeding their productivity. Change

\footnote{Software Homeostasis in the sense that Mary Shaw in \cite{homeostasis} has defined as: “Homeostasis is the mechanism through which a system acts to maintain a stable internal environment despite external variations.”}
propagation and subsequent model synchronization tasks thus have to be carried out as transparently to the user as possible. These requirements motivate that the synchronization of large models be performed incrementally, i.e., the synchronizer should only reconcile the affected fractions of models and bypass the unchanged elements. The system should also facilitate analyzing the impact of a change, and let developers preview its alternative outcomes before conducting it. This analysis is particularly useful for manipulating large models.

Generative software development uses transformations to create software artifacts of different types from one another [21]. The consistency rules of these artifacts are often embedded inside the source code of the artifact generator. These generators are not always thoroughly documented. Moreover, development processes occasionally integrate third party transformations, no information about whose internal logic is available. For the same reasons mentioned earlier, to synchronize efficiently the models generated with these model transformations, is a desirable feature. Nevertheless, reverse engineering the transformation rules for these artifact generators tend to be project specific and take time and resources. Any update propagation method capable of incrementally reconciling generated models—without the need to reverse engineer and re-implement them—has considerable utility.

1.1 Problem Description

The objective of change management is to co-evolve a set of models whose consistency is disturbed by local changes towards a new consistent state. Consistency between two interdependent models can, in general, be described using a relationship defined over the state space of the mutations of those models. Change management process relies on the ability to synchronize two models based on a given consistency relationship between them.

Change management relies on synchronization as one of its fundamental building block. Therefore, to tractably manage changes between large models with complex relationships, model synchronization has to be incremental. This, in principle, requires the complexity of propagating changes be proportional to the size of changes, and not that of the source and target models. In other words, the synchronizer should manipulate only the parts that are affected by the changes; any redundant rewrite of the parts that need no modifications
should be avoided. This, to some extent, is analogous to the concept of compilation with the make utility, which only recompiles the most recently changed modules. Only, the level of granularity for model synchronization is reduced down to the individual model elements from code modules in the case of program compilation.

Figure 1.1 illustrates the problem of model synchronization in a more precise manner. In this N is the target model which is generated by applying transformation $T_{Gen}$ on the source model, namely M. The consistency relationships between models can originate from different sources. One particularly important category is when these relationships are implicitly enforced by a model transformation. For example in Figure 1.1, the consistency requirement is connoted by a transformation that has originally been used to generate (or update) the target model from the source model. As such, future consistency, when needed, can be established by re-invoking that transformation. Although, these transformations are often non-incremental—hence suffer from the problems mentioned earlier—they, nonetheless, can serve as an objective measure of correctness for change propagation. This, in other words, asserts that model synchronizers should incrementally produce updated models that are compatible with the result of the original model transformation, should it run again.
over the altered source model.

Transformations can be mathematically described as mappings between domains, which are specified by metamodels. Analogous to the terminology of formal languages, a metamodel (or a domain model) can be considered as a generative grammar for a domain; \( \mathcal{L}(MM) \) denotes the language generated by metamodel specification \( MM \), i.e., the (possibly infinite) set of all models conforming to \( MM \).

Changes in artifacts can be described as endogenous transformations, that is, transformations whose source and target models conform to the same metamodel. \( \Delta : \mathcal{L}(MM) \mapsto \mathcal{L}(MM) \) defines transformation \( \Delta \) applicable to models conforming to metamodel \( MM \). Given this, interdependent model synchronization becomes the problem of finding the most concise change operation \( N' = \Delta_2(N) \) applicable to model \( N \in \mathcal{L}(NN) \) for change \( \Delta_1(M) = M' \) applied on model \( M \in \mathcal{L}(MM) \), such that \( N' \) and \( M' \) remain consistent according to consistency criteria the same as those between \( M \) and \( N \). By concise, we mean that \( \Delta_2 \) should consist of the minimal number of change operations whose collective impact is confined to the affected elements of \( N \).

Artifact generation is an example of exogenous model transformation, that is, its source and target models adhere to different metamodels. Mapping \( T_{Gen} : MM \mapsto NN \) transforms source models that conform to metamodel \( MM \) to target models conforming to metamodel \( NN \). Assuming that models \( M \) and \( N \) are synchronized—i.e., they are consistent according to some relation \( R \)—a simple approach to re-synchronizing \( M' \) and \( N \) is to re-apply \( T_{Gen} \) on \( M' \) to obtain \( N' \). However, when \( M \) is large enough relative to \( \Delta_1 \), and/or \( T_{Gen} \) is computationally expensive, this approach based on indiscriminate re-generation would not be sufficiently performant for modern, interactive development environments. Furthermore, regeneration does not provide any means for reflecting the changes made to the target model back to the source model. As the diagram depicts, a mapping such as \( Sync \) could be used to generate from \( \Delta_1 \) a sequence of change operations, \( \Delta_2 \), which is applicable to \( N \), and yields a model identical to \( N' = T_{Gen}(M') \), which is consistent with \( M' \) according to the same relation, \( R \). We say that \( Sync \) is incremental, if the number of elements in \( N \) that \( \Delta_2 \) modifies to obtain \( N' \) is minimal. Thus, the problem of model synchronization for the models bound together with transformation rules such as \( M = T_{Gen}(N) \) reduces to determining \( \Delta_2 : MM \mapsto NN \) for every change \( \Delta_1 \) defined on the source model, such that \( \Delta_2(N) = T_{Gen}(\Delta_1(M)) \).
1.2 Summary of Contributions

In this section we present an outline of the major contributions made in this research work.

- **Foundations**
  - Abstract and formal characterization of models, metamodels, transformations and change operations. We define a succinct notation to express MOF like models using mathematical constructs. We also define atomic change operations, which serve as building blocks for composing general manipulation of model structures.
  - Two efficient algorithms for computing edit distance of models based on the insert/delete and the append/drop change operations. The first algorithm leverages the definitions of models and atomic change operations to first find a series of changes to make both models structurally similar, and then update attribute values where there are discrepancies. The second algorithm uses a dynamic programming approach to find a minimum-cost edit script based on the cost associated with each atomic change operation.
  - Algebraic solution for the simplification of composite change scripts. This methodology is used to obtain canonical forms for edit scripts, in which changes whose effects are cancelled by later changes in the script are eliminated.
  - Characterization of the impact set of a change and presenting an algebraic, mechanical solution for change impact analysis, whereby making it possible to calculate the impact-set of a complex change \textit{a priori} before applying the change operations.
  - Classification of model transformations based on their change translation behavior and proving several important properties
  - Formal definition for incremental and bi-directional synchronization schemes
  - Canonical implementation of models, metamodels and change operations in Haskell

\footnote{Parts of this work has been published in these peer-reviewed conferences \cite{61, 63, 62}, and, at the time of this writing, several hitherto unpublished parts are being prepared as manuscripts for submission.}
• **Black-box synchronization of model transformations**
  
  – Sync: a novel methodology for the synchronization of blackbox model transformations
  
  – Analysis of completeness, soundness and efficiency of the proposed black-box synchronization scheme
  
  – Canonical implementation of black-box synchronization in Haskell
  
  – Implementation and integration with Eclipse Web Tools Platform (WTP) and application to real-world transformation scenarios of service oriented artifacts
  
  – Conducting experiments for performance evaluation of the framework

• **White-box synchronization of model transformations**
  
  – Methodology for the White-box synchronization of imperative model transformations based on partial evaluation
  
  – Design and specification of several reduction algorithms for the specialization of model transformations
  
  – Design and implementation of QvtMix: a hybrid partial evaluator for QVT Operational Mappings implemented in QVT itself
  
  – Big-step operational semantics for a subset of QVT Operational Mappings
  
  – Experiments for the characterization of the performance and the space overhead of the partial evaluator

1.3 **Thesis Outline**

The remainder of this dissertation is organized as follows. A comprehensive survey of related research and technologies is presented in the next chapter. Chapter 3 presents preliminary materials and establishes the foundation for the rest of the thesis. It includes, among other topics, the discussion of change factorization algorithms, change script simplification, change impact analysis and the definition of incrementality for model synchronization. Chapter 4 introduces the black-box synchronization techniques, and discusses various properties of the black-box framework. Chapter 5 gives an elaborate account of
the proposed techniques for white-box model synchronization. QvtMix, a partial evaluator for the QVT Operational Mappings transformation language, is presented to introduce several partial evaluation techniques, their application in the context of model transformations, and how they aid us achieve incrementality for model synchronization scenarios. Chapter 6 summarizes the dissertation and confers on several avenues for further research.

We believe that, in the age of inexpensive bits and storage, no software engineering manuscript is complete without supplementary proof of concept implementation code. This helps make the document self-contained, and improves the reproducibility of claimed results. As such, we present canonical implementation for the crux of the algorithms presented here in three appendices. More specifically, the minimalist and semi-formal notation we have established for models and change operators, along with the most fundamental parts of the black-box synchronization framework, are implemented in Haskell, which is presented in Appendix A. Appendix B presents the elaborate implementation of QvtMix. The code for the dynamic-programming change factorization algorithm is presented in Appendix E. Moreover, throughout the main body of the thesis, various pointers have been inserted to the electronic version of this document, so as to enable the reader quickly navigate to pertinent parts of the implementation.

Some peripheral material regarding the QVT-OM language are provided as appendix: an elaborate abstract syntax of the language which illustrates its relationship with other parts of OMG’s ecosystem of modeling standards (in particular, MOF and OCL) are presented in Appendix B. Appendix C specifies the semantics of the subset of the QVT-OM language used for discussing the design of QvtMix.
Chapter 2

Related Work

2.1 Perspective on Model Change

There are two major perspectives on change. The first one, the state-based view, considers changes as morphisms that map the state of the model they are applied upon to a new state in the general state-space of instance models outlined by the metamodel. The second school of thought is what is referred to as the operational view, which defines a set of change operations and represents the differences between models as a sequence of those operations. Examples of frameworks that have adopted the latter representation include: Lenses in Harmony [28], Rondo [50], and most graph transformation based frameworks, e.g., AGG and VIATRA. Alternatively, some examples of frameworks which incorporated the operational perspective include: the works of Porres and Alanen in [60] and [4], ATL and SyncATL [81], Deltaware [36], Beanbag [82] and the paper by Blanc et al [13]. Our notion of change, introduced in Chapter 3, is operational, which defines 5 primary atomic change operations composeable into arbitrary changes.

2.2 Change Propagation

Model driven projects typically comprise several inter-related models, and therefore modification of a model may cause several violations of the consistency relationships between the inter-related models in a project. To reconcile the system back into a consistent state,
it is needed to appropriately propagate the changes to other models that are invalidated by the recently performed change operations. Other terms commonly used for change propagation in the MDD community are: Model Synchronization [31], [81] and [6], update transformation [60], update propagation [17] and update translation [28].

### 2.3 Metamodel Evolution

In MDD, everything is a model. This general rule also applies to metamodels and, therefore, they can also be modified using the same scheme developed for changing models. However, metamodels are essentially grammars that describe a space of instances. As such, changing a metamodel can invalidate some of its former instances, extend the instance space of said metamodel, or do a combination of both. Changing metamodels along with the repercussions of such changes on their instances have been explored in the literature. Wachsmuth proposes definitions for metamodel adaptation and model co-adaptation in [78], in which some important properties of metamodel changes, such as Instance Preservation, are discussed. Another related term in this context is Model Co-evolution, which is used to describe the changes of instance models along with those of their metamodels in a coupled evolutionary development process, during which the conformance of models to their metamodels have to be preserved [16], [37] and [45]. In this paper, we have assumed that during the life-cycle of the software artifact maintained by the synchronization framework, their metamodel remains unchanged. However, supporting such changes, is a feature worthy of future research.

#### 2.3.1 Model Transformations

The major premise of our work is to avoid re-implementation of an existing transformation in another language or framework. This sets an immediate diverging point between our approach and that of other incremental synchronization frameworks, majority of which define some sort of specification mechanism, whereby an existing transformation has to be re-implemented. In contrast, we take the transformation’s implementation as a blackbox, and build the support for incremental synchronization around it, only assuming a few generic properties about the transformation.
A catalogue of various model synchronization schemes is offered in [6]. The paper focuses on formulating the external behavior of model transformations. A general classification of model transformation frameworks can be found in [20]. Two features that specifically concern the problem of model synchronization are bi-directionality and incrementality. The paper introduces several frameworks that support either or both of these features, most notable of which is the Object Management Group’s Query View and Transformation (QVT) [10]. QVT proposes the Relation and the Core languages, both of which capable of denoting incremental and bi-direction transformations with different expressiveness and complementary levels of abstraction.

Triple Graph Grammar (TGG) is a graphical, declarative, incremental and bi-directional model transformation methodology based on graph transformation [67]. It has been advocated to be an effective foundation for tool integration [5]. Beanbag is an emerging framework which offers a language that supports intra-relations between models [82]. Our conceptualization scheme also supports intra-relations.

Bi-directional transformations and their application in model synchronization have been investigated by researchers in the programming languages community. Foster et al. proposed the Harmony framework based on the notion of Lenses for bi-directional synchronization [28]. They propose a language in which programs are inherently bi-directional. The Harmony framework has a state-based perspective on change. Contrariwise, Alanen and Porres have investigated syntactic merging, differentiation and union of structural models in [4] by adopting an operational and compositional view of changes.

One solution that works with existing transformations is SyncATL, proposed by Xiong et al. in [81]. They have extended the bytecode of the ATL virtual machine [8], whereby supporting automated backward synchronization of models linked by an ATL transformation. For the forward synchronization, SyncATL relies on re-invoking the transformation and merging the results with the existing target. The proposed technique, however, does not address incremental synchronization at all, as the framework relies on re-executing the transformation for forward change propagation. The other drawback of this approach is its tight integration with a specific technology, i.e., ATL.

Another framework with the theme of building incremental synchronization around existing transformation engines is presented in [36]. This time the Tefkat transformation engine, which has logical programming flavor, is decorated with support for incrementality.
In Tefkat, transformation rules are specified as logical predicates, and are performed using SLD resolution. Their synchronization framework avoids redundant computation in successive transformation of the same model by preserving the intermediate SLD trees. Another logic based framework which exploits answer set programming for change propagation is presented in [15].

In [76], Tratt articulates the spectra of challenges involved in model driven tool integration with model transformations being their centerpiece. The paper stresses the importance of the particularly challenging problem of change propagation and enumerates the reasons for inadequacy of solutions based unilaterally on commonly championed panaceas such as bidirectional transformations or traceability. He concludes that a comprehensive change propagation scheme, to be pragmatically effective, should function tractably over the most common patterns of transformations. Another white box approach for incremental execution of program transformations is based on the concepts of chip and chop proposed by Sittampalam et al. [71]. The proposed approach is, however, concentrated on the incremental execution of transformation of executable specification with an emphasis on behavioral preservation. As such, it is more fitting to the context of code refactoring than that of data model synchronization.

The problem of incremental model synchronization parallels the extensively investigated, yet in some degrees open, problem of view maintenance in databases. Two noteworthy approaches to this problem are presented in [32] and [34]. The view maintenance problem, along with the view update problem, have inspired software engineers to look at interrelated software artifacts as essentially different views of a common database. The bi-directionality of synchronization can thus be remedied by solutions transpired for the view update problem, and incremental synchronization becomes analogous to view maintenance. In spite of similarities, there are also key differences between maintaining database views and synchronization of software models that warrant a distinct research agenda dedicated to the latter. Briefly, database views are defined in few, precise view definition languages, upon which the database community has come to a unanimous consensus. In comparison, model transformation frameworks are rather diverse and immature. Furthermore, the relational database model denotes flat structures. In contrast, models comprise containment hierarchies which pose a semantic for deletion that can be peculiar to handle using purely relational techniques.

Finally, CLIME [64] and MView [33] are constraint based consistency management
frameworks for incremental maintenance of software artifacts. These frameworks rely on the existence of a well-defined set of constraints for ensuring the consistency of the interlinked models, and are capable of incremental resolution of inconsistencies for such cases.

The general methodology as to how interdependent models should be incrementally synchronized is outlined in [40]. The proposed approach, however, requires tight integration with a specific modeling technology, and needs detail knowledge of transformations.

### 2.3.2 Model Consistency and Dependency

Model Dependency is the problem of specifying and analyzing dependencies between model elements either within the scope of one model (Intra model), or across the boundaries of two inter-related and possibly heterogeneous models (Inter model).

In MDSE, developers use different models for specifying different aspects of a system hence the dispersion of information across multiple documents. It is important to assess the consistency of different but inter-related models to ensure accuracy of models and prevent occurrence of conflicting patterns and behavior discrepancy in the system. Model consistency is the area of research in modeling that deals with identification, formalization and specification of consistency rules and also provides verification and reconciliation mechanisms to ensure the satisfaction of such constraints.

Description Logics [14] is used as a notation for representing consistency rules between models [73] [72] [43]. Keefe used Dynamic Logic, an extension of model logic intended for reasoning on dynamic behavior of computer programs, as basis of a framework for model consistency [57]. Dynamic Logic broadens model logic with two special model operators that denote post-occurrence necessary and possible predicates for actions. Utilizing these operators, Keefe is able to establish consistency rules between UML class, sequence and state chart diagrams and diagnose inconsistencies by logical inference.

Fombelle et al. in [22] argue that pure active consistency management by monitoring at editing time tend to enforce consistency rules stringently and hence too prohibitive to let developers explore all possibilities which often require tolerating tentative inconsistencies. Instead, they adopt a hybrid approach that manages inconsistencies by defining a set of automata. Collectively, the automata embroil the state space of models into states that capture evolving model parts' consistency or lack there of. Transitions between the
states assert whether an inconsistency is being introduced, resolved or the current status is retained. They exemplified their approach by a scenario that involves class, state chart and sequence diagrams.

Sabetzadeh and Easterbrook in [66] proposed an approach for analyzing inconsistency of fuzzy viewpoints. Their framework is based on fuzzy set theory to model the viewpoints. They demonstrate that fuzzy viewpoints and fuzzy viewpoints morphisms constitute a finitely co-complete category which as a corollary entails that the objects of a category can be completely interconnected without the need for any additional gluing structures. They provide an abstract framework for analyzing inconsistencies during merging incomplete and inconsistent fuzzy viewpoints.

Another approach, proposed by Blanc et al., towards detecting model inconsistencies is to represent models by a sequence of constructive operations that incrementally build the model [13]. This view of models is imperative in nature in contrast to declarative representations conventionally formalized by sets and graph. Structural consistency rules are expressed as logical constraints on the sequences of constructive operations.

A powerful technique for detection and resolution of inconsistency conflicts in graph transformations is critical pair analysis [25] and [3]. Mens et al. have done extensive research work capitalizing on this technique to manage inconsistencies during model refactoring and model transformation scenarios [53], [54], [51] and [52].

Formal Concept Analysis is non-deterministic technique which used for the identification of dependency links between models of different levels of abstraction [30] and [27]. Ivkovic and Kotogiannis have adopted this sort of analysis to automatically establish dependency links between PIM and PSM models in a setting purported to the development of commerce applications [39].

Ensuring consistency of models and systems after modification, propagating changes to other software artifacts and model synchronization are overlapping areas of research that are progressively receiving more attention. These problems have been considered before in the context of traditional programming environments as well as data-base systems [33, 64, 33, 17, 1, 34].
2.4 Model Refactoring

Refactorings are behavior preserving modifications intended to improve the internal structure and design of the system without altering its observable interface. Refactoring for models is an important research area that aims to firstly provide similar capabilities of code refactoring facilities that programmers currently enjoy for model developers and also investigate novel opportunities for refactoring operations specific to modeling.

Correa and Werner investigate the common problems in OCL annotated UML models in [19]. They define a collection of OCL smells, analogous to the more notorious code smells blueprints and proceed to offer two categories of refactoring based solutions to resolve these culprit patterns. The first category is specific to OCL and the second one is UML diagram exclusive. The paper gives an outline for the possibility of automating the proposed refactoring in a UML tool. To carry out such refactorings, the OCL expressions have to be parsed as an instance of the OCL metamodel defined by OMG. The paper proposes to use an OCL-like language to perform the updates on the models but inasmuch as OCL is a sans side-effect language, it cannot be used to update graph instances per se. Thus they propose a modified metamodel for OCL to simply tree traversals and define an imperative scripting language that utilizes this modified OCL for queries.

Other works tackling various aspects of model refactoring include [47], [48], [49], [73], [53], [77].

2.5 Partial Evaluation

There exist a vast body of research on partial evaluation. An excellent introductory resource is the book authored by Jones et al. [42], which also provides an exhaustive list of references to the existing literature. Shorter entry point to the area of partial evaluation can be found in [41] and [18]. Sundaresh et al. [74] were amongst the first researchers to exploit partial evaluation for deriving incremental programs, albeit in the context of code-driven programming languages. The indexing model employed in [58] is similar to the scheme we have used for accessing the elements of cached collections. Most of the framework however focus on programming languages; we are not aware of any other research work that leverages partial evaluation techniques in the context of model transformation
languages. As we discuss in Section 5.5, there also tend to be differences in the specialization model transformations, that rely extensively on collection manipulation, and ordinary programs, wherein collection-based caching strategies are not as predominant.
Chapter 3

Characterization of Change

3.1 Model Elements, Models and Metamodels

In a Model Driven Development Environment, every artifact is a model described using a
metamodeling language. We adopt the notation $M_d : \text{MM}_d$ to express that model $M_d$ is
an instance of metamodel $\text{MM}_d$ in the context of domain $d$. Intuitively, a model is a set
of model elements.

For our purpose, model elements are instances of the types defined in the metamodel of
a model. In essence, model elements are information bearing entities that enclose primitive
and complex information in their attributes. They can also have several containers, each
containing other model elements. Model elements can also reference other model elements.
The following definition captures these characteristics.

**Definition 1 (Model Element)** Model element $m$ is a tuple $(C, A, R, T)$, in which $C$ is
the list (i.e., a strictly ordered multi-set) of containers, $A$ is the list of attribute values,$R$ is the list of references, $T$ is a mapping to the metamodel, which ascribes a type to the
model element.

We use the same letters as function names for the projection of individual components
of model elements, that is to say, $m = (C(m), A(m), R(m), T(m))$. It should be noted that
other typical features of models such as names, types and constraints of the attributes and
containers are all specified in the metamodel and not in the model element itself. This separation allows for type (and also name) agnostic manipulation of model elements.

Definition 1 is recursive; for the members of containment lists are assumed to be model elements themselves. We unify the definition of models and that of model elements by representing a model as a model element whose type denotes its metamodel. Differently put, models are regarded as root-level model elements, with only one container which contains all the top-level model elements.

Models are represented in various ways depending on the particular aspect of them that needs to be highlighted. In this chapter, we offer a two-fold representation of models and model elements, to wit, compositional and referential. These two views are complementary, and to fully specify the semantics of models, they should both be considered along side one another. Nevertheless, decoupling the two views significantly facilitates the analysis and formal reasoning of the algorithms we discuss in this chapter.

There also exists a pragmatic reason for imposing an order on the containers of a model elements. For many change management operations, it is essential to be able to efficiently tell apart two different models, and systematically characterize the differences by a list of edit operations. The general tree-alignment and tree-edit problem, for unordered-trees, has shown to be NP-Hard [11], but is tractable if the tree is ordered.

Figure 3.1 illustrates a sample model, which is an abstraction of Java code in the UML notation. The top-level element represents a Java class. Each class can contain a number of methods, each of which may contain a number of arguments. Furthermore, there are
two kind of references in the model: one is `returnType` which designates the return type of a method, and the other one is `type` signifying the type of a method parameter. Using Definition 1, the Java model of Figure 3.1 is as follows.

\[
c1 = (\langle m1 \rangle, \langle "SrvImpl" \rangle, \emptyset, \text{JClass})
m1 = (\langle p1 \rangle, \langle "method" \rangle, \langle \text{String} \rangle, \text{JMethod})
p1 = (\emptyset, \langle "str" \rangle, \langle \text{String} \rangle, \text{JParam})
\]

In this set of definitions, \( c1 \) is specified as a class that contains method \( m1 \). The second element of the tuple, i.e., "SrvImpl" is the value of the name attribute. In the definition of method \( m1 \) (and similarly parameter \( p1 \)) the third element of the tuple denotes the references specifying the return type of methods (or the type of parameters). For the case of \( m1 \) and \( p1 \), the reference refers to a primitive type, \text{String}.

As noted, information such as attributes and containers’ names as well as their types, can be obtained from metamodels. The definition we offered for models, however, is only concerned with instances. For our purpose, we use the following definition for metamodels.

**Definition 2 (MetaModel)** A metamodel is a 3-tuple, \( (\text{Sig}_C, \text{Sig}_A, \text{Sig}_R) \) consisting of three signature functions \( \text{Sig}_C : \mathbb{N} \rightarrow \Sigma^* \times \mathcal{T} \), \( \text{Sig}_A : \mathbb{N} \rightarrow \Sigma^* \times \mathcal{T} \) and \( \text{Sig}_R : \mathbb{N} \rightarrow \Sigma^* \times \mathcal{T} \), which respectively map each container, attribute and reference in the instance model to a tuple comprising its name and type.

In the definition above, the first argument of each signature function is the index of the attribute for which the meta-information (i.e., name and type) is inquired. The functions return a tuple respectively consisting of a name in \( \Sigma^* \) and a type in \( \mathcal{T} \), that is, the set of all types. For example, the following denotes the metamodel of Java code presented earlier in Figure 3.1.

\[
\text{Class} = (\langle\langle 1, ("methods", \text{JMethod}) \rangle, \langle\langle 1, ("name", \text{String}) \rangle, \emptyset \rangle)
\]

\[
\text{Method} = (\langle\langle 1, ("params", \text{JParam}) \rangle, \langle\langle 1, ("name", \text{String}) \rangle, \langle\langle 1, ("returnType", \text{JClass}) \rangle \rangle)
\]

\[
\text{Param} = (\emptyset, \langle\langle 1, ("name", \text{String}) \rangle, \langle\langle 1, ("type", \text{JClass}) \rangle \rangle)
\]

We define the following auxiliary operators on model elements: \( \bigcup \mathcal{C} \) is a union of all members of \( \mathcal{C} \) which in Definition 1 is defined as a nested list. \( \varnothing \mathcal{m} \) returns a flattened list
that contains all the elements that are directly or indirectly contained by model element $m$. Both operators are defined formally in the following. It should be noted that the definition of $\psi$ is recursive and uses $\bigcup$.

$$\bigcup \mathcal{C} \overset{\text{def}}{=} \bigcup_{C_i \in \mathcal{C}} C_i$$

$$\text{let } m = (\mathcal{C}, A, R, T), \quad \psi m \overset{\text{def}}{=} \begin{cases} \emptyset & \mathcal{C} = \emptyset \\ \bigcup_{m_i \in \bigcup \mathcal{C}} (\{m_i\} \cup \psi m_i) & \text{otherwise} \end{cases}$$

The semantics of containment for model elements is defined as $m \in \psi M$, needless to say $m \in \cup \mathcal{C}(M)$ implies $m \in \psi M$ but not necessarily the other way around; the latter is immediate containment by model (element) $M$, while the former allows containment by any of $M$'s children as well as itself. The following is an example of these two operators on the example Java model.

$$\bigcup \mathcal{C}(\text{Class}) = \{\text{Method1}\}$$

$$\psi \text{Class} = \{\text{Method1, Param1}\}$$

The compositional view of a model (element) represents how all of its contained elements and, likewise, their own contained elements, are structured. For the sake of precision, we should note that the standard set operations, e.g. union and intersection, when applied to lists, implicitly convert the list to multi-sets, by dropping the order, and thus result in multi-sets.

In the realm of type theory, there are two general approaches toward type checking of objects in a type system\[59\]. One is the Church style of typing, which gives more priority to typing than does the second style, namely the Curry style. In the Church style, only well-typed statements are considered valid. In contrast, the Curry style conceives type checking as a validation process that weeds out ill-typed statements. In the scope of model driven environments, the Curry style of typing seems to be more appropriate, primarily due to the fact that even loosely-typed models can also be used for communication purposes.

Another advantage of isolating the compositional properties of model elements from their referential aspects is that it provides a semantics for *anonymously* addressing model elements within models. Anonymous addresses are name-agnostic. They refer to model
elements by their structural position in the containment hierarchy of a model, without being tied to the name of the containers and those of its higher-up elements. This form of addressing model elements is particularly important when handling modifications, because unlike name-sensitive addresses, such as URIs and Unix Paths, they are invariant to re-names and updates of other elements. To give an analogy, the street address of a building changes as does the name of the street in which it resides, even though the building itself is located at exactly the same geographical coordinates. Latitude and longitude, contrariwise, provide immutable positioning references. Similarly, typical addressing schemes such as URI and XPath that rely on the name of parent elements for identifying elements are volatile to renaming. By separating referential and structural aspects of models and giving prominence to the latter, we devise an addressing scheme that solely relies on the structural positions of model elements which is invariant to any non-structural changes.

More specifically, to form the anonymous address of an element we define two operators, “/” and “.”, which are defined in the following.

**Definition 3 (Container Selector)** Operator “/” is a function with the signature: “/” : $\mathcal{M} \times \mathbb{N} \rightarrow \mathcal{C}$ in which $\mathcal{M}$ is the set of all model elements, $\mathcal{C}$ is the set of all containers, $\mathbb{N}$ is the set of positive integers.

$$(\mathcal{C}, \mathcal{A}, \mathcal{R}, \mathcal{T})/i = \pi_i(\mathcal{C})$$

The “/” operator is a binary operator whose first operand is a model element and its second operand is a positive integer. $m/n$ returns the $n_{th}$ container of model element $m$. For convenience, we extend the notation of this (and the following) operators to also allow names, as well as positions, when needed.

**Definition 4 (Element Selector)** Operator “.” is a function with the signature: “.” : $\mathcal{C} \times \mathbb{N} \rightarrow \mathcal{M}$ where $\mathcal{M}$ is the set of all model elements, $\mathcal{C}$ is the set of all containers, $\mathbb{N}$ is the set of positive integers.

$$C.j = \pi_j(C)$$

**Definition 5 (Attribute Selector)** Operator “@” is a function with the signature: “@” : $\mathcal{M} \times \mathbb{N} \rightarrow \Sigma^*$ where $\mathcal{M}$ is the set of all model elements, $\mathcal{C}$ is the set of all containers, $\mathbb{N}$ is the set of positive integers.

$$(\mathcal{C}, \mathcal{A}, \mathcal{R}, \mathcal{T})@i = \pi_i(A)$$
By cascading the container/element segments formed by these operators, we can devise an anonymous addressing scheme which resemble paths, and allow us to navigate, query and access different model elements, and their properties, in a containment hierarchy. The following grammar specifies the syntax of this addressing scheme using the BNF notation.

$$
\text{Addr} ::= \text{ID} | . | ((\text{Container.Element}) | .)* \\
\text{Cont} ::= \text{ID} | \text{Number} \\
\text{Element} ::= \text{ID} | \text{Number} \\
\text{Attr} ::= \text{Addr@}(\text{ID} | \text{Number})
$$

**Definition 6 (Parent Operator)** Operator ".." is a function with the following signature ".." : $M \times M \rightarrow M$, where $M$ is the set of all model elements.

$$m..M = \begin{cases} 
p & m \in \psi p \land p \in \psi M \land \forall p' \ m \in \psi p' \Rightarrow p = p' \\
\bot & \text{otherwise} \end{cases}$$

The parent operator finds the immediate element above its given argument in a containment hierarchy. The second argument, the root of the hierarchy, is often implicit in the context and is omitted for the sake of brevity.

**Definition 7 (Address)** The (model specific) address of an element is defined as follows.

$$\text{Addr}_M(M/\alpha) = \alpha$$

For example, the address of $p_1$ in the previous example is /1.1/1.1 which reads as: element zero (i.e., $p_1$) of container zero of element zero (i.e., $m_1$) of container zero of the root element (i.e., $c_1$). This definition implies that $m = M/\text{Addr}_M(m)$.

### 3.1.1 Equivalence of two model elements

Equivalence of two model elements asserts that they must have the same types, same attribute values, same references and must also contain equivalent child elements, stationed identically in the structure of both models’ containment hierarchies. The given definition for the equivalence of two model elements is recursive, for its last criterion in Definition 8.
namely, the equivalence of container lists, requires the equivalence of their members which are again model elements; hence recursively invoking the same equivalence relation of Definition 8.

**Definition 8 (Equivalence)** *(Deep) Equivalence relation of two model elements* \( m_1 = (C_1, A_1, R_1, T_1) \) and \( m_2 = (C_2, A_2, R_2, T_2) \) *is defined as:*

\[
\begin{align*}
m_1 = m_2 \iff & \ T_1 = T_2 \\
& A_1 = A_2 \\
& R_1 = R_2 \\
& C_1 = C_2
\end{align*}
\]

As an example, consider the following model definition, alongside the definition of \( c_1 \) previously presented.

\[
c_1 = (\langle m_1 \rangle, \langle "SrvImpl" \rangle, \emptyset, JClass)
\]

\[
m_1 = (\langle p_1 \rangle, \langle "method" \rangle, \langle String \rangle, JMethod)
\]

\[
p_1 = (\emptyset, \langle "str" \rangle, \langle String \rangle, JParam)
\]

\[
c_2 = (\langle m_2 \rangle, \langle "SrvImpl" \rangle, \emptyset, JClass)
\]

\[
c_3 = (\langle m_3 \rangle, \langle "SrvImpl" \rangle, \emptyset, JClass)
\]

\[
m_2 = (\langle p_2 \rangle, \langle "method" \rangle, \langle String \rangle, JMethod)
\]

\[
p_2 = (\emptyset, \langle "str" \rangle, \langle String \rangle, JParam)
\]

\[
m_3 = (\langle p_3, p_4 \rangle, \langle "method" \rangle, \langle String \rangle, JMethod)
\]

\[
p_3 = (\emptyset, \langle "num" \rangle, \langle Integer \rangle, JParam)
\]

\[
p_4 = (\emptyset, \langle "str" \rangle, \langle String \rangle, JParam)
\]

Based on the definition of the equivalence relation of two model elements, \( m_1 = m_2 \), but \( m_1 \neq m_3 \) because of the discrepancy in their \textit{name} attributes and the extra parameter the latter possesses. Consequently, \( c_1 \neq c_3 \) due to the inequality of \( m_1 \) and \( m_3 \). The equivalence relation of two model elements compares the two elements out of the context of a containing model. To account for the containers and the elements’ positions inside them, we augment the equivalence relation with an extra condition which requires the
two model elements have the exact same structural positions in the models in which they reside: model elements \( \text{m}_1 \) and \( \text{m}_2 \) are held to be congruent with respect to models \( M_1 \) and \( M_2 \) according to the following definition.

**Definition 9 (Congruence)** *The congruence relation of two model elements \( \text{m}_1 = (\mathcal{C}_1, A_1, R_1, T_1) \) and \( \text{m}_2 = (\mathcal{C}_2, A_2, R_2, T_2) \) with respect to models \( M_1 \) and \( M_2 \) is defined as:*

\[
\text{m}_1 \overset{M_1,M_2}{\cong} \text{m}_2 \iff \begin{cases} 
\text{m}_1 = \text{m}_2 \\
\text{Addr}_{M_1}(\text{m}_1) = \text{Addr}_{M_2}(\text{m}_2)
\end{cases}
\]

In the previous example, \( p_1 = p_2 = p_4 \). However, \( p_1 \equiv p_2 \) but \( p_1 \not\equiv p_4 \), because \( \text{Addr}_{c_1}(p_1) = /1.1/1.1 \), whereas \( \text{Addr}_{c_3}(p_4) = /1.1/1.2 \).

It is also possible to compare two model elements only with respect to the number of their direct children and their attribute values and references, that is to say, to ignore discrepancies between their non-immediate descendant elements. The notion of weak equivalence is defined in the following to allow for such comparisons.

**Definition 10 (Weak Equivalence)** *The weak equivalence relation of two model elements \( \text{m}_1 = (\mathcal{C}_1, A_1, R_1, T_1) \) and \( \text{m}_2 = (\mathcal{C}_2, A_2, R_2, T_2) \) is defined as follows:*

\[
\text{m}_1 \preceq \text{m}_2 \iff \begin{cases} 
T_1 = T_2 \\
A_1 = A_2 \\
R_1 = R_2 \\
\forall i \ |\pi_i(\mathcal{C}_1)| = |\pi_i(\mathcal{C}_2)|
\end{cases}
\]

Weak equivalence is an equivalence relation inasmuch as it is manifestly reflexive, transitive and symmetric. It is primarily defined to only hold two elements to be identical, notwithstanding the differences down in the containment hierarchy or lack thereof. The primarily motivation for defining the weak equivalence relation is to confine the impact set of a change, that is, to avoid including all the elements in the containment path of a changed element up to the root of the containment hierarchy. For example, even though \( c_1 \neq c_3 \), they are weakly equivalent, since comparing only \( \text{m}1 \) with \( \text{m}3 \) notwithstanding their children (i.e., parameters) results in no difference.
3.2 Atomic Model Manipulation Operators

We adopt a compositional representation of change in models. That is to say, every change operation can be expressed as a composition of finer grained change operators. This requires identifying a number of atomic change operations as the basis of the space of possible changes. We introduce five atomic change operations that take part in change composition. The notations used for each of the atomic changes are listed as follows.

In the following function signatures, $\mathcal{M}$ is the set of all model elements, $\mathcal{T}$ is the set of all types, $\mathcal{C}$ set of all containers, $\mathcal{A}$ set of all attributes, and $\Sigma^*$ is the set of all attribute values. The purpose of unions is to overload the functions to accept both positions of elements (and containers) as well as their symbolic references in appropriate contexts.

**Definition 11 (Update)** The atomic update attribute is a function:

$$\circledast : \mathcal{M} \times (\mathcal{A} \cup \mathbb{N}) \times \Sigma^* \rightarrow \mathcal{M}$$

For any model element $m = (\mathcal{C}, \langle a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_{|\mathcal{A}|} \rangle, R, T)$

$$\circledast(m, i, v) = (\mathcal{C}, \langle a_1, \ldots, a_{i-1}, v, a_{i+1}, \ldots, a_{|\mathcal{A}|} \rangle, R, T)$$

The update function returns a model element that is identical to $m$ except for its $i_{th}$ attribute whose value is $v$.

**Definition 12 (Insertion)** The atomic element insertion is a function:

$$\circledast : \mathcal{M} \times \mathcal{T} \times (\mathcal{C} \cup \mathbb{N}) \rightarrow \mathcal{M}$$

For any model element $m = (\mathcal{C}, \mathcal{A}, R, T)$

in which $\mathcal{C} = \langle C_1, \ldots, C_{i-1}, \langle m_1, \ldots, m_{|\mathcal{C}_i|} \rangle, C_{i+1}, \ldots, C_{|\mathcal{C}|} \rangle$

$$\circledast(m, T', i) = (\mathcal{C}', \mathcal{A}, R, T)$$

where $\mathcal{C}' = \langle C_1, \ldots, C_{i-1}, \langle m_1, \ldots, m_{|\mathcal{C}_i|}, m' \rangle, C_{i+1}, \ldots, C_{|\mathcal{C}|} \rangle$ and

$$m' = (\langle \emptyset, \ldots, \emptyset \rangle, \langle \bot, \ldots, \bot \rangle, \langle \emptyset, \ldots, \emptyset \rangle, T')$$

$|\mathcal{A}_T|$, $|R_T|$ and $|\mathcal{C}_T|$ are the number of attributes, the number of reference groups and number of containers of type $T'$, respectively. The atomic element insertion function returns
a model element that is identical to \( m \) in type, attributes, references and all container but the \( i \)th one which is appended with a new raw model element of type \( T' \).

**Definition 13 (Deletion)** The atomic element deletion is a function:

\[
\nabla: \mathcal{M} \times (\mathcal{C} \cup \mathbb{N}) \rightarrow \mathcal{M}
\]

For any model element \( m = (\mathcal{C}, A, R, T) \)
in which \( \mathcal{C} = \langle C_1, \ldots, m_{|C_i|-1}, m_{|C_i|}, C_{i+1}, \ldots, C_{|\mathcal{C}|} \rangle \)

\[
\nabla(m, i) = (\langle C_1, \ldots, m_{|C_i|}, C_{i+1}, \ldots, C_{|\mathcal{C}|}, A, R, T \rangle)
\]

The atomic delete function purges the last element of the \( i \)th container of its input model element.

**Definition 14** The atomic reference insertion is a function:

\[
\Delta: \mathcal{M} \times \mathcal{D}_M \rightarrow \mathcal{M}
\]

For any model element \( m = (\mathcal{C}, A, R, T) \)
in which

\[
\Delta(m, r) = (\mathcal{C}, A, R \cup \langle r \rangle, T)
\]

The atomic reference insertion function returns a model element that is identical to \( m \) in type, attributes, containers but it has an extra reference \( r \) appended to its otherwise identical list of references (i.e., \( R \)).

**Definition 15** The atomic reference deletion is a function:

\[
\nabla: \mathcal{M} \rightarrow \mathcal{M}
\]

\[
\nabla((\mathcal{C}, A, \langle r_1, \ldots, r_n \rangle, T)) = (\mathcal{C}, A, \langle r_1, \ldots, r_{n-1} \rangle, T)
\]

The atomic reference deletion function purges the last referenced element of its input model element.

The semantics of composition is also similar to that of mathematical functions. For example, the following composite change operation adds a new parameter to the method of the Java example, and then, updates its name to “param2”:

\[
\bullet(\Delta(m1, params, JParam), name, ”param2”)
\]
The same operation can also be expressed using the anonymous scheme:

\[ \diamond (\langle 1, 1, \text{JParam}, 1, ", \text{param2}\rangle) \]

We adopt the following syntactic sugar for updating attributes: \( \diamond (\text{m}@\text{a} \mapsto \text{v}) = \diamond (\text{m}, \text{ia}, \text{v}) \), where \( \text{ia} \) is the index of attribute \( \text{a} \) taken from the type of \( \text{m} \).

### 3.3 Change Factorization and Model Edit Distance

As mentioned in Chapter 2, there are two possible perspectives toward changing models, namely, state-based and operation-based views of change. Change factorization is based on tree edit-distance and tree alignment problems. The original treatment of these problems are due to Tai \[75\]. We, present here a series of algorithms that, given two models, calculate a sequence of atomic operations that converts the first model to the second one. The returned sequence has minimum edit cost, with respect to some optimization criteria. This is similar to the edit distance of two strings, only it is applied on model elements.

**Lemma 3.3.1 (Completeness of Atomic Changes)** For any two given models \( \text{m}_1 \) and \( \text{m}_2 \) of the same type \( \text{T} \), there exists a sequence of change \( \delta^* = \delta_1 \circ \delta_2 \circ \ldots \circ \delta_n \) such that \( \delta^*(\text{m}_1) = \text{m}_2 \).

**Proof.** There is a trivial change sequence that first reduces \( \text{m}_1 \) to \( (\emptyset, \langle \perp, .., \perp \rangle, \emptyset, \text{T}) \) and from there it constructs \( \text{m}_2 \) element by element.

A factor that influences the process of change factorization is the precise semantics chosen for change operations, in particular the structural ones. If we allow insertion and deletion operations to only affect the last element of each container, we end up with a different set of operations than when we allow insertion and deletion at arbitrary locations in any container. We investigate both cases in the following.

**Change factorization using** \( \diamond (\text{m}, \text{i}, \text{v}), \triangle (\text{m}/\text{i}), \nabla (\text{m}/\text{i}) \)

The introduced insertion and deletion operations in Section 3.2, are not able to insert to and delete from an arbitrary position in a container. They are designed so that their application would preserve the structural address of the remaining elements of a model.
Another advantage of adopting these as the primitive structural change operations is that an algorithm with linear time and constant space complexity exists to factorize any two given models into a composition of these changes, as listed in Algorithm 1. The essence of this algorithm is that first tries to make the two models equivalent up to isomorphism (i.e., having no structural differences), and then apply update operations to make them have the same attribute values. This algorithm effectively associates zero cost to update and non-zero costs to insertion and deletion. In the pseudo code (listed in Algorithm 1), lines 2-6 updates the root element of $M$ to match that of $N$. The algorithm then proceeds to streamline the children of the roots of $M$ and $N$ by recursion. In lines 11 to 14 it recursively calls itself to align each existing element. If $M$ has more elements in a container than $N$ does in the corresponding container, the algorithm deletes the excess elements from $M$ (lines 21-25). Otherwise, it inserts new elements and updates them to match the extra elements in $N$ (lines 15-20).

The algorithm recursively traverses the containment hierarchy of both model elements, visiting each element only once. The runtime cost of updating an element is constant therefore the algorithm has the worst case runtime complexity of $O(\max(|M|, |N|))$. Since no memoization, other than the local variables, are required, a judicious implementation can achieve constant space complexity.

Change factorization using $\diamond (m, i, v)$, $\blacktriangle (m/i, j)$, $\blacktriangledown (m/i, j)$

Algorithm 1 can yield to quite inexorable edit sequences due to the limitation that elements can only be appended or dropped to/from containers. Although the algorithm produces minimum-cost edit scripts with respect to these given change operations, shorter edit sequences can be achieved if we lift these constraints. Assuming that model elements can be inserted to and delete from arbitrary positions of containers, we can achieve simpler and more pragmatic edit scripts.

String edit distance is a classic problem to align two arbitrary sequences of characters by applying a series of change operations comprising updating a character, inserting a new character and deleting a character to one sequence so as to make it match the other one\[^69\]. The classic solution is based on dynamic programming and has the run-time complexity of $O(m \cdot n)$, where $m$ and $n$ are the sizes the two sequences. A similar solution for model elements with multiple containers is presented in Algorithm 2. In Algorithm 2 the $\texttt{last}$ function returns the last container of a model element. $\gamma$ associates a cost to each change.
Algorithm 1 Change Factorization with append and drop

1: function factorize(M, N, ∆)
2:     for i ← 1..|A(M)| do
3:         if π_i(A^M) ≠ π_i(A^N) then
4:             ∆ ← ∆ ψ ♦(M, i, π_i(A^N))
5:         end if
6:     end for
7:     for C^M i = π_i(C^M), C^N i = π_i(C^N), i ← 1..|C^M| do
8:         j ← 0
9:         m ← |C^M_i|
10:        n ← |C^N_i|
11:        while j ≤ min(m, n) do
12:            ∆ ← ∆ ψ factorize(π_j(C^M_i), π_j(C^N_i), ∆)
13:            j ← j + 1
14:        end while
15:        if m < n then
16:            while j ≤ n do
17:                ∆ ← ∆ ψ ▲(M/i)
18:                ∆ ← ∆ ψ factorize((∅, ⊥, .., ⊥), ∅, Type(C^M_i), π_j(C^N_i))
19:                j ← j + 1
20:            end while
21:        else if m > n then
22:            while j ≤ m do
23:                ∆ ← ∆ ψ ▼(M/i)
24:                j ← j + 1
25:            end while
26:        end if
27:     end for
28: return ∆
29: end function
operation, and, \(d(s, t)\) is a table that memoizes \(\delta(s, t)\): the minimum edit distance for converting model element \(s\) to model element \(t\) using a combination of update attribute, insertion to, and deletion from an arbitrary point in the container. The following equations highlight the gist of the algorithm.

\[
\text{last}(\mathcal{C}) = \sup \{ i | \pi_i(\mathcal{C}) \neq \emptyset \land \forall j > i \ \pi_j(\mathcal{C}) = \emptyset \} \cup \{0\} \quad (1)
\]

\[
\gamma(a, b) = \begin{cases} 
0 & a = b \\
1 & \text{otherwise}
\end{cases} \quad (2)
\]

\[
\gamma(\langle a_1, ..., a_n \rangle, \langle b_1, ..., b_n \rangle) = \sum_{i=1}^{n} \gamma(a_i, b_i) \quad (3)
\]

\[
\delta((\emptyset, A), (\emptyset, B)) = \gamma(A, B) \quad (4)
\]

\[
\delta(s = \langle \mathcal{C}, A \rangle, t = (\emptyset, B)) = \gamma(\mathbb{V}) + \delta(\mathbb{V}(s/1), t) \quad (5)
\]

\[
\delta(s = (\emptyset, A), t = (\mathcal{D}, B)) = \gamma(\mathbb{A}) + \delta(s, \mathbb{V}(t/1)) + \delta(\emptyset, t/\text{last}(\mathcal{D}).|\mathcal{D}_{\text{last}(\mathcal{D})}|) \quad (6)
\]

\[
\delta(s = (\mathcal{C}, A), t = (\mathcal{D}, B)) = \\
\min \left\{ \begin{array}{c}
\delta(\mathbb{V}(s/\text{last}(\mathcal{C})), \mathbb{V}(t/\text{last}(\mathcal{D}))) + \delta(s/\text{last}(\mathcal{C}).|\mathcal{C}_{\text{last}(\mathcal{C})}|, t/\text{last}(\mathcal{D}).|\mathcal{D}_{\text{last}(\mathcal{D})}|) \\
\delta(s, \mathbb{V}(t/\text{last}(\mathcal{D}))) + \delta((\emptyset, \emptyset), t/\text{last}(\mathcal{D}).|\mathcal{D}_{\text{last}(\mathcal{D})}|) + \gamma(\mathbb{A}) \\
\delta(\mathbb{V}(s/\text{last}(\mathcal{C})), t) + \gamma(\mathbb{V})
\end{array} \right\} \quad (7)
\]

Equation 1 defines an auxiliary function, \(\text{last}\), which returns the index of the last non-empty container in a list of containers. It looks for an index after which there is either no container in the list, or all other subsequent containers are empty. For an empty list it returns 0. As mentioned, \(\gamma\) is the cost function. Equation 2 indicates that the cost of matching two attributes is zero if they are equal, or 1 otherwise. The cost function is overloaded for lists of attributes in equation 3. The cost of matching two lists is simply the sum of matching their attributes in identical positions. Equations 4-7 define the edit-distance of two models based on a given cost function. The distance of two model elements that contain no children is defined in equation 4 as the cost of matching their attributes, as defined in equation 3. Equation 5 states the distance between a model element that possibly contains some children nodes and a leaf model element: it is recursively defined as the cost of deleting one element from the source model \(s\) plus the distance between the resulting model and \(t\). Similarly, equation 6 recursively defines the opposite case by an insertion operation. Equation 7 defines the generic distance function for arbitrary model
elements: at each step of recursion, one of the three operations which minimizes the total distance is chosen. Repeated expansion of equation 7, along with the base case equations 4-6, allows for calculating the minimum edit distance of two models.

**Theorem 3.3.1** Algorithm 2 has runtime and space complexity of

\[
O\left( \sum_{k=0}^{h_{\text{max}}} \sum_{i} |s_k/i| \| t_k/i| \right)
\]

Where \( h_{\text{max}} = \max\{\text{depth}(S), \text{depth}(T)\} \) and \( s_k \) and \( t_k \) are the \( k \text{th} \) descendant elements of \( S \) and \( T \) respectively.

**Proof.** Effectively, the algorithm aligns the elements of each container with its counterpart in the changed model. Assume that each container has a special null value. Aligning each element on \( S \)'s side with this null value is tantamount to deleting that element. Similarly, aligning the null element of \( S \) with each element of \( T \) signifies the insertion of a new element to the container. Any other alignment pertains to updating the aligned element of \( S \) to match an element in \( T \). If the container \( i \) of \( S \) has \( m \) elements, and that of \( t \) has \( n \) elements, there exist \( (m+1)(n+1) \) alignments for the whole container. As explicated in the last equation above (and on line 26 of Algorithm 1), each alignment leads to at most three subproblems, each of which occupies a constant space in the memorization table and a constant time for calculating its cost. To obtain the total space and time complexity orders we, thus, have to sum this over all the containers of any two corresponding model elements in the corresponding depth level of the containment hierarchies in \( S \) and \( T \).

\[\square\]

Figures 3.2 illustrate the result of experiments that have been done on random trees using Algorithm 2. It is evident from the figure that both measured size (represented by \( |d| \)) and elapsed time (represented by \( T \)) are asymptotically bound by the equation of **Theorem 3.3.1** (represented, using constant factor 3, in the figure by \( O(d) \)).

In Algorithmalg:factorize2, the \( c(S, T) \) matrix holds the actual change selected as the optimized edit step for models \( S \) and \( T \). This is used in conjunction with \( d(S, T) \), the table for edit cost of each subproblem, to compute using simple back-tracking the actual change sequence needed to transform \( S \) to \( T \).
Algorithm 2 Change Factorization with insert and delete

1: function $\delta(S, T)$
2: if (cached ← $d((S, T)) \neq \emptyset$) then
3:   return cached
4: end if
5: if $C^S = C^T = \emptyset$ then
6:   $\Delta \leftarrow \emptyset$
7:   for $i \leftarrow 1..|A^S|$ do
8:     if $\pi_i(A^S) \neq \pi_i(A^T)$ then
9:       $\Delta = \Delta \uplus (S, i, \pi_i(A^T))$
10:   end if
11: end for
12: $d(S, T) \leftarrow |\Delta| \times \gamma(\uplus)$
13: return $d(S, T)$
14: end if
15: if $C^S = \emptyset$ then
16:   change($S, T$) $\leftarrow \triangledown$(S/last($C^S$))
17:   $d(S, T) \leftarrow \gamma(\triangledown) + \delta(S, \triangledown(T/last(C^T)), T/last(C^T), |C^T_last(C^T)|)$
18: return $d(S, T)$
19: end if
20: if $C^T = \emptyset$ then
21:   change($S, T$) $\leftarrow \triangledown(T/last(C^T))$
22:   $d(S, T) \leftarrow \gamma(\triangledown)$
23: return $d(S, T)$
24: end if
25: \[
\begin{cases}
\text{updateCost} = \delta(\triangledown(S/last(C^S)), \triangledown(T/last(C^T))) + \\
\delta(S/last(C^S), |C^S_last(C^S)|, T/last(C^T), |C^T_last(C^T)|)
\end{cases}
\]
26: $d(S, T) \leftarrow \min$
27: \[
\begin{cases}
\text{insertCost} = \delta(S, \triangledown(T/last(C^T))) + \\
\delta((\emptyset, \emptyset), t/last(C^T), |C^T_last(C^T)|) + \gamma(\triangledown)
\end{cases}
\]
28: \[
\begin{cases}
\text{deleteCost} = \delta(\triangledown(S/last(C^S)), T)
\end{cases}
\]
29: if $d(S, T) = \text{updateCost}$ then
30: change($S, T$) $\leftarrow \uplus$
31: else if $d(S, T) = \text{insertCost}$ then
32: change($S, T$) $\leftarrow \triangledown$
33: else
34: change($S, T$) $\leftarrow \triangledown$
35: end if
36: return $d(S, T)$
37: end function
Figure 3.2: Runtime and Space Complexity of Dynamic Programming Change Factorization Algorithm. X-Axis is $n$, the size of the randomly generated model, and Y-Axis measures the value of $O(d)$ as represented in Theorem 3.3.1 with a constant of 3, the actual size of the table is represented by $|d|$, the product of the size of two models is represented by $|s||t|$ for comparison. $T$ is the runtime of the algorithm which is normalized and projected for comparison with the space complexity.

### 3.3.1 Edit Script Normalization

In this section, we present a set of rules that can be used to normalize edit-sequences to equivalent ones by eliminating idempotent operations and those whose effects cancel one another. These rules should be viewed akin to algebraic rules used for symbolic simplification of algebraic expression. The presented rules are value agnostic; they can be applied before effecting the change to any model to simplify the edit script.
Proofs of these rules are straight-forward corollaries of the definitions of change operators. The first rule basically states that an update operation is cancelled by the deletion of one of its direct or indirect parents. Otherwise, the two changes are independent and their order can be interchanged. The second rule asserts that an update and an insertion are always independent and can always be interchanged. The third rule states that if two updates target the same attribute of the same element then the update that is applied latest overrides the earlier one. A corollary of this rule is that Update is idempotent. The forth rule highlights the net effect of two delete operations. If either target of the delete element is a descendant of the other operation then only the deletion of the element which is higher in the containment hierarchy is needed due to the fact that it purges the other one as well. Finally, the last two rules express the cancelation of insertion by deletion.

### 3.3.2 Impact set of a Change

Intuitively, an impact set of a change operation on a model is the set of all elements in the model that would be modified if the change is applied. Formally, we define the Impact Set for an update $\Delta$ over model $M$ as:

**Definition 16 (Impact Set)** The Impact set of change $\Delta : \mathcal{L}(MM) \rightarrow \mathcal{L}(MM)$ on model $M \in \mathcal{L}(MM)$ is defined as:
The constructive definition denoted above includes all elements inside model \( M \) whose positions in \( \Delta(M) \), the modified model, is occupied by an unequal (in the weak equivalence sense) element. The expression applies the overloaded "/" operator to refer to the model element in \( \Delta(M) \) that resides in the exact anonymous address denoted by \( \text{Addr}_M(m) \), i.e., the address of model element \( m \) in model \( M \). Impact set essentially includes the elements of the source model that are modified as a result of applying \( \Delta \). Weak equivalence is chosen over the stronger notion, because it is desired to only include the elements that are modified and not the entire upward path to the root of the containment hierarchy. The impact set for the atomic change operations are as follows.

**Lemma 3.3.2**

\[
\begin{align*}
J^\Delta_M &= \{ m \in \wp(M) | \Delta(M)/\text{Addr}_M(m) \not\approx m \} \\
\end{align*}
\]

**Proof.** Straightforward from the definitions of change operators and impact set. \(\square\)

**Lemma 3.3.3** \( \forall \Delta \in \{\Diamond, \Box\}^* \) : \( m \xrightarrow{M,N} n \Rightarrow \Delta(M)/\text{Addr}_M(m) \xrightarrow{\Delta(M), \Delta(N)} \Delta(N)/\text{Addr}_N(n) \)

**Proof.** The congruence relationship requires the equivalence of addresses in the two models, which easily follows from the premise of the lemma and Definition 7.

\[
\text{Addr}_{\Delta(M)}(\Delta(M)/\text{Addr}_M(m)) = \text{Addr}_M(m) = \text{Addr}_N(n) = \text{Addr}_{\Delta(N)}(\Delta(N)/\text{Addr}_N(n))
\]

To show the equivalence of elements \( m \) and \( n \), we use structural induction only on \( \Delta \)s that affect elements \( m \) and \( n \), i.e., \( m \in J^\Delta_M \) and \( n \in J^\Delta_N \).
• **Base case** There are two cases: \( \Delta = \diamond(\alpha) \) and \( \Delta = \blacklozenge(\alpha) \). For both cases from \( m \in J^\Delta_M \) and Lemma 3.3.2 it follows that \( \alpha = \text{Addr}_M(m) \). From the definitions of \( \diamond \) and \( \blacklozenge \) it follows that \( \Delta(M)/\alpha = \Delta(N)/\alpha \).

- **Induction Step** for \( \Delta \in \{\diamond, \blacklozenge\}^* \) we assume that the lemma holds. From the base case it immediately follows that for any \( \delta \in \{\diamond, \blacklozenge\} \) we have:

\[
\delta(M)/\text{Addr}_M(m) \overset{\delta(M), \delta(N)}{=} \delta(N)/\text{Addr}_N(n)
\]

\[\square\]

Lemma 3.3.3 is not necessarily valid for change sequences that involve deletion. The following is a counter example:

\[
M = (\langle \langle m \rangle \rangle, \Box, \Box, \Box)  \\
N = (\langle \langle m, n \rangle \rangle, \Box, \Box, \Box)  \\
\Delta = \blacklozenge(\Box/1)  \\
\Delta(M) = (\emptyset, \Box, \Box, \Box)  \\
\Delta(N) = (\langle \langle m \rangle \rangle, \Box, \Box, \Box)  \\
(\Delta(M)/1 = \bot) \neq (\Delta(N)/1 = m)
\]

**Lemma 3.3.4** \( \forall \Delta \in \{\diamond, \blacklozenge\}^* \). \( m \in \wp M \land n \in \wp N \land \text{Addr}_M(m) = \text{Addr}_N(n) \land m \simeq n \Rightarrow \Delta(M)/\text{Addr}_M(m) \simeq \Delta(N)/\text{Addr}_N(n) \)

**Proof.** Identical to the proof of Lemma 3.3.3 replacing equivalence with weak-equivalence. \[\square\]

**Theorem 3.3.2** For model \( M \) and two arbitrary compositions of atomic insertion and update changes \( \Delta_1, \Delta_2 \in \{\diamond, \blacklozenge\}^* \), \( j_M^\Delta_2 \circ j_M^\Delta_1 \subseteq j_M^\Delta_1 \cup j_M^\Delta_2 \)

**Proof**

For each \( m \in \wp M \), \( n = \Delta_1(M)/\text{Addr}_M(m) \) and \( k = \Delta_2(\Delta_1(M))/\text{Addr}_M(m) \), the following five different cases are conceivable:

1. \( m \overset{\Delta_1}{\sim} m \overset{\Delta_2}{\sim} m \)
2. \( m \xrightarrow{\Delta_1} n \xrightarrow{\Delta_2} m \land m \not\equiv n \)

3. \( m \xrightarrow{\Delta_1} n \xrightarrow{\Delta_2} n \land m \not\equiv n \)

4. \( m \xrightarrow{\Delta_1} m \xrightarrow{\Delta_2} k \land m \not\equiv k \)

5. \( m \xrightarrow{\Delta_1} n \xrightarrow{\Delta_2} k \land m \not\equiv n \land n \not\equiv k \land m \not\equiv k \)

According to Definition 16, \( m \in \mathcal{I}_{M}^{\Delta_2 \circ \Delta_1} \) only for cases 3, 4 and 5 where \( m \not\equiv \Delta_2 \circ \Delta_1(M)/\text{Addr}_M(m) \). For cases 3 and 5, \( m \in \mathcal{I}_{M}^{\Delta_1} \), as \( m \not\equiv n \), thus the theorem is evident. To complete the proof, we ought to demonstrate for case 4 that \( m \not\equiv \Delta_2(M)/\text{Addr}_M(m) \), and therefore is a member of \( \mathcal{I}_{M}^{\Delta_2} \).

We do this demonstration by structural induction over \( \Delta_2 \):

The Base Case: For \( \Delta_2 = \diamondsuit(m, i, v) \) or \( \Delta_2 = \blacktriangle(m, i) \). is already established in Lemma 3.3.2.

Induction Step: We have to show that the theorem holds in case 4 for \( \Delta'_2 = \delta \circ \Delta_2 \) where \( \delta \in \{\diamondsuit, \blacktriangle\} \). To simplify the argument, let

\[
\begin{align*}
\text{Addr}_M(m) &= \mu \\
n &= \Delta_2 \circ \Delta_1(M)/\mu \\
k &= \delta \circ \Delta_2 \circ \Delta_1(M)/\mu \\
n' &= \Delta_2(M)/\mu \\
k' &= \delta \circ \Delta_2(M)/\mu
\end{align*}
\]

The following diagram illuminates these relationships.

The induction premise is that

\( n \not\equiv m \Rightarrow n' \not\equiv m \)
We need to demonstrate the induction step, that is:

\[ k \not\equiv m \Rightarrow k' \not\equiv m \]

We prove the induction step by contradiction, that is to say, we assume that \( k' = m \). Similar to the base case, we have the following two possibilities: \( \delta = \land (\Box/\alpha @ a_i \rightarrow v) \) or \( \delta = \land (\Box/\alpha . c_i) \). From Lemma 3.3.2, we know that in either case \( I_M^\delta = \{ M/\alpha \} \), eitherway. Therefore, the only possible way for \( m \simeq k' \) is that \( \alpha = \mu \). We discuss each case for \( \delta \) individually:

**case I** \( \delta = \land (\Box/\mu @ a_i \rightarrow v) \): It follows that \( k' @ a_i = m @ a_i = v \neq m' @ a_i \). Since \( n \overset{\delta}{\Rightarrow} k \), thus \( k @ a_i = v \), too. Thus,

\[ k \not\equiv m \Rightarrow \exists a_j \neq a_i . m @ a_j \neq k @ a_j \]

For the first case we have:

\[ \begin{align*}
\exists a_j \neq a_i . m @ a_j & \neq k @ a_j \\
\therefore (\Delta_2 = \Delta_2'; \land (M/\mu @ a_j \rightarrow k @ a_j)) \\
\land (\forall v . \Delta_2'' = \Delta_2'''; \land (M/\mu @ a_j \rightarrow v)) \Rightarrow \Delta_2''' \Rightarrow v = k @ a_j)
\end{align*} \]

\[ \therefore m' @ a_j = k' @ a_j = n @ a_j = k @ a_j \]

But this contradicts \( m \simeq k' \). Similarly, we have:

\[ \exists c_j . |m/c_j| \neq |k/c_j| \]

\[ \therefore \Delta_2 = \Delta_2' ; \land (M/\mu/c_j) ; \Delta_2'' \]

\[ \therefore |m'/c_j| = |k'/c_j| \neq |m/c_j| \]

Which is a contradiction.

**case II** \( \delta = \land (\Box/\mu . c_i) \): This implies \( |k'/c_i| > |m'/c_i| \). But \( k' \simeq m \) implies \( |m/c_i| = |k'/c_i| \) which leads to contradiction, inasmuch as \( \Delta_2, \delta \in \{ \land, \land \}^* \Rightarrow |m/c_i| \leq |m'/c_i| \).
Lemma 3.3.5

\[ J_M^{k(m/i)} = \bigcup_{j=|m/i|-k+1} m/i.j \]

**Proof.** Straight forward induction using Lemma 3.3.2. □

Lemma 3.3.6 If \( \Delta \) and \( \nabla^* \) are respectively delete-free and delete-only change sequences, then

\[ J_M^{\nabla^* \circ \Delta} \subseteq J_M^{\nabla^*} \cup J_M^\Delta \]

**Proof.** Similar to the proof of Theorem 3.3.2, we ought to consider the following and show that \( m' \not\equiv m \).

\[ m \sim_{\Delta} m \sim_{\nabla^*} n \not\equiv m \]

However,

\[ m \not\equiv n \Rightarrow (n = \Delta(M)/\mu = \bot) \lor (\exists c_i (|m/c_i| \neq |n/c_i|) \land |m/c_i| > 0) \]

For the first case we have:

\[ n = \Delta(M)/\mu = \bot \]

\[ \therefore \exists \rho, c_i, j. m \in \psi M/\rho/c_i.j \land \nabla^* = \nabla^*_1 \land \nabla^*_2 \]

\[ \therefore m \sim_{\nabla^*} (\psi \rho/c_i) \nabla^* \]

And for the second case,

\[ \exists c_i (|m/c_i| \neq |n/c_i|) \land |m/c_i| > 0 \]

\[ \therefore \nabla^* = \nabla^*_1 \land \nabla^*_2 \]

\[ \therefore |m'/c_i| < |m/c_i| \]

\[ \therefore m \not\equiv m' \]
Theorem 3.3.2, Lemma 3.3.5 and Lemma 3.3.6 along with the normalization rules of Section 3.3.1 constitute an algebraic apparatus to calculate an upper-bound for the impact of a composite change on a model, thereby enabling us to perform change impact analysis in a mechanical way, i.e., without actually applying the changes. First, we use the normalization rules to move all the delete operations to the end of the change script and obtain a composite change $\nabla^* \circ \Delta$, where $\Delta$ includes no delete operations. We use Theorem 3.3.2 to calculate $I^*_M$ and Lemma 3.3.5 to calculate $J^*_M$. Finally we combine these two to postulate an upper-bound for the total composite change according to Lemma 3.3.6.

**Definition 17 (Reset Operator)** Reset attribute operator, $\text{\textbullet} \perp$, is a special operator that sets an attribute’s value to $\perp$, a reserved value that can only be produced by this operator, and Insertion. The purpose of reset operator is to analyze the intrinsic impact of an update on a model element regardless of the current values that its attributes hold. When applied on an entire model, it resets the values of all attributes.

For a complex change $\Delta$, the impact set of this change on model $M$ depends on the current values of attributes in the model as well as the atomic change operations that constitute $\Delta$. Some of these change operations, although explicitly define an update value for an attribute, do not actually alter their target attribute value, simply because the current value is the same as the one that the change operator enforces. This prohibits the inclusion of the model element to be accounted for in the impact set. The same element however, only if it assumed a different value for that attribute, would be present in the impact set. For deducing the impact of composite changes from that of their components, it is essential to isolate the effect of the current values of attributes in target model on the impact set. Therefore we define intrinsic impact of change $\Delta$ on model $M$ as the set of all elements in $M$ that $\Delta$ touches regardless of whether they render differently from their origin or not. This set would be equivalent to the real impact of $\Delta$ if all the updates affect their target elements. Therefore, this set is equivalent of the impact of $\Delta$ on the reset version of model $M$, as by definition, it is not possible for any other change to map an attribute’s value to $\perp$. Hence the intrinsic impact of $\Delta$ on model $M$ is equivalent to $J^*_{\text{\textbullet} \perp M}$. The real impact of a change is a subset of its intrinsic impact: $J^*_M \subseteq J^*_{\text{\textbullet} \perp M}$. The difference of these two sets are the elements that are mapped to their same current values.
3.3.3 Model Transformations

Transformations can be mathematically expressed as mappings between source and target domains. With an analogy to formal languages terminology, a metamodel (or a domain model) can be considered as a generative grammar for that domain. Therefore metamodel $L(MM_d)$ defines the language of the metamodel $MM_d$, i.e., the (possibly infinite) set of all models that conform to this metamodel. Endogenous transformations, the transformations whose source and target models conform to the same metamodel, as defined in [20] can be expressed as $T_{en}: L(MM_d) \rightarrow L(MM_d)$. Similarly, exogenous transformations which have different source and target domains can be formulated as $T_{ex}: L(MM_s) \rightarrow L(MM_t)$.

Changes in artifacts can be described as endogenous transformations. $\Delta_d: L(MM_d) \rightarrow L(MM_d)$ defines the transformation $\Delta_d$ over the domain $d$ and applicable to models conforming to metamodel $MM_d$. Using this notation, interdependent model synchronization becomes the problem of finding the most computationally efficient change operations $M'_t = \Delta_t(M_t)$ applicable to model $M_t : MM_t$ for change $M'_s = \Delta_s(M_s)$ applied on model $M_s : MM_s$ such that $M'_t$ and $M'_s$ remain consistent according to consistency criteria between $M_s$ and $M_t$.

Artifact generation is, generally, an example of exogenous model transformation as its source and target’s metamodels are not necessarily the same. Mapping $T_{Gen}: MM_s \rightarrow MM_t$ transforms models of domain $s$ that conform to metamodel $MM_s$ to models of domain $t$ conforming to metamodel $MM_t$.

We define consistency between two models in general as a mathematical relation defined over the Cartesian product of their metamodel languages. For example, models $M_1 : MM_1$ and $M_2 : MM_2$ are consistent with respect to consistency relationship $R \subseteq L(MM_1) \times L(MM_2)$ if $(M_1, M_2) \in R$.

3.3.4 Classification of Transformations

Model transformation can be classified into certain groups based on the behavior of the transformation with respect to the change operations applied on the source model. As described, these changes correspond to a set of changes that need to be applied on the target side and result in a consistent model.
Definition 18 (Homomorphic Transformation) Transformation \( T \) is said to be homomorphic if \( T(\Delta(M)) = \Delta'(T(M)) \Rightarrow T(\Delta \circ \delta(M)) = \Delta'(T(\delta(M))) \), that is to say, the following diagram commutes.

\[
\begin{array}{c}
M \xrightarrow{\delta} \delta(M) \xrightarrow{\Delta} \Delta(\delta(M)) \\
\downarrow T \quad \quad \quad \downarrow T \\
T(\delta(M)) \xrightarrow{\Delta'} \Delta'(T(\delta(M)))
\end{array}
\]

Definition 19 (Uniform Transformation) Transformation \( T \) is said to be uniform if \( T(\Delta(M)) = \Delta'(T(M)) \Rightarrow T(\Delta \circ \Delta(M)) = \Delta' \circ \Delta'(T(M)) \), that is to say, the following diagram commutes.

\[
\begin{array}{c}
M \xrightarrow{\Delta} M' \xrightarrow{\Delta} M'' \\
\downarrow T \quad \quad \quad \downarrow T \\
N \xrightarrow{\Delta'} N' \xrightarrow{\Delta'} N''
\end{array}
\]

Corollary 3.3.7 A homomorphic transformation is uniform.

Lemma 3.3.8 For homomorphic transformation \( T \) an update always translates to a sequence of updates, i.e., \( \blacklozenge \xrightarrow{T} \blacklozenge' \)

Proof. Suppose \( \blacklozenge(\square/\alpha, a, \nu) \xrightarrow{T} \delta \). \( M' = \blacklozenge(M/\alpha@a \rightarrow \nu) \) is a fixed-point of function \( \blacklozenge(\square/\alpha, a, \nu) \). Thus from \( T \) being a homomorphism it follows that its transformed counterpart, \( \delta(T(M)) \), is also a fixed-point for \( \delta \):

\[
\begin{array}{c}
M \xrightarrow{\blacklozenge(a\rightarrow\nu)} M' \\
\downarrow T \quad \downarrow T \\
N \xrightarrow{\delta} N'
\end{array}
\]

To have a fixed-point, \( \delta \) cannot have any \( \blacklozenge \) operations in its normalized form. Suppose \( \delta = \blacklozenge'(\square/\beta) \circ \delta' \). For \( N' \) to be a fixed-point for \( \delta \) it requires that \( N'/\beta = \emptyset \). It follows that the container pointed to by \( \beta \) should also be empty in \( N = T(M) \), as it is also equal
to \( T(\hat{(M'/\alpha \circ \alpha a \mapsto M/\alpha \circ \alpha a)}) \), i.e., \( N/\beta = \emptyset \), otherwise we would need to have for the reverse update operation: \( \hat{(\square/\alpha \circ \alpha a \mapsto M/\alpha \circ \alpha a)} \xrightarrow{T} \hat{(\square/\beta)} \circ \delta'' \), which was just proved to be impossible. Therefore, the normalized form of \( \delta \) does not comprise any deletion either. Thus \( \delta = \hat{\ast} \).

\[ \square \]

**Lemma 3.3.9** For homomorphic transformation \( T \)

\[
\hat{\square}(\alpha/\alpha) \xrightarrow{T} \hat{\square}(\beta/\beta \circ \alpha a \mapsto \nu) \Rightarrow \forall M \quad T(M)/\beta \circ \alpha a = \nu
\]

**Proof.** Suppose \( \nabla(\square/\alpha) \xrightarrow{T} \delta \), for any \( M \) we have

\[
T(M) = T(\nabla(\hat{\square}(M/\alpha))) = \delta(T(\hat{\square}(M))) = \delta(\hat{\square}(T(M)/\beta \circ \alpha a \mapsto \nu))
\]

if \( T(M)/\beta \circ \alpha a = x \neq \nu \) then we should have \( \delta = \hat{\square}(\beta/\beta \circ \alpha a \mapsto x) \), but this results in contradiction:

\[
T(\hat{\square}^2(M/\alpha)) = \hat{\square}^2(T(M)/\beta \circ \alpha a \mapsto \nu)
\]

\[
= \hat{\square}(T(M)/\beta \circ \alpha a \mapsto \nu)
\]

\[
T(\nabla(\hat{\square}^2(T(M)/\alpha))) = T(\hat{\square}(M/\alpha))
\]

\[
= \hat{\square}(T(M)/\beta \circ \alpha a \mapsto \nu)
\]

or diagrammatically:

\[
\begin{array}{c}
\hat{\square}\end{array}
\]

\[
\begin{array}{ccc}
M & \xrightarrow{T} & \hat{\square}(M) \xrightarrow{T} \hat{\square}^2(M)
\end{array}
\]

but also:

\[
T(\nabla(\hat{\square}^2(M/\alpha))) = \hat{\square}(T(\hat{\square}^2(M/\alpha))/\beta \circ \alpha a \mapsto x)
\]

\[
= \hat{\square}(T(M)/\beta \circ \alpha a \mapsto \nu)/\beta \circ \alpha a \mapsto x)
\]

\[
= T(M)
\]

\[ \square \]
Lemma 3.3.10  For homomorphic transformation $T$

\[
\Delta(\Box/\alpha) \xrightarrow{T} \nabla(\Box/\beta) \Rightarrow \forall M \ T(M)/\beta = \emptyset
\]

Proof. Suppose $M$ is a model element, then $M_0 = \nabla^{M/\alpha}(M/\alpha)$ is a fixed point for function $\nabla(\Box/\alpha)$. If $\nabla(\Box/\alpha) \xrightarrow{T} \delta$ the following set of equations holds for $N_0 = T(M_0)$ and $N_1^+ = T(M_1^+) = T(\Delta(M_0/\alpha))$.

\[
\begin{align*}
\nabla(N_0/\beta) &= N_1^+ \\
\delta(N_1^+) &= N_0 \\
\delta(N_0) &= N_0
\end{align*}
\]

The third equation is the translation of the fact that $M_0$ is a fixed-point of $\nabla(\Box/\alpha)$ to the other side of the transformation. These equations only have one solution: $N_0 = N_1^+$, $N_0/\beta = \emptyset$ and $\delta = \text{identity}$. From Lemma 3.3.8 it follows that $N_1/\beta = \emptyset$. A straightforward induction using the same argument yields that $N/\beta = \emptyset$.

\[\Box\]

Lemma 3.3.11  For homomorphic transformation $T$

\[
\nabla(\Box/\alpha) \xrightarrow{T} \Diamond(\Box/\beta @ \alpha \mapsto \nu) \Rightarrow \forall M \ T(M)/\beta @ \alpha = \nu
\]

Proof. For model $M$ let $N = T(M), M^+ = \Delta(M/\alpha), N^+ = T(M^+)$.
As the above diagram shows $N = \Phi(N^+ / \beta @ a \rightarrow v)$ thus we can infer that $N / \beta @ a = v$.

**Lemma 3.3.12** The normalized translation of a delete operation by a homomorphic transformation $T$ cannot comprise any insertion.

**Proof.** For the change $\nabla(\square / \alpha)$ consider $M_0$ to be its fixed-point, i.e., $|M_0 / \alpha| = 0$, and let $N_0 = T(M_0)$. Assume that there is an un-cancelled insert in the mapped change sequence.

$$T(\nabla(M_0 / \alpha)) = N' = \delta' \circ \triangledown(\square / \beta) \circ \delta(N_0)$$

$$\therefore \ |N' / \beta| \geq |N_0 / \beta| + 1$$

Which is a contradiction, since $N' = T(M_0) = N_0$.

**Lemma 3.3.13** For homomorphic transformation $T$

$$\nabla(\square / \alpha) \xrightarrow{T} \nabla(\square / \beta) \Rightarrow \triangledown(\square / \alpha) \xrightarrow{T} \Phi^* \circ \triangledown(\square / \beta)$$

**Proof.** From the following diagram and edit sequence simplification rules,

it is evident that $\delta = \Phi^* \circ \triangledown(N / \beta)$.
Lemma 3.3.14 (Locality) For homomorphic transformation $T$

\[
\nabla(\square/\alpha) \xrightarrow{T} \nabla(\square/\beta) \wedge (\square/\alpha.i@a \rightarrow v) \xrightarrow{T} \Box(\square/\gamma.j@a' \rightarrow v') \Rightarrow \begin{cases} 
\gamma = \beta.i/p & \text{or} \\
T(M)/\gamma.j@a' = v & \forall M 
\end{cases}
\]

For some attribute $a'$ and value $v'$.

Proof. For model $M$ first let $i = |M/\alpha|$. In the following diagram $\delta$, since $T$ is homomorphic, is an update operation.

\[
\begin{array}{c}
\text{M} \\
\downarrow T \\
\text{N} \\
\downarrow \delta \\
\text{N'} \\
\end{array} \quad \begin{array}{c}
\nabla \\
\downarrow T \\
\Box \\
\end{array}
\]

Let $\delta = \Box(\square/\gamma.j@a' \rightarrow v')$. If $N'/\gamma \not\subseteq N'/\beta.i$—that is to say, if the element at address $\beta.i$ is not a parent of the elements in container $\gamma$—then $\nabla(N'/\beta)/\gamma.j@a' = v'$, because the element at $\gamma.j$ is not eliminated by the delete operation. But we also have $N = \nabla(N'/\beta)$, therefore, $N/\gamma.j@a' = v'$. A straight forward induction on $n = 1..|M/\alpha|$ establishes the lemma for any element at $M/\alpha.(|M/\alpha| - n + 1)$.

The above Lemma is important because it tells us that for homomorphic transformations the remote impact of an update is locally bound to the elements contained by the corresponding element on the target side.

Definition 20 (Monotonic Transformation) Transformation $T$ is said to be monotonic if

\[
\begin{align*}
T(\Box(M)) &= \Box^*(T(M)) \\
T(\triangle(M)) &= \triangle^*(T(M)) \\
T(\nabla(M)) &= \nabla^*(T(M))
\end{align*}
\]
3.3.5 Traces and Dependencies across Transformations

A model element is said to be dependent on (or traced to) another model element through a transformation if making changes to the source element would require making a change to the target element. This notion is defined formally in the following.

**Definition 21 (Dependency)** Model elements $m \in \mathcal{M}$ and $n \in \mathcal{N}$ are said to be dependent through transformation $N = T(M)$ if there exist changes $\delta$ and $\delta'$ such that $T(\delta(M)) = \delta'(N) \land \mathcal{I}_N^\delta = \{m\} \land n \in \mathcal{I}_N^{\delta'}$.

The most common form of dependency between model elements on the target and source side of a transformation is when some attribute values of the element on the target side depend on some of the attribute values of the element in the source model. The following definition gives a formal account of this type of dependency between model elements.

**Definition 22 (Value Dependency)** Model elements $m \in \mathcal{M}$ and $n \in \mathcal{N}$ are value-dependent if

$$\exists a_i, a_j \ (m \circ a_i) \rightarrow (n \circ a_j)$$

The source and target models are called existentially dependent if removing the former from the source model would entail the removal of the latter from the target model.

**Definition 23 (Existential Dependency)** Model elements $m \in \mathcal{M}$ and $n \in \mathcal{N}$ are existentially dependent through transformation $T$ if

$$\forall (M, m) \rightarrow (N, n)$$

**Definition 24** A dependency relationship between two model elements is called purely existential if the two elements are existentially-dependent but they are not value-dependent.

According to Lemma 3.3.13 if two elements are existentially dependent through a homomorphic transformation then the dependency also extends to their containers. That is, insertion to the source’s container also entails insertion to the corresponding target’s container. In other words, all elements in the containers of two existentially dependent elements are existentially dependent.
3.3.6 Semantics of Synchronization for Generated Artifacts

Trivially, when the source artifact changes, the same transformation whereby the target artifact was generated from the source artifact can be reinvoked to generate a new version of the target model that is consistent with the modified source. Whether this regeneration of the target artifact is computationally efficient depends on the size of the models and the complexity of the invoked transformation. Nevertheless, the consistency relationship between the source and target models is implicitly enforced by the embedded transformation rules. Thus the problem of model synchronization for the models bound together with transformation rules such as, $M_t = T_{Gen}(M_s)$, reduces to determining $\Delta_t : MM_t \rightarrow MM_s$ for every change $\Delta_s$ defined on the source model, such that $\Delta_t(M_t) = T_{Gen}(\Delta_s(M_s))$. It should be noted that this process does not necessarily involve complete regeneration of the target model. In fact, to have an efficient model synchronizer we would like to translate the changes of the source domain to the minimum set of modifications in the target domain that change the target producing the same model that the transformation would yield if applied on the updated source model.

3.3.7 Bi-directional synchronization

Unlike compilation of programs into machine code, in MDD it cannot be assumed that the end product of a transformation will not be independently changed. Although a model generated from another model is a live software document that independently evolves, the source artifact still has to be reconciled with it accordingly. The model transformer used for generating the target model is not necessarily a bi-directional transformation, hence, not usable for backward synchronization. Although, it may not always be possible to reconcile models in both directions, a supervised synchronization scheme that is capable of propagating changes in both directions can be of tremendous practical value even if it does not provide full consistency. This problem can be stated as finding change function $\Delta_s : M_s \rightarrow M_s$ for a change $\Delta_t$ over domain $t$ such that if the transformation $T_{Gen}$ is re-executed on $M'_s = \Delta_s(M_s)$ it would result in $M'_t = \Delta_t(M_t)$. 
3.3.8 Incremental synchronization

Definition 25 (Incrementality) For two models $M_s$ and $M_t$ that are to be synchronized under consistency relationship $R$, synchronizer. $\text{Sync} : MM_s \times MM_s \times MM_t \rightarrow MM_t \times MM_t$ computes the change $\Delta_t = \text{Sync}_{s,t}(M_s, \Delta_s(M_s), M_t)$ applicable to model $M_t$. $\text{Sync}$ is called (Full)-Incremental if the following conditions hold:

\[
\begin{align*}
\Delta_s(M_s) = \emptyset & \Rightarrow \Delta_t(M_t) = \emptyset \\
\Delta_t(M_t) = \Delta_t(M_t) & \\
\forall \Delta_t' \in MM_t \times MM_t. (\Delta_s(M_s), \Delta_t'(M_t)) \in R \Rightarrow \Delta_t(M_t) \subseteq \Delta_t'(M_t)
\end{align*}
\]

Definition 26 (Partial Incrementality) $\text{Sync}$ is said to be Partially-Incremental if $\Delta_t(M_t) \subseteq \emptyset M_t$.

Definition 27 (Non-Incrementality) $\text{Sync}$ is called Non-Incremental if $\Delta_t(M_t) = \emptyset M_t$.

The above definitions utilize the notion of the impact set to give a precise definition for incremental synchronization that does not directly rely on time complexity. The essence of Definition 25 is that it considers a synchronization operation incremental, if for a given change applied to the source model of the transformation, the impact of the change sequence it produces for the target side is smaller than any other change sequence that converts the target model to the new version. In other words, it only alters elements that need to be modified. A non-incremental synchronization always touches all the elements of the target model regardless of the input change. A partially incremental synchronization is in between; its impact is a subset of the target model.
Chapter 4

Incremental Synchronization of Black-box Transformations

In this chapter, we turn our attention to the problem of model synchronization for the situations where the consistency requirements between dependent software artifacts are established by software generators, i.e., when some artifacts are generated by transforming some others using a number of transformations. In this context, there exists a definitive measure for consistency between the source and the target of the generator. If the source changes, it is possible to attain a consistent target by re-applying the generator on the new version of the source artifact. Regeneration, however, can be inefficient if it needs to be done recurrently. An incremental synchronization scheme is one that can reconcile the source with the target of the artifact by only modifying the affected elements in the target, thereby avoiding the superfluous re-computation of unaffected target elements.

The general strategy for deriving incremental synchronization for a given transformation has been to specify the transformation in a framework that supports the execution of transformations in an incremental fashion. This approach has some practical burdens, though: it requires re-implementing an existing piece of software in a new language; a notoriously challenging problem for practitioners. In this chapter, we treat existing transformations as black-boxes and try to build synchronization as an added feature that re-uses the transformation’s implementation. This saves the effort required to reverse-engineer the logic of the transformation from the existing implementation and re-implement them in another notation.
4.1 Architecture Overview

The architecture of our proposed solution is depicted in the block-diagram diagram of Figure 4.1. The crux of our idea is to sift the information for all inter-related models into small pieces, store them in one centralized place, and refer to them by a unique identifier across all heterogeneous models. To that end, we propose a process called Conceptualization. This process involves identifying, abstracting, tagging and centralizing the data encapsulated within a software artifact into logical entities called Concepts. Mutual information in related artifacts can be traced to each other through concepts; Related artifacts share mutual information that can be linked by concepts; two or more inter-related elements which reside in different artifacts may represent the same piece of information by referring to the same concept.

The overall architecture of the black-box synchronization framework is presented in Figure 4.1. Concepts are stored in a centralized location referred to as the concept pool. The framework provides facilities to efficiently trace a concept from any given position inside a model, to its corresponding entity in the concept pool and vice versa. The synchronizer unit listens for changes in the interrelated artifacts. When a change occurs, the system updates the values of concepts corresponding to the modified elements in the concept pool. The synchronization of inter-dependent models may be conducted lazily, that is, when the models need to be re-synchronized, the values in the affected concepts in the concept pool are propagated to the model elements that are indexed by the same unique identifier of the modified element. This propagation takes the form of value replacement, and can be carried out in linear time with respect to the number of elements involved. As the figure indicates, the synchronizer is, in principle, able to propagate changes in both directions, notwithstanding the subject transformation’s support for bi-directionality or lack thereof.

Automatic transformations are used in many software development environments to generate new artifacts from the existing ones. SOAP-based Web Services is one such domain that incorporates various software specifications. At the very core of a web-service lies the service implementations code, authored in a programming language such as Java. There is a service description denoted in an XML based format called Web Services Description Language (WSDL), which specifies the interface of a web-service. State of the art SOA development tools provide automatic (or semi-automatic) means for the generation of, among other artifacts, WSDL from Java. In Eclipse Web Tools Platform (WTP),
for example, WSDL can be generated from Java source code via a transformation called Java2WSDL, as depicted in Figure 4.2. When an element of the source artifact, such as the name of the method in this example, is updated, or a method is added to the source code, the target file, e.g., WSDL in this case, has to be changed accordingly. We will use this simplified version of Java2WSDL transformation and its pertaining source and target artifacts of Figure 4.2 as our running example to demonstrate the various steps of our generic incremental model synchronization methodology.

In the running example, the source and target artifacts respectively adhere to the Java grammar and the WSDL schema. To cope with diversity, all artifacts are represented in a canonical representation format; abstraction models, which are defined using a unified metamodel (e.g., MOF or EMF). Figure 4.3 depicts the JavaSrvImpl metamodel for abstracting Java implementations of web services. For brevity, this abstraction only retains the elements pertinent to the Java2WSDL transformation. Thus, the code inside method bodies is filtered out.

Figure 4.4 presents the metamodel for the Web Service Description Language. The top level element is Definition. Each definition contains a number of Service instances, which
bind to a concrete location by a Binding. Services respond to messages each represented by an instance of the Message type, which also belongs to the top-level Definition type. The concrete interface of each Message is specified by a PortType. The WSDL metamodel has some peculiarities, too. Specifically, WSDL features a type definition section which corresponds to XML schema metamodel. In other words, the elements defined in the type definition section of WSDL, ExtensibilityElements, are subclasses of XMLSchema. Figure 4.5 illustrates the abstracted models of Figure 4.2 according to these two metamodels.

4.1.1 Overall Process

When a software artifact is changed, the first synchronization step is to reconcile the rest of the elements in the same artifact with the changed ones. We refer to this step as
Intra-Model Change Propagation, for its impact is limited to the boundary of the changed artifact. The second step, Inter-Model Change Propagation, propagates the changes made to a software artifact to other inter-dependent artifacts in the system.

To have a fully synchronized model of the subject system, it is necessary to carry out both types of propagation. When a change is induced to an artifact, it should first be propagated inside the same artifact by triggering the Intra-Model Change Propagation strategies. Having made the artifact coherent within itself, the next phase is to synchronize other interrelated artifacts with the altered one via Inter-Model Change Propagation. Yet the process does not end at this step; the changes made to other artifacts can trigger additional inconsistencies between models and even to the original changed model. Our solution based on centralized concept pool does not need repetitive synchronization performed in such cases. In contrast, it is capable of resolving inconsistencies of concepts
residing in multiple models in one shot, due to the fact that these artifacts all refer to
the centralized concept pool for fetching new values. The general process for carrying out
change propagation in a multi-model environment is described in the following.
Synchronization Process

1. Check preconditions and Control Strategy for termination
2. Perform Intra Model Change Propagation for changed artifacts
3. Check Post Conditions
4. Calculate list of all affected artifacts
5. For each artifact in the impact list
   Perform Inter-Model Update Propagation
6. Verify the integrity of altered models

As we shall discuss later, multiple consistency requirements can give rise to non-terminating chains of synchronization, which should, in general, be supervised by a control strategy to break non-terminating cycles. The pre and post conditions verify that each step of the process results in well-formed artifacts.

4.1.2 Conceptualization Phase

Concepts are defined as primitive abstract entities that semantically associate two or more information carrying elements across a pool of heterogeneous software models. Models consist of model elements, which enclose several attributes to represent information. A concept, however, can be even smaller than an attribute value; attribute values can be composed of multiple concepts. Concepts, ideally, represent quanta of information, which cannot be broken down into smaller pieces. From this point of view, models provide organization, structure and semantics to an amalgamation of concepts by encapsulating them into various model elements of different types.

Furthermore, related artifacts share mutual information. Conceptualization assists the synchronization of this mutual information in two major ways. First, concepts provide a systematic way for tracing piecemeal the propagation of transformed data, inside and outside the boundaries of the artifact in which they are located. Second, concepts can establish fine-grained interdependencies between two or more inter-related artifacts; different elements in multiple artifacts can be made represent the same concept by referencing its unique identifier. Conceptualization is the process of extracting, indexing, tagging and
centralizing concepts. Concepts are stored in a database embedded in the development environment. This database is referred to as the Concept Pool.

Conceptualization can directly resolve intra-model and inter-model inconsistencies. Conceptualized artifacts have the property that their interdependent elements share mutual information through centralized concepts. Therefore, a change to one of the elements causes the related concept values be updated in the concept pool. An update can, thus, be easily propagated by pushing the values of the modified concepts to all referencing elements. Inter-model change propagation can also be embodied by utilizing the conceptualization process, granted that the models’ dependent elements are linked by referring to the same concepts in the concept pool.

Definition 28 (Concept) A concept \( c \in \mathcal{ID} \times \Sigma^* \times 2^{\text{Addr}} \) is a tuple; \( \mathcal{ID} \) is the set of all IDs, \( \Sigma^* \) is the set of all values, and \( 2^{\text{Addr}} \) is the powerset of all anonymous addresses.

We denote \( a \leftrightarrow c \in \mathcal{CP} \), to state that attribute \( a \) refers to concept \( c \) in concept pool \( \mathcal{CP} \).

Figure 4.6 illustrates the results of conceptualization for the case of the Java2WSDL example. The dependencies between the source and the target of the transformation are established using concepts, as illustrated in Figure 4.6.

Propagating changes from one model to another in this scheme takes the form of updating the pertinent concepts in the concept pool followed by fetching new values to each affected element, provided that there exists a mechanism whereby the system can pinpoint the related concepts in the concept pool for a given model element. For example, in Figure 4.6, if the method argument name “str” is changed in the Java model, the framework updates its related concept in the pool and notifies its dependent element in the WSDL side, i.e. WSDL message, to updates its “name” attribute with the new value. A modification made to the elements of the target side can likewise be propagated to the source side.

4.1.3 Shadow Phase

As mentioned earlier, it is essential to locate the concepts associated with each model element, and also respond to queries about elements sharing the same concepts. To enable
Figure 4.6: Conceptualization of Java and WSDL models
its corresponding entry in the concept pool. To address the former, we propose *Shadow Models*; they are intermediate models intended to facilitate answering to queries about elements sharing the same concepts. Shadow models closely mimic the structure of the original models. They are, in fact, produced by exchanging the values of the identified concepts in the models by their unique identifiers. Shadow models make accessible the identified concepts in the concept pool since a concept entry of an attribute value in the concept pool can be located by obtaining its concept ID from the exact same position of the attribute in the shadow model.

**Algorithm 3**

```
function Shadow(M)
    Input Model M = (C, A, R, T) and Concept-Pool CP
    Output Shadow Model S
    1: S ← Clone(M) ← Create a clone of the original model
    2: for all m ∈ S do
    3:     for all a_i ∈ A(m) do  ⇒ Replace all attributes’ values with their concept IDs
    4:         ♦(m, i, CP.getConceptID(a_i))
    5:     end for
    6: end for
    7: return S
end function
```

The algorithm for creating shadow models is listed as Algorithm 3. In the algorithm, the value of all conceptualized attributes are replaced by the identifier of their pertinent concepts. An important practical note when generating unique concept identifiers is to ensure them to be valid identifiers with respect to the grammar of both the source and the target artifacts. For the case of Java and WSDL, this means that they need to be constructed using the characters allowed in the Java grammar and the WSDL schema for class and method names and also WSDL identifiers. This requirement is to guarantee that the shadow model is a well-formed artifact of the same type of the original one, and can be used as input to transformers applicable on the original artifact. We insist that shadow artifacts be valid documents of the same type of the source model, because we use them as inputs to the transformation to generate shadow models of the target domain, thereby achieving traceability through the common concepts appearing in the shadow models of both sides of the transformation.

Figure 4.7 depicts the application of the *shadow* algorithm (Algorithm 3) on the ex-
ample input model, `srv.java.JavaSrvImpl`. The output of the `shadow` function, as shown in the figure, is structurally identical to the original model of Figure 4.5, but the values of its conceptualized attributes are replaced by the unique identifiers of their corresponding concepts. The structural reciprocity between models and their shadows enable us to easily trace each attribute to its related concept(s) in the concept pool; we only ought to look at the exact position of said attribute’s element in the shadow model to obtain its concept identifier, and, thereafter, perform a concept lookup in the concept pool.

The shadow of the target model is attained by applying the transformation on the shadow model of the source. Figure 4.8 depicts the application of the `Java2WSDL` transformation on the shadow of the source Java model, whence the shadow WSDL ensues. The reason for obtaining the target WSDL model via the application of the transformation, rather than using the `shadow` function on the target model, is to ensure both shadows use the same concept IDs to refer to interlinked elements, as is the case for the Java and WSDL shadows in Figures 4.7 and 4.8.
4.1.4 De-Shadow

The deShadow operation, listed in Algorithm 4, performs the opposite operation of the shadow algorithm. That is, it scans through the shadow artifacts, extracting the patterns for concept IDs from the attributes of model elements (an attribute can contain more than one concept by means of concatenation). For each concept ID found, it fetches its value from the concept pool, and replaces it with the value.

In this algorithm, attributes are allowed to have multiple concepts. The function match-ConceptID takes the value of an attribute in the shadow model, and matches the concept ID pattern against it. This results in a list of concept identifiers found in the attribute value. Each concept ID is then replaced by its value obtained from the concept pool.
Algorithm 4 \textit{deShadow} \textit{S}

\textbf{Input} Shadow Model \( S=(C,A,R,T) \) and Concept-Pool \( CP \)

\textbf{Output} deShadowed Model \( M \)

1: \textbf{function} \textit{deShadow}(\( S, CP \))
2: \( M \leftarrow \text{Clone}(S) \)
3: \textbf{for all} \( s \) contained in \( M \) \textbf{do}
4: \textbf{for all} \( a_i \in A(s) \) \textbf{do}
5: \( \text{id}[1..n] \leftarrow \text{matchConceptID}(a_i) \) \( \triangleright \) extract concept ID patterns in \( a_i \)
6: \( v \leftarrow a_i \)
7: \textbf{for} \( j \leftarrow 1 \) \textbf{to} \( n \) \textbf{do}
8: \( cv \leftarrow CP.\text{getConceptVal}(\text{id}[j]) \)
9: \textbf{if} \( cv \neq v \) \textbf{then}
10: \( v \leftarrow \text{replace}(v, \text{id}[j], cv) \) \( \triangleright \) replace the conceptID pattern with its concept value, fetched from the concept pool
11: \textbf{end if}
12: \textbf{end for}
13: \textbf{if} \( v \neq a_i \) \textbf{then}
14: \( \triangleright \text{Update the attribute’s value} \)
15: \( \triangleright (s, i, v) \)
16: \textbf{end if}
17: \textbf{end for}
18: \textbf{return} \( M \)
19: \textbf{end function}

4.1.5 Synchronization Process

The idea of propagating model dependencies using shadow models relies on the presumption that, under the course of the transformation, the unique concept IDs in the shadow artifact are not subject to manipulations that make them unrecognizable in the resulting target shadow model. In other words, the essence of transformations for which this methodology is applicable is re-organization of concepts. Consequently, the attribute values of model elements should only be subject to a category of re-writing operations that do not dismantle concept IDs. For example, the transformations are allowed to concatenate two concept values, or add a prefix (or a suffix) to a concept. Operations such as shuffling the characters of a concept, cutting some of the letters or anything else that does not preserve the concept IDs are not directly permitted. This, nonetheless, is not a major limitation for two reasons. On the one hand, concepts are, ideally, the most prim-
itive and finest grained pieces of information in a model. With that perspective, a wide range of meaningful transformations are expected to simply re-organize these quanta of information into different encapsulating data types, rather than tearing them apart. A transformation with such behavior is called a non-mutilating transformation (formally defined in the following). On the other hand, it is possible to work around these limitations. The general methodology to enable such anomaly cases of string re-write operations is to provide post-synchronization fix-up transformations to produce the desired effect of these rules.

**Definition 29** Monotonic transformation \( T \) is said to be non-mutilating if \( 
\diamond (\square \rightarrow \nu) \xrightarrow{T} \\
\diamond^*(\square \rightarrow \nu) \).
rocally using their mutual concepts. These concepts are stored in the central concept pool along with other concepts in the system, and each is individually and uniquely identifiable by a universal ID. When an attribute of an element in either the source or the target model the object of a change, essentially concepts that represent said attribute are updated in the concept pool. The affected concepts are also tagged as modified, and the time-stamp of the latest change is also recorded in the corresponding entry of those concepts in the concept pool. Two strategies are conceivable for propagating the values of updated concepts to the other artifacts that carry those concepts. The first strategy is to disseminate the changes to all relevant artifacts as soon as they occur. This needs maintaining a list of referencing artifacts for each concept entry in the concept pool. The second strategy is that the synchronization be carried out lazily, that is, each artifact is updated only when it is opened or it is focused on by the user. The outline of the synchronization process that is composed of three Phases, is listed below.

<table>
<thead>
<tr>
<th>Phase A: Conceptualization</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Create Models from Artifacts by abstracting to the canonical representation</td>
</tr>
<tr>
<td>2. Extract Model Concepts by conceptualization</td>
</tr>
<tr>
<td>3. Store concepts in Concept Pool</td>
</tr>
<tr>
<td>4. Create Shadow Models</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Phase B: Artifact Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5. apply the transformation on source shadows</td>
</tr>
<tr>
<td>6. deShadow the generated shadow artifact</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Phase C: Change propagation</th>
</tr>
</thead>
<tbody>
<tr>
<td>7. Upon Change in Source or Target:</td>
</tr>
<tr>
<td>8. Update related concepts in the concept pool</td>
</tr>
<tr>
<td>9. For all modified concepts</td>
</tr>
<tr>
<td>10. Get the locations of all referencing elements</td>
</tr>
<tr>
<td>11. If synchronization strategy is immediate</td>
</tr>
<tr>
<td>12. push the changes to all impacted elements enumerated in the concept pool</td>
</tr>
<tr>
<td>13. else</td>
</tr>
<tr>
<td>14. render the changes when the affected artifacts are opened or focused on</td>
</tr>
</tbody>
</table>

In the first phase of the process above, we set the stage for incremental model synchronization by abstracting the involved software artifacts to a canonical modeling notation. Then we conceptualize the resulting models, and assign unique IDs to each identified concept. The first phase concludes by creating shadow artifacts. The shadow models are denoted in the canonical modeling representation. To be able to use them with the transformation, we need to convert them back to the original artifact’s format. For example, the
shadow model created for the abstracted Java code needs to be converted to the textual code representation of a Java class so as to be readable by the Java2WSDL transformation. In the second phase of the process, we execute the artifact generations on the created shadow artifacts, thereby obtaining the shadow models for target artifacts. Subsequently, we run the deShadow algorithm to convert the target shadow to the desired target artifact. The third phase of the process is triggered whenever a change operation in one model raises the need for re-synchronization. The framework traces the modified elements to their concept entries in the concept pool, and updates the corresponding entries in the concept pool.

The synchronization operation needs to render the affected artifacts with the updated concept values. The re-invocation of deShadow for each artifact routinely achieves the desired result, for it fetches the values of all concepts in the shadow artifact from the concept pool, and replaces the concept identifiers with their values, which results in updating the modified ones. It is legitimate to dispute that this last step of the synchronization, i.e., deShadow, is not incrementally performed since it is essentially replacing all concepts, thus, its intrinsic impact is the model in its entirety. Such argument, although valid in theory, does not impair the liveness and responsiveness of the synchronization process in practice as the deShadow operation, in effect, is about the same order of complexity as simply saving and loading artifacts. Nevertheless, the framework also offers full incrementality by maintaining a list of changed shadows in the Concept pool and using the entries of referencing model elements in the concept pool as depicted in Figure 4.9, and only performing deShadow on those entries. The tradeoff, however, is making the concept pool larger and more complex.

Figure 4.9 illustrates the synchronization of the Java2WSDL example utilizing the Shadow/Transform/deShadow process. On the top of Figure 4.9, lies the Java source code, which is the input artifact of the Java2WSDL transformation. The Shadow operation encapsulates the following steps in the order given: First, the abstraction of the Java code into the canonical format (Figure 4.5). Second, the conceptualization of the resulting model. Third, performing Algorithm 3 on the abstract model to create its shadow model. Finally, serializing back the resulting shadow model into the Java code format. The result is the shadow code, which, as portrayed in the figure, is structurally identical with the original code in the segments that are relevant to the transformation, modulo the values of the concepts which are replaced by their unique identifiers. Specifically for this transformation,
the bodies of the methods are ignored by the transformation, hence no manifestation thereof in the abstract models, and consequently, neither in the shadow code.

The shadow code is used as the input to Java2WSDL, yielding the shadow WSDL. To obtain the target model, Algorithm 4 is run over the target shadow (illustrated in Figure 4.9 as the deShadow arrow). Any two related elements in either sides of the transformation, e.g., the name of the Java class and that of the WSDL portType, refer to the same entity in the concept pool and thus have the same value, because the concept IDs obtained by looking at the same locations of these elements in their shadows are identical.

It should be noted that the update synchronization aided by concepts and shadow models is two-way, even if the original transformation is unidirectional. This is one of the major benefits of the proposed approach. The synchronization engine only exploits the artifacts, their shadows and the implicit correspondences denoted in the concept pool. No further invocation of the original artifact generator is required in this process. There is no distinction between the source and target of the transformation after it is applied initially. It is therefore possible that the target of the original transformation becomes the source of model synchronization, i.e. the updates that need to be propagated are made to the model that was the product of the transformation.

A special case that deserves further attention is when an updated attribute carries more than one concept. It can be represented by concatenation of conceptualized values and prefixes (or similarly suffixes). For example an attribute can comprise a fixed prefix, the first name, a hyphen–which is another invariant segment–and the last name of a person. The first name and the last name are the conceptualized and variable segments of the attribute value, whereas the title and the hyphen are constant. When the value of such attribute is updated, the system identifies the concepts that are modified and extracts the new segmental values corresponding to those concepts. To detect the altered concepts, the attribute value is screened against its counterpart in the shadow model. The fixed segments appear in both sequences and are used for aligning concept IDs and their segmental values.
4.2 Insertion and Deletion

4.2.1 Propagation of Insertion Induced Changes

Unlike Update, Insertion and Deletion are structure altering changes. The propagation of insertion and deletion changes for an arbitrary model transformation can be multifaceted and complex. This complexity can fortunately be tamed by assuming that the transformation is homomorphic and monotonic (as defined in Chapter 3). The former is to guarantee that the transformation is not sensitive to any particular instance model in its domain, but rather it transforms them all uniformly. The latter requires the change operations to have similar effects on both sides. These two properties seem to be valid for a wide range of model transformations used in practice..

Model elements, according to Definition 1, have containers, which can be inter-dependent across multiple models. In other words, the dependency of element types can be viewed as dependency between the containers of those types. This interpretation of dependency links implicitly requires that insertion of an element to one container (only) result in addition of elements in its inter-dependent containers. We assume that Insertion (and similarly Deletion) homogeneously results in Insertion (and respectively Deletion) type of changes in other inter-related containers. We refer to this property of transformations as monotonicity. If insertion of an element results in update or deletion type of changes in the target model, then the transformation is non-monotonic.

Furthermore, we assume that introducing an element into a container follows a uniform pattern of insertions that is independent to the current state of the model (e.g., to the number of elements inside said container). For example, the addition of the third parameter to a method ought not trigger a different pattern of changes in its inter-dependent containers on the WSDL side, than does adding second parameter. We call this property of transformations continuity and such transformations are called continuous.

Although these two assumptions may seem too restraining, in practice they are in compliance with many artifact generators. In fact, the space of transformations that common relational frameworks such as QVT, TGG, Tefkat etc. are capable of expressing are also uniform and, for the most part, monotonic. For non-monotonic and/or non-uniform transformations, the explicit definitions of the transformation rules that happen to violate either of these two assumptions have to be known.
μ-Templates Overview

The proposed methodology for the propagation of Insertion, for uniform and monotonic transformations, involves deliberately injecting each container in the source shadow model with a dummy placeholder element called a μ-template. When these mutated shadow models are thereafter used as input to the artifact generation process, the μ-templates are transformed and instantiated as target artifacts. More specifically, using μ-templates, target artifacts are generated with an additional hypothetical new element into each of the source’s containers. When an actual insertion change to a container in the source model takes place, the added element replaces the available μ-template in the container; this μ-template is consumed into an actual element in the source artifact. To accommodate future insertions, a new, unsubstantiated μ-template is subsequently created.

Conceptually, in this process we a priori assume the possibility of having an additional element to be inserted in the future for each container, and reserve in advance the appropriate structure and space (i.e., the μ-template) for contingent elements in the container. Upon need, we use these reserved places for adding a new element to containers by updating the values of their attributes and rendering them as visible.

Consuming the reserved space of μ-templates, per se, prohibits adding more elements to the container in the future; simply because each container only has one extra space. Therefore, to continue supporting insertion of new elements into the container, it is necessary to create a new reserved space before consuming the available μ-template. The uniformity assumption enables us to provide a new μ-template by simply duplicating the old one and assigning new concepts to it.

Injection of μ-Template

To better demonstrate how μ-templates are utilized to propagate Insertion, utilizing μ-templates, we proceed with the example scenario of synchronizing an insertion of a method argument to the Java side with its corresponding WSDL model. The steps of the process are illustrated in Figures 4.10, 4.11, 4.12 and 4.13. The first step is injecting μ-templates into the containers of the source artifact. To disguise this amalgamation from the user and make the synchronization as transparent as possible, μ-templates are, in fact, injected into the Shadow models right after their creation. Shadow models, as discussed, closely mirror
the structure of an artifact and are kept invisible from the user.

Algorithm 5 \(\mu\)-Template Injection

**Input** Shadow Model \(S = (\mathcal{C}, A, \mathcal{R}, T)\) and Concept-Pool \(\mathcal{CP}\)

**Output** Shadow Model \(S\)

1: function \(\mu - \text{Inject}(S, \mathcal{CP})\)
2: for all \(m \in S\) do
3:   for all \(C_i \in \mathcal{C}_m\) do
4:     \(m' \leftarrow \Delta(m, C_i, T_m(C_i))\) /* Insert a new element in container \(C_i\) of element \(e\) of model \(M\)
5:       for all \(a_i \in A(m')\) do
6:         \(cid \leftarrow \mathcal{CP}.\text{addNewConcept}(\mu, \text{Addr}_S(m'), i)\)
7:     \(\diamondsuit(m', i, cid)\)
8:   end for
9: \(\mu - \text{Inject}(m', \mathcal{CP})\)
10: end for
11: return \(S\)
12: end function

The details of injecting \(\mu\)-templates into shadow models is listed in Algorithm 5. An extra element is injected in each container in the shadow model, and for each attribute of these elements a new, special concept is added to the concept pool. Figure 4.10 shows (the shadow of) an abstracted Java code for a web service on the right hand side. On the left, the same Java model is shown after it is populated by \(\mu\)-templates. As the figure illustrates using dashed lines, in every container in the model, an extra element is injected.
Figure 4.10: Injecting μ-templates
Figure 4.11: Transformation of μ-templates

Transformation of μ-Templates

As stated previously, the shadow of the source artifact is used, in lieu of the artifact, to generate the target of the transformation. The same steps are also involved for synchronizing Insertion changes, Figure 4.11 shows the shadow model of the Java abstraction populated with μ-templates, and its resulting WSDL model which includes μ-templates in several places. μ-templates are discerned from normal elements by having all of their enclosed concepts marked in the concept pool. A model element that hosts a μ-template concept is a μ-element. μ-elements do not manifest in the target model. This requires an extra step to filter μ-elements before rendering the output by deShadow.

Consumption of μ-Templates

Figure 4.12 illustrates the steps involved during the insertion of a new element, a process
we refer to as consuming a $\mu$-template. When an element is inserted in one of the containers of the source artifact, the first step is to conceptualize the new element, i.e., capturing the concepts that appear in the new model element. This is realized through consuming the $\mu$-element that is provided for the insertion of a new element in the container. Algorithm 6 presents the details of consuming $\mu$-templates. The $\mu$-element is turned into a normal element by essentially unmarking its $\mu$-template concepts in the concept pool. When consumed, $\mu$-template concepts’ IDs remain intact. The synchronization framework, when encounters the IDs of such concepts in other models, deduces that they belong to a recently consumed $\mu$-template concept. The container of the consumed $\mu$-templates need to be populated by new $\mu$-templates to allow for further insertion of model elements. Therefore a new $\mu$-template is created and injected into the container.

**Algorithm 6 $\mu$-Template Consumption**

<table>
<thead>
<tr>
<th>Input</th>
<th>Shadow Model $S = (C, A, R, T)$, Model $M$, Element $e$, Container $c$, Element Type $T$, Concept pool $CP$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Shadow Model $S$</td>
</tr>
</tbody>
</table>

1. **function** $\mu$-CONSUME($S, M, e, c, T, CP$)
2. $\mu \leftarrow \text{last}(S/\text{Addr}_M(c))$
3. $\mu' \leftarrow \mu - \text{Inject}(S/\text{Addr}_M(e), c, T(c))$  \(\triangleright\) add a new $\mu$ element for further additions
4. **for all** $a_i \in \mu$ **do**
5. $CP$.updateConcept($a_i, A(\text{last}(e).c).i$)  \(\triangleright\) Update concept’s values of $\mu$-template concepts in the pool
6. $CP$.setNext$\mu$Pointer($a_i, A(\mu').i$)  \(\triangleright\) Point to the new $\mu$-template concepts in the concept pool
7. **end for**
8. **return** $S$
9. **end function**

**Propagation of Changes**

As discussed in Subsection 4.1.1, in order to synchronize the target model with the modified source model, the deShadower is reapplied on the target shadow model. The $\mu$-template concepts are converted to normal elements in the concept pool at consumption time, i.e., when the source model was modified by an insertion. Therefore, the deShadower, when reinvoked over the target shadow model, would no longer discard these elements and
Figure 4.12: µ-template Consumption
will render them in the output model, as demonstrated in Figure 4.13. Similarly to the source model, a new $\mu$-template needs to be placed in the container to enable further insertion of elements in it. Unlike the source model, providing a new $\mu$-element to the target model involves a few more steps than simply duplicating the former $\mu$-element and adding brand new concepts to it. In particular, the dependency links between the new $\mu$-template and the one that was just inserted in the source model have to be established by making them reference the same concepts. To that end, we need to find out the concept IDs that are assigned to the newly created $\mu$-template in the source model when the old $\mu$-template was consumed. As usual, our medium for communicating such information is the concept pool. Therefore, this can be enabled by providing pointers in the entries of consumed $\mu$-template concepts in the concept pool to the new concepts created for the new $\mu$-template. For example, in Figure 4.12, when the $\mu$-template is consumed (and its value is updated to “arg2” in the concept pool) it points to the concept associated with the newly created $\mu$-template in its container. The target side is only aware of the consumed $\mu$-templates’ IDs, since the new ones were not present in the model at the time of transformation. However, the deShadow algorithm follows these pointers for each concept to reach the new $\mu$-template’s concept IDs.

Algorithm 7 enumerates the steps outlined above for propagating insertion changes using $\mu$-templates.

---

**Algorithm 7 $\mu$-Filter**

1: function $\mu$-Filter($S$, $\mathcal{CP}$)
2: \hspace{1em} $R \leftarrow \text{clone}(S)$
3: \hspace{1em} for all $s \in \psi S \land \mu - \text{elem}(s) \land \neg \mu - \text{elem}(s/.)$ do
4: \hspace{2em} if $\bigwedge_{\mu_i \in A(s)} \mathcal{CP}.\mu_i.\text{new} - \mu \neq \bot$ then
5: \hspace{3em} $s' \leftarrow \text{clone}(s)$
6: \hspace{3em} for all $a_i \in A(s)$ $s@a_i = \mu_i$ do
7: \hspace{4em} $s' \leftarrow \diamond(S/s'@a_i \mapsto \mathcal{CP}.\mu_i.\text{new} - \mu)$
8: \hspace{3em} end for
9: \hspace{1em} else
10: \hspace{2em} $S \leftarrow \nabla(R/s)$
11: \hspace{1em} end if
12: \hspace{1em} end for
13: end function
Figure 4.13: Propagating Insertion
Using \( \mu \)-templates, we have reduced the problem of propagating Insertion to an already solved problem of propagating Update.

### 4.2.2 Propagation of Deletion

When an element is deleted, all of its enclosed concepts are tagged as \textit{deleted} in the concept pool. Deletion is handled simply by hiding model elements whose concepts are tagged with the special flag \textit{deleted} in the concept pool. For example, when the parameter of a method is deleted from the source code, all the concepts of its corresponding element in its abstract model are tagged as \textit{deleted} in the concept pool. When synchronization is carried out on other models, the framework simply conceals the elements, any of whose concepts are flagged as \textit{deleted} in the concept pool.

When considering containment, propagating deletion changes raises some ontological issues. More specifically, we need to recognize the existential causality relationships between the model elements in order to properly identify the elements that have to be purged as a result of a deletion change. The semantics of containment relationship provides useful guidelines for such reasoning. Briefly, deletion of a containment results in purging all its contained elements and, consequently, their constituting concepts. Therefore, when performing inter-model change propagation, all elements whose any concepts are flagged as \textit{deleted} will likewise be flagged as \textit{deleted}.

### 4.2.3 Dependency Inference

It is possible to conceive elements in the target model of a transformation that do not contain any conceptualized information from the source model, yet their existence is caused by the existence of some other elements in the source model. Such dependency relationship is called purely existential, inasmuch as it can only be violated by structural changes, namely, Insertion and Deletion. An example of elements with this kind of relation is the \texttt{ComplexType} elements in the type definition segment of the WSDL models. For each \texttt{Method} in the Java model, the Java2WSDL transformation creates a pattern comprising a \texttt{ComplexType} node, which has no attributes. As such, it remains unaffected by any Update changes applied to the corresponding method.
However, the propagation of the Insertion changes using $\mu$-templates needs some adaptation to cope with these purely existential dependencies. As explained, $\mu$-templates are recognized—and on that basis filtered in the output target model—by the virtue of their enclosing concepts being tagged as such in the concept pool. But, an element such as $\text{ComplexType}$ does not share any concepts with its originating $\mu$-template in the source model, even though its *raison d’être* is the transformation of that element from the source model. This existential relationship posits a twofold problem for Insertion. On the one hand, these elements cannot be recognized as $\mu$-template on the target models and will incorrectly pass through $\mu$-Filter into the final target model. On the other hand, when their corresponding $\mu$-template is consumed in the source model, they are not properly duplicated along with other $\mu$-template bearing elements for future additions.

The following presents a portion of type definition in WSDL that corresponds to a $\mu$-template of a method in the source.

$$
\text{type} = (\langle\langle\text{elem1}, \text{elem2}\rangle\rangle, \emptyset, \emptyset, \text{Type}) \\
\text{elem2} = (\langle\langle\text{ct2}\rangle, \langle"\_\_\_2\_\_\_\_\rangle, \emptyset, \text{Element}\rangle) \\
\text{ct2} = (\langle\langle\text{elem3}\rangle, \emptyset, \emptyset, \text{ComplexType}\rangle) \\
\text{elem3} = (\emptyset, \langle"\_\_\_2\_\_\_\rangle, \emptyset, \text{Element})
$$

When rendering the output models, $\mu$-Filter detects $\text{elem2}$ and $\text{elem3}$ as $\mu$-templates, but is not able to do so for $\text{ct2}$ because it does not have any $\mu$-template concepts. In order for the resolve these issues, we make some additional rules as to when an element should be filtered out as a $\mu$-element. An element is considered a $\mu$-element, if:

1. It has no concepts and all of its children are $\mu$-elements.
   $$
   \forall n \in \varnothing. \quad \mu - \text{elem}(n) \land A(m) = \emptyset \quad (1)
   $$

2. It is contained by a $\mu$-template element.
   $$
   \exists p. \quad m \in \varnothing p \land \mu - \text{elem}(p)) \quad (2)
   $$

---

1 Reason for being
3. One of its attributes consists of a $\mu$-template concept.

$$\exists a \in A(m). \exists c \in \mathcal{P} \quad \mu - \text{Template}?(c) \land a \rightarrow c \quad (3)$$

These rules help deduce whether an element is a purely existential $\mu$-template by the
aid of their enclosing or enclosed $\mu$-templates. However, if the subject element lacks such
parents or children (e.g., if $ct2$ were contained by type and had no children) then the
framework is not able to detect it. Such situations have to be explicitly specified for each
transformation, and they have to be taken care of as post-synchronization fixes. In the
first case above, the purely existential $\mu$-element is consumed if any of its children are
consumed. In the second case, it is consumed when its parent is consumed and in the third
case it is consumed when all its $\mu$-templates are consumed.

A transformation is defined to be $\mu$-preserving if all elements dependent on the $\mu$-
templates injected to the source side can be detected using the three rules outline above.

4.3 Complete Picture for The Running Example

Figure 4.14 conjures up all the components of the synchronization framework we have so
far described into a workflow. On the right side of each step, the corresponding interim
artifacts for the Java2WSDL example are illustrated. This setup workflow is deployed in
lieu of the original transformation. The Abstractor and DeAbstractor are domain-specific
units that translate artifacts from their original format (e.g., Java code or XML) to the
adopted unifying model format (e.g., EMF).

As a result, the target and source artifacts are interlinked through shadow models, which
also enable the addition of new elements to both sides. In the life cycle of these artifacts,
this workflow is executed only once. Once the artifacts are in place and in adjunction with
the concept pool, the algorithm listed in Algorithm 8 carries out the synchronization of
artifacts for a given composite change.

In the synchronization algorithm, the applied change to the source model is first fac-
torized, resulting in a sequence of atomic change operations. For operation based systems,
changes are already represented as such as composite. For state based systems, change fac-
torization can be performed using Algorithm 1 or 2 or a variant thereof. This step, in fact,
Figure 4.14: Black-Box Synchronization Process
Algorithm 8 Full Synchronization Process

1: function $\text{Sync}(M, \Delta, N)$
2:    let $\text{Sh}_M =$ Shadow of $M$ and $\text{Sh}_N =$ Shadow of $N$
3:    [$\delta_1, ..., \delta_n$] $\leftarrow$ factorize($\Delta$)
4:    for $\delta_i(p), i \leftarrow 1..n$ do
5:        $s \leftarrow \text{Sh}_M/\text{Addr}_M(p)$
6:        if $\delta_i = \spadesuit(p, i, v)$ then
7:            $\text{cid} \leftarrow s@i$ $\triangleright$ obtain the concept id by looking at the same attribute in the
8:            shadow element
9:            $\mathcal{CP}.\text{updateConcept}(\text{cid}, v)$
10:           else if $\delta_i = \blacktriangle(p, i)$ then
11:              $T_i \leftarrow \pi_2(\pi_1(\overline{T(p)}))$ $\triangleright$ type of the $i$th container. Types are the second
12:                element of signature returns
13:              $\mu \leftarrow \text{Consume}(\text{Sh}_M, M, p, i, T_i, \mathcal{CP})$
14:           else if $\delta_i = \blacklozenge(p, i)$ then
15:              for all $e \in (\psi(\text{last}(s/i))) \cup \{\text{last}(s/i)\}$ do
16:                  for all $\text{cid} \in A(e)$ do
17:                      $\mathcal{CP}.\text{setDeleted}(\text{cid})$
18:              end for
19:           end if
20:    end for
21: return $\text{deShadow}(\mu - \text{Filter}(\text{deleteFilter}(\text{Sh}_N)))$
22: end function
performs differentiation for state-based environments between the two versions of altered models. More specifically, the changed elements can be detected by pair-wise traversal of models and their shadows, comparing the concept values with attribute values to detect update changes, and scanning the positions of μ-template elements in the model to detect insertions.

Given the sequence of atomic changes, the synchronization is carried out for each change operation individually. For updates, the value of altered concepts in the concept pool are updated. For insertions, the type of the container is extracted from the metamodel, which is passed to μ-Consume (Algorithm 6), along with the address of the container in which the new element is inserted. For the case of deletion, the elements are not purged from the concept pool or the shadow model; instead, all the concepts’ belonging to deleted elements are marked as deleted, which proscribes them from being rendered in the output. Finally, the last stage is to synchronize the target model. The elements with a deleted concept ID are first filtered from the shadow model. The result is passed to another filter which purges the unused μ-template elements, and as described in the previous section, also duplicate consumed μ-templates and replace their concept IDs according to the pointers set in the concept pool during μ – Consume. The filtered shadow model of the target artifact is then deShadowed to obtain the consistent target model. A DeAbstraction step is also performed in case the ultimate schema for the target model is different than our unifying model format.

4.4 Properties of the Black-box Synchronizer

4.4.1 Termination

Theorem 4.4.1 If MM and NN allow only cycle-free containers, then the black-box model synchronization Sync process always terminates for any uniform and monotonic transformation T : L(MM) → L(NN), M : MM, and an arbitrary change sequence Δ.

Proof. Algorithm 8 for a given pair of original and modified model, first factors out the change sequence (line 3) presented in Algorithm 1, which is a terminating algorithm because it is a simple recursive traversal of the containment hierarchies of the models and results in a change sequence consisting of a finite number of atomic operations. The body
of the loop between lines 4 and 14 also consist of constant-time operations (as well as the call to μ – \texttt{Consume} listed in Algorithm 6, which is evidently terminating). The loop in the segment that handles the deletion case (lines 14-18) also performs a constant-time operation on a finite number of elements (descendants of the one being deleted), hence always terminates, too. The last-line, calls the \texttt{deShadow} procedure listed in Algorithm 4, while applying two simple filters to the model. The canonical listing for \texttt{deShadow} is also a simple one-pass traversal of the target shadow model, which is guaranteed to terminate.

4.4.2 Transformation Chains

Although we have established the termination of the synchronization process for a single model transformation, in general, the termination property might not hold when several model transformations are chained together to posit a multi-lateral consistency relationship between models. Figure 4.15 outlines such a situation when three models are involved. Models M1, M2 and M3 are in an initially consistent state. This is characterized by relationships R_{1,2}, R_{2,3} and R_{3,1}. These relationships can be realized by model transformations. The homeostasis of the outer circle can, however, be disrupted by a change occurring to any of the models. In the figure, model M1 is altered by change Δ_{11}, which modifies it to M'_1. Because the modified model is no-longer consistent with M2, the synchronization process is triggered, which translates the source change Δ_{11} with respect to relationship R_{1,2} to a change applicable to M_3, viz., Δ_{21}, which converts M_2 to M'_2. Similarly, to reconcile R_{23}, the synchronization process translates Δ_{21} to Δ_{31}, which modifies M_3 to M'_3. This, however, may violate the R_{3,1} relationship, hence another change is generated by \texttt{Sync}, i.e., Δ_{12}, which migrates M'_1 to a second modified state M''_1. This chain of events spirally continues until a multi-lateral consistency state is re-established (e.g., in two rounds for the scenario depicted in Figure 4.15).

Figure 4.16 exemplifies a concrete scenario of synchronization cycles in transformation chains. A very simple model element is defined to represent variables. The first model is a variable in the \textit{under score} notation. This is transformed to what is known as the \textit{Camel Case} convention, by \texttt{Underscore2CamelCase (U2C)}, which essentially removes the underscore character and capitalizes the first letter of its succeeding part, and concatenates it with the first part. The camel case notation is then converted to so called \textit{Hungarian} notation.
by transformation \texttt{CamelCase2Hungrain} (H2C). The second transformation concatenates the first letter of the type name of the variable to the camel-case variable name after capitalizing its first letter. A third transformation, viz., \texttt{Hungarian2Underscore} (H2U) reverts the hungarian notation back to the underscore notation. This is done by removing the
first letter—which represents the type in the Hungarian notation—, making the second letter lowercase, splitting the string at the second occurrences of an upper case character, interspersing an underscore there, and, finally, concatenating with the second half whose first character is converted to lower-case. The QVT-OM implementation of these three mappings are presented in the following.

As the example of Figure 4.16 illustrates, the first round of applying the transformations does not reach a consistent state, however applying them for another round reconciles the inconsistencies. The important question to answer for these cases is whether the synchronization chain in such a situation ever terminates. There is always a possibility of
**intermediate class** Variable {
    name : String;
    type : String;
}

**mapping** Variable::Underscore2CamelCase : Variable {
    **init** {
        var usPos := self.name.indexOf(’_’);
        var firstPart := self.name.substring(1, usPos);
        var secondPart := self.name.substring(usPos + 1, self.name.length()).firstToUpper();
    }
    type := self.type;
    name := firstPart + secondPart;
}

**mapping** Variable::CamelsCase2Hungarian : Variable {
    type := self.type;
    name := type.substring(1, 1) + self.name.firstToUpper();
}

**mapping** Variable::Hungarian2Underscore : Variable {
    **init** {
        var i := 3;
        while (self.name.substring(i, i) >= ’a’
            and self.name.substring(i, i) <= ’z’) {
            i := i + 1;
        };
        var firstPart := self.name.substring(2, i - 1);
        var secondPart := self.name.substring(i, self.name.length());
    }
    type := self.type;
    name := firstPart + ’_’ + secondPart;
}
oscillating between two states, thus being trapped in a non-halting loop. A pragmatic meta-
rule imposed by a control strategy can aid in these situations. One obvious example is to
put a hard limit on the number of allowed synchronization cycles, and allow inconsistencies
to exist after attempting certain number of attempts.

It should be noted that the above example is not the kind of transformation that the
black-box model synchronization methodology readily handles, as it contains attribute
mutilating mappings. These are typically handled by suplementing the synchronizer with
ad hoc post-fix rules, and often utilizing specialized conceptulization schemes. For this
particular example, the variable name in the first model can be conceptualized into two
different concatenated concepts. The shadow value for name in the first model is thus
"cid1_cid2". The two concepts' values represented by cid1 and cid2 are "java" and "code",
respectively. This is converted to "cid1cid2" which is no longer recognizable by the de-
shadow operation. Assume that we have two fixup rules that respectively coverts this value
to "cid1cid2" and performs the required capitalization after de-shadowing. Similarly, we
will have "icid1cid2" as the name of the third variable. Updating the first model updates
cid1 from "java" to "Qvt" in the concept pool. Performing the Sync operation on each of
the two models followed by the fixups, results in the same values as shown in the first cycle
of Figure 4.16. Achieving consistency in this case requires two additonal fixup rules for
the first and second models, that is converting to lower-case the value of the first concept
after the de-shadow phase.

To avoid termination issues, a system using Sync for black-box synchronization of
multiple models adopts the general strategy of using post-fixes for reconciling remaining
descrepencies, thereby avoiding performing multiple updates to the concept-pool for a given
change. This ensures that the argument made for the termination of a single transformation
synchronization also holds valid for the multiple-transformation and cyclic cases.

4.4.3 Soundness

One constraint on Sync is that the transformation should be non-mutilating. It means
that on the target side of a non-mutilating transformation, attribute values would be
recognizable by lexicographic matching. That is to say what the transformation does to
its input model is essentially restructuring and re-organizing the information encapsulated
in the attribute values. The black-box synchronization methodology is able to capture
the logic of this restructuring without explicit knowledge of the transformation rules. The following theorem posits the soundness of this synchronization scheme for non-mutilating, uniform and monotonic transformations.

**Theorem 4.4.2 (Soundness)** For a given uniform, monotonic, non-mutilating and μ-preserving transformation $T : \mathcal{L}(MM) \rightarrow \mathcal{L}(NN)$ and a composite change $\Delta$, the black-box synchronization is sound, that is,

$$\forall M \in \mathcal{L}(MM) \quad \text{Sync}_T(M, \Delta) = T(\Delta(M))$$

**Proof** Correctness of the Sync process for update changes is evident based on the discussion in the Subsections 4.1.2, 4.1.3 and 4.1.4. We briefly re-iterate the gist of the argument here. Consider model elements $m$ and a change $\diamondsuit (m@a_i \mapsto v)$. Because the transformation is assumed to be monotonic, update changes translate to a number of updates on the target side of the transformation. Without loss of generality we can assume that the change maps to a single update change on the target. Let $n$ be an element on the target side affected by the translation of the update, that is, $\diamondsuit (n@a_j \mapsto p \lor p')$. Let $S$ and $S'$ be the shadow model in the source and the target side, respectively. The Shadow Algorithm 3 ensures that $S/\text{Addr}_M(m)@a_i = \text{cid}$. Because $T$ is non-mutilating, $S'/\text{Addr}_N(n)@a_j = p \lor \text{cid} p'$. As presented in Algorithm 8, Sync updates the corresponding concept entry—found by looking at the shadow model of the source model—in the concept pool, that is, after the change, $\mathcal{CP}.\text{getConceptVal}(\text{cid}) = v$. Thus de-shadow (Algorithm 4) updates the value of $n$ to $p \lor v \lor p$.

Due to monotonicity, $\blacktriangle(M/\alpha) \rightarrow^{i} \blacktriangle(N/\beta_i)$. Due to Lemma 3.3.14, all the μ-templates in $M/\alpha$ are localized in elements contained by $N/\beta_i$. Algorithm 4 will thus render each element $N/\beta_i$ as visible because they all contain μ-templates that are consumed. Similar argument establishes that deleted elements are correctly filtered by the Sync process.

**4.5 Complexity**

To assess the complexity of the synchronization we define the notion of change complexity, that is, the number of change operations performed on the model. This implicitly means
taxing update operations over other kinds such as simple model traversals, when estimating the runtime complexity. The complexity projected in this way however remains in acceptable correspondence with reality for most pragmatic cases. Furthermore, many of the algorithms here that have full-model traversals (e.g., deShadow in Algorithmalg:deshadow) are presented as such for the sake of clarity. As the concept-pull maintains for each concept a list of referencing model elements, it is possible to realize these algorithm in an efficient way that seeks right to the affected element, thus eliminating the need for full model traversal. The framework’s implementation in Eclipse which has been the subject of our experiments presented in the next section has used several kinds of optimizations such as the ones described.

Theorem 4.5.1 The black-box synchronizer has worst-case complexity of $O(|\Delta|)$ and space complexity of $O(|M| + |N| + |\Delta|)$.

Proof. Space overhead is basically associated with the shadow models on both sides of the transformation and the space occupied in the concept pool which is linear with respect to the size of the models. The number of changes performed by Algorithm 8 is also proportionate with the size of the factorized change sequence processed in line 4, as for each kind of change a constant number of change operations are performed.

\[\Box\]

4.6 Experiments and Evaluation

4.6.1 Experiment Setup

For the evaluation of the proposed framework, we have designed a prototype which we applied for the incremental synchronization of models in the Eclipse Web Tools Platform (WTP). WTP encompasses several types of software artifacts each having different format and schema. Furthermore, it extensively uses transformations for converting these artifacts to one another. One such transformation, which we also have used throughout this chapter for presentation, and also for conducting our assessments, is Java2WSDL. Java code is a textual file that is parsed into an Abstract Syntax Tree (AST), and is compiled into a binary class file. In contrast, WSDL is an XML document that conforms to the WSDL schema,
which is specified in the XML schema format. Moreover, because of its extensibility type
definitions, which are XML schema elements, the type definition part of WSDL conforms
to XML Schema for XML Schema (the meta-metamodel of XML).

Our experiments were conducted on a personal computer featuring Intel®Core™2 pro-
cessor and 2.00 GB of main memory, with ample disk space made available for virtual
memory utilization. Eclipse 3.4 (Ganymede) equipped with WTP 3.0—with no unneces-
sary plugin installed—was run on Microsoft®Windows™XP Service Pack 3.0. No other
major application or service was running simultaneously during the course of experiments.
Lightweight programmatic instrumentation and time measurement (console I/O essen-
tially) was used. The time spent on UI interactions needed for initiating several processes
(e.g., launching the transformation wizard) was excluded.

4.6.2 Results

Figure 4.17 compares the execution time of synchronization against that of re-transformation
of increasingly larger Java classes, i.e. with more methods, and their resulting WSDL files.
The Y (i.e., elapsed-time in seconds) axis is outlined in logarithmic scale. This graph also
shows the framework’s setup time, that is, the time spent to initialize the framework and
create the shadow models. It is evident that the time cost of synchronization is almost
independent of the models’ sizes; contrary to the time required for regeneration, which
acutely increases as the size of the input model grows. From the graphs in Figure 4.17,
we observe that a Web Service system exposing close to ten thousand methods (a quite
excessive figure for practical systems) is taking approximately 1000 seconds to regenerate
all the models using all the available transformation rules (top line), while the time to
perform the initial setup and to incrementally synchronize the models is approximately 6
seconds and 8 seconds respectively.

The Java2WSDL transformation, in fact, ranges over multiple input files. This situation
arises when one Java class references another class through methods’ argument types or
return types. In such cases, Java2WSDL also creates type definition for the referenced class
in the WSDL file. To further assess the performance of our synchronization framework, we
utilized this aspect of the transformation as an instance where the complexity of the subject
transformation also progressively increases. Figure 4.18 shows the result of synchronization
for hierarchies of multiple Java beans and their corresponding WSDL. The setup time
Figure 4.17: Performance Evaluation (Log Y-Axis)

is higher than the previous experiments, albeit still markedly faster than regeneration. This difference is predominantly due to the relatively high overhead of file I/O in Eclipse workspace, and the fact that the experiment with multiple beans naturally involves many more such operations. The results in this figure indicate that for a system composed of 512 beans the complete regeneration takes approximately 300 seconds while the setup time for the incremental synchronization process takes approximately 150 seconds. Once the setup process is complete, then synchronization due to insertion and undate induced changes takes almost constant time of less than 10 seconds. Nevertheless, the setup process takes place only once at the beginning, so, in effect, it does not slow down the synchronization phase.

The extra space required by the shadow models is reported in Figure 4.19. As the figure depicts, the space overhead of shadow models and $\mu$-templates tends to be on the same order of magnitude of the size of the models, hence it does not pose any limitations
4.7 Application and Integration

In this section we discuss the integration of the presented synchronization framework with the Eclipse Web Tools Platform (WTP) \[80\]. WTP is a collection of bedrock technologies aimed to facilitate the development and maintenance of Web applications in general, and Web Services specifically, within the Eclipse ecosystem \[24\]. WTP features a comprehensive Web Services subsystem that allows developers to develop, assemble, deploy, invoke and test Web Services in a manner congenial with the needs of large scale enterprise projects. Two primary trajectories for developing web services has been envisaged: bottom-up and top-Down development. In the former approach, users start from the implementation of
a web service, usually in Java, and apply the provided chain of transformation to create the service level artifacts, that is to say, WSDL, Web Services Deployment Descriptor (WSDD), metadata for application server configuration, and other artifacts. In contrast, in the top-down path, another chain of transformations is applied on the description of web services (i.e., WSDL) to generate stubs for Java implementation. Moreover, in either of the approaches, further software artifacts—e.g., Java code for unit testing and service invocation clients as well as Java Server Pages (JSP) proxies—can be added to the already eclectic mix of interdependent models.

WTP utilizes a third party tool from Apache Axis project [7], Java2WSDL, in order to generate WSDL files from Java code. The developer initiates the generation process by choosing the Bottom-Up Web Service Creation strategy in a WTP UI wizard and then guides it through for further customization. In addition to the time needed by the transformation, going through this elaborate, multi-step user interface for each modification to
the source code is a cumbersome task for developers. Another advantage attained from deploying live synchronization is the elimination of such redundant UI interactions.

To effectively leverage the proposed synchronization framework in an existing IDE, a number of design decisions, on the basis of the characteristics of the IDE, has to be made. To fully harness the power of the framework, the host IDE has to cater for certain requirements. In the following, we discuss the particulars of each decision point.

**Notification Mechanism**

Most IDEs provide an event driven environment to some degree. The offerings range from resource and project level events to fine-grained notifications delivered to individual model elements. Fine grained notification can be instrumental for live, on-the-fly synchronization of simultaneously open models. Eclipse Modeling Framework (EMF) provides notification mechanisms of this sort (by means of the Notifier class) which is the root of the type hierarchy in EMF. As such, all EMF objects can potentially emanate EMF Notification messages, when they are modified.

**Unifying Metamodel**

To deal with the diversity of artifacts, the proposed solution takes advantage of a unifying metamodel. As described, the synchronization process, in a step referred to as abstraction, creates from software artifacts models that are expressed in this canonical, unifying metamodel. The described synchronization algorithm are then performed on these abstract models, which are ultimately converted back to their original format. A lightweight meta-modeling scheme close to the formalism we offered in Section is preferred.

**Change Listener**

For software artifacts that are not opened in the IDE, a change listener mechanism is needed to notify the synchronization framework of the changes made to the artifacts. Eclipse IResourceChangeListener and several other classes provide this sort of facility. In the absence of this feature, the synchronization should be initiated manually by the user.

**Change Factorization**
As noted, this solution has adopted an operational view of change. This implies that for a complex change operation, the system should provide details of the constituting atomic change operations and the location of the updated elements for each atomic change. Both Eclipse (for resources) and EMF (for resources as well as in memory Objects) provide such elaborations. For state-based settings lacking this feature, the atomic changes may be calculated in two ways. The first one is the joint traversal of models with their shadows while comparing the values of each attribute with those of their representing concepts in the concept pool. The second one is comparing models with its previous version if it is retained in the system. The first approach is always plausible while the second one can, by and large, be expected to perform better since it requires fewer inquiries from the concept pool.

UI/Editor Integration

Proper integration with editors’ user interfaces enables the framework to operate speculatively, that is, it can detect the changes as they are made in the file, even before the resource is saved. The system can thus give the user interactive guides as to the valid ways of editing a model and previewing the impact of each edit before it is persisted in the file.

Concept Pool Choices

We presented the concept pool as a database of concepts. While this abstraction can be directly realized by deploying one of the available object oriented embedded databases (e.g., Apache Derby) other alternatives may be more preferable in some cases. In our implementation, we simply have used an XML file, which persists the in-memory hash maps that indexes the concepts.

Transparency

It is essential to hide the extra resources created by the framework (e.g., shadow models) from users. In Eclipse, the metadata directory is the de facto place for storing such information.

Lazy versus Eager Synchronization Strategy
The choice of synchronization strategy between eager synchronization—that is, imme-
diately pushing updates of a concept to all its referencing artifacts—and deferring the syn-
chronization to when affected artifacts are accessed, which we call lazy synchronization, is
to be made based on the characteristics of projects. For a project with numerous artifacts
the lazy synchronization strategy is preferred, inasmuch as users’ span of attention will,
in practice, only survey over a handful of resources at a given time. However, cases where
models are required to be consistent at all times are conceivable (e.g., in debug mode or
in deployed services with hot-replacement capabilities).

**Server Deployment and Runtime Issues**

In IDEs that support the deployment of generated executables remotely or in local,
embedded execution platforms (such as embedded Servlet containers like Apache Tomcat)
extra steps have to be taken to gracefully re-deploy the updated models, and/or restart
the server with modified configuration files.

**Refactoring Framework**

If the IDE offers a refactoring subsystem, it is paramount to couple it with the syn-
chronization framework so as to enable the sophisticated, and often semantics, refactoring
operations for the updates propagated by the framework. Eclipse for instance, offers the
Language ToolKit (LTK) subsystem to handle the refactoring operations in a generic fash-
ion. We coupled our prototype to this framework for enabling refactorings to happen on
non-conceptualized, and as such un-entangled, pieces of code in the projects, which still
have inter-dependencies with the conceptualized resources that the framework manages.
Chapter 5

White-Box Incrementalization of Transformations

In generative Model Driven Engineering environments, it is common to apply model transformations iteratively and frequently. As we have discussed, the repetitive application of these transformations, especially when they are complex and/or when the source models are copious, can take a significant amount of time. In the previous chapters, we introduced methodologies for the incremental synchronization of transformations by exploiting their generic properties in a black-box fashion. However, the black-box approach, albeit relatively inexpensive to employ, is inherently limited. There are transformation patterns whose synchronization requires greater amount of information about their internal structure than what can be elicited using the methods presented in Chapter 4. Such information can still be collected by opening the so called “box” and analyzing the source code of the transformations’ implementation. We refer to this methodology of deriving incremental transformations by performing static analysis on the source code of the transformation as white-box synchronization.

In this chapter, we discuss a strategy of incrementalizing model transformations based on partial evaluation—that is, pre-evaluating parts of programs using input data that are known a priori. To this end, a prototypical partial evaluator for Object Management Group’s Query, View and Transformation (QVT) Operational language is developed and used to specialize experimental QVT transformations.

Partial evaluation is an established methodology in the area of programming languages
research that is based on the premise that programs can be executed on a subset of input data known *a priori*, so as to generate a *residual* program, whose expressions are, to the extent allowed by the availability of known data, statically pre-evaluated. Thus, the residual program, when executed over the dynamic inputs, does not need to compute the parts of the code that correspond to the known inputs. Performing fewer computations at runtime, residual programs are expected to run faster than their original counterparts.

The key idea behind our approach is that model transformations are also a kind of software programs, which get as input instances of a metamodel (e.g., MOF or similarly Ecore). When a transformation is applied iteratively, the altered elements of the source model can be considered as dynamic data, and the invariant fragments as static. The catch is to reduce the number of computations performed when a complex transformation is invoked by pre-computing and storing in a residual transformation the expressions that are not affected by a model change. As a proof of concept, we have developed a prototype of a partial evaluator for a subset of OMG’s imperative model transformation language, i.e., QVT Operational Mappings. Transformations denoted in QVT, and in other model transformation languages alike, make extensive use of collection operations to manipulate the elements of input models and their containers, to form the target model. Therefore, our technique primarily focuses on the specialization of these sort of expressions. Our prototype partial evaluator is also implemented in the QVT-OM language. This design decision has two important methodological consequences. On the one hand, it adheres to the general philosophy of model driven engineering, which strives to treat all major software component as models; in particular, the object of our partial evaluator—i.e., QVT transformation—are themselves treated as MOF models, and are manipulated as such. On the other hand, this enables the concept of self-application, that is, specializing the partial evaluator by itself.

### 5.1 Introduction to Partial Evaluation

Partial evaluation of software programs refers to a pre-execution process whereby parts of the program are pre-computed with values known before runtime so as to yield a new residual (or specialized) program equivalent with the original one—in the sense that the
resulting program, when executed at runtime over full set of inputs, will produce the same output as the original program. The partial evaluation process aims to produce a residual program that ideally computes only the parts of the program that correspond to the unknown inputs. Therefore, it is expected to exhibit better overall performance than the original program.

More specifically, let

\[ \llbracket \text{prog} \rrbracket_L[\text{in}] = \text{out} \]  

(5.1)

denote that prog is a program specified in language L and produces output out for the sequence of inputs in. The \(\llbracket \rrbracket_L\) notation indicates that prog is evaluated as expressions written in language L. Suppose that the first \(m\) inputs of the program (denoted as \(\langle k_1, \ldots, k_m \rangle\)) are known a priori—that is, before the execution time—and the rest of the inputs are unknown (denoted as \(\langle u_1, \ldots, u_n \rangle\)). A partial evaluator for prog is a program such as \(PE\), inputs of which are the source code of program prog in language L, and its set of known inputs. It transforms the input program to a specialization of it, referred to as prog\(_{res}\), with respect to this set of known inputs.

Figure 5.1 depicts the general process of partial-evaluation. Partial evaluation is a form of program transformation as it produces another program as output. Let

\[ \llbracket \text{PE} \rrbracket_{L'}[\text{prog}, k_1, \ldots, k_m] = \text{prog}_{res} \]  

(5.2)

indicate that \(PE\) is interpreted as expressions of language \(L'\) and processes prog (written in \(L\)) and inputs \(\langle k_1, \ldots, k_m \rangle\). The result of this process is a residual program prog\(_{res}\) with the property that when run on the rest of the inputs it yields the same output, that is to say:

\[ \llbracket \text{prog}_{res} \rrbracket_{L''}[u_1, \ldots, u_n] = \llbracket \text{prog} \rrbracket_L[k_1, \ldots, k_n, u_1, \ldots, u_n] = \text{out} \]  

(5.3)

Note that in the general case, the target language of the residual program (i.e., \(L''\)) and the language used for the specification of PE (i.e., \(L'\)) need not be the same as the original program, viz., L. In fact, when \(L \neq L'\) the process involves a concomitant source translation phase.
$\pi E$ aims to pre-compute as many of expressions of $\text{prog}$ as possible at specialization time with respect to static data and merge the pre-computed values with the rest of the program. Because some of the expressions in $\text{prog}$ are replaced by statically pre-evaluated values in $\text{prog}_{\text{res}}$, the latter is intuitively expected to perform better than the former.

Partial evaluation usually consists of two phases. During the first phase, which is called Binding Time Analysis, the source code of the program is analyzed with respect to the set of known inputs and thereby the constructs inside the source code are annotated as either static or dynamic. Following this analysis, in the second phase the constructs that are determined to be static are evaluated, starting from the inputs and incrementally replacing each static expression with its evaluated value. Dynamic expressions in contrast are substituted with symbolic expressions that are derived from the values of static expressions and other dependent dynamic expressions. The second phase yields the residual program, which only needs the unknown subset of the inputs of the original program to run. There are two common strategies to carry out these two phases; explicitly and separately in offline evaluators, versus online evaluation by performing static analysis on the go.
along with specialization [42]. Either way, statically computable expressions of the original programs are replaced with pre-evaluated values in the residual program and thus will not be recomputed during runtime. The comparative efficiency of the residual program is thus a direct product of how aggressively can the partial evaluator determine the binding time of intermediate expressions and reduces them to correct specialized expressions.

It should be noted that partial evaluation is generally no guarantee of performance. In fact, the most naive way of partially evaluating $\text{prog}[k, u]$ with respect to $k$ is to inline the value of $k$ in a wrapper that simply invokes $\text{prog}$. Partial evaluation may also inflate the size of the program, which in certain execution contexts (e.g., for interpreted languages) may result in relatively poor performance due to the overhead incurred on the parser as a result of code bloat.

There are two particular factors involved in improving performance of residual programs. First, the more precise can the partial evaluator determine the binding time of intermediate variables based on the input division the higher the fraction of expressions which it can statically reduce. A conservative strategy would let many reduction opportunities pass unscathed by failing to recognize that they can be evaluated statically. Second, coalescing statically computed expressions with the rest of the program can be challenging and may require many ad hoc strategies whose complexity in part depends on the uniformity and sophistication of the target language. Languages such as Lisp or Scheme have relatively simple and elegant syntactic structures, and as a result lend themselves easily to static analysis processes, including partial evaluation. Some imperative languages (e.g., C++), in contrast, pose daunting obstacles for partial evaluation due to their complex (and sometimes inconsistent) syntax and semantics and multitude of unsafe features that they allow programmers to exploit.

We discuss the white-box synchronization techniques by presenting QvtMix a partial evaluator of the Object Management Group’s Query, View and Transformation language for Operational Mappings (QVT-OM). QVT-OM provides a hybrid collection of imperative and declarative constructs, and is designed to operate in one direction with no direct support for incremental execution of transformations.
5.2 Introduction to Query View Transformation Operational Mappings

Query, View and Transformation (QVT) is a specification devised and published by Object Management Group (OMG). It defines three related languages for expressing transformations between MOF compliant models. QVT has emerged out of seven initial submissions to a request for proposal by OMG in 2002 to unify the hitherto varigated methods used for expressing model transformations. QVT’s key requirements have been:

1. Compositionality of transformations

2. Support for arbitrary domain models specified in a unified meta-modeling notation such as MOF

3. Support for different levels of complexity to be able to express simple and sophisticated transformations

4. Interoperability with transformations denoted in other technologies/languages

5. Provision of a visual, model-oriented notations in the spirit of MDE

6. Support for seamless interchange of models and transformations

Figure 5.2: Query View Transformation stack of model transformation languages

QVT Relations (QVT-R) provides declarative relations for establishing correspondence between models, as well as validating such correspondences. It defines a graphical notation
(similar to TGG [31]) as well as a textual syntax for the specification of the relations. The relations are type-driven and are qualified using trigger patterns, annotations in the form of when clauses that determine the applicability of a relation to model elements. Transformations are defined as relations (in case of QVT-R) or mappings (QVT-OM) between meta-model elements and are then applied on instances of metamodels.

The core language is another declarative fragment of the QVT stack, which provides lower-level constructs for model transformations than those of the relation language. The relation language can, in fact, be implemented by mapping its constructs to those of the core language. QVT provides an interface called black-box for the invocation of external transformations. This provides a venue for extending the language and its standard libraries using general purpose programming languages and take advantage of existing libraries and frameworks.

5.2.1 Model Transformation with QVT Operational Mappings

QVT-OM is an imperative language designed to facilitate the specification of transformations between models denoted in formalisms like MOF or Ecore. The imperative features of QVT-OM offer greater expressive power than the declarative parts of the language suited for creating complex structures. In this section we aim to provide a brief overview of the language and introduce the constructs and features that are essential for understanding the remainder of this thesis.

The abstract syntax of QVT is defined as a UML metamodel using MOF notation. This makes possible treating QVT transformations as models and using QVT to define higher-order transformations. Furthermore, it enables many possibilities for easily developing various kinds of useful meta-transformations and code generators.

The basic construct of the QVT language is a module. Each module contains a transformation, which takes a number of argument. Each argument refers to a model and has a name, a type name, which corresponds to a metamodel, and a direction, which indicates if the model is input, output or both input and output (for in-place, endogenous transformations). The following is an example signature of a stereotypical transformation that maps the input metamodel MM to the output metamodel NN, which are illustrated in Figures 5.3 and 5.4, respectively.
transformation T(in M : MM, out N : NN);

Transformations consist of a number of operations, with one being a mandatory entry operation, namely, main. QVT-OM defines three different kinds of operations:

1. Query: side-effect free, pure functions answering to OCL types of queries on models
2. Helper: auxiliary, imperative functions for general purpose computation
3. Mapping: uni-directional, type-driven correspondences between input and output models of the transformation

The entry operation is typically used to access the elements of input models, sift them based on their types and other criteria, and invoke some mapping operations on them to obtain target elements. For example, the body of transformation $T$ is listed below. It accesses the top level objects of the input model using a built-in library function of QVT, namely, rootObjects, selects the first root object of type $M_1$ and assigns it to a variable named $m_1$, upon which it invokes mapping operation $M_1ToN1$. 

103
main() {
    var m1 : M := M.rootObjects()[M1]->asOrderedSet()->first();
    m1.map M1ToN1();
}

A mapping operation includes a signature, a body, optionally, a guard (when clause) and a post-condition (where clause). QVT also defines several other more complex relationships such as disjunction and inheritance between mapping operations. We deliberately ignore these features in this tutorial for the sake of brevity and also because their effect can be emulated using other more familiar language constructs. Each invocation of a mapping operation results in the instantiation of an instance of the target type of the mapping operation and its subsequent storage in the extent of the mapping operation. Model extents annotate instantiation expressions of QVT to explicitly refer to the destination model in which the instantiated object should be stored. Default extents, based on transformation parameters and element types, are assumed if no explicit model extent is give.

The body of a mapping operation consists of three separate sections: initialization, population and end section. The initialization section marked by init, which is executed prior to the instantiation of the target model element. Typically, statements in this section define and initialize auxiliary variables and perform intermediate computations that are needed for computing values of attributes of output elements. The population section, which can be optionally marked by population, is essentially assignment to the slots of the target instance implicitly instantiated before entering the population and after executing the init section. The instantiated object is referred to by keyword result. The following is an example of a mapping operation that maps instances of M to N, assigning to attribute att1 of N1 an attribute of the same name of M1. The resulting element is stored in model N as indicated by the model extent annotation.

mapping M1::M1ToN1() : N1@N {
    init {
    }
    population {
        result.att1 := self.att1;
        ...
    }
}
Mapping operators can be effectively chained for visiting of containment hierarchies of models and transforming each element type to its mapped target as the model is traversed. In our example, T maps both elements of type M3 and M4 to N2 while ignoring the intermediate containers of type M2. This behavior can readily be denoted in the population section of the M1ToN1 mapping by invoking two separate mapping operations for types M3 and M4.

```plaintext
...
result.n2 := self.m2->m3->flatten()->map M3ToN2()->asOrderedSet();
result.n2 += self.m2->m4->flatten()->map M4ToN2()->asOrderedSet();
}
}
```

QVT, like OCL, has two different notations for performing operations on collections and scalar values. Mapping scalar expressions is denoted by dots, and mapping collections is denoted by arrows (i.e., ->). In the above QVT snippet, m2 is a container of M1 elements (containing models of type M2), which is a collection in the QVT type system. The first arrow, ->m3, is technically a syntactic sugar for the ->xcollect(e.m3) expression, which results in a collection comprising elements of the m3 container for each element of self.m2, which is a collection itself. Thus, the result is a collection of collections of type M3. In our example, m1.m2->m3 evaluates to \{\{m4\},\{m5\}\}. Built-in operation flatten, as the name suggests, removes the nesting, resulting in a flat set of elements of type M3 containing \{m4, m5\}. The collection is subsequently mapped to a collection of type N and assigned to the target element containment feature by invoking the mapping operation and casting its collection type to OrderedSet. The second line performs similar operations on type M4 and merging its result with that of the first mapping invocation. The code for the two mapping operations M3ToN2 and M4ToN2 are listed below, respectively.

```plaintext
mapping M3::M3ToN2() : N2 {
  att2 := self.att3;
}

mapping M4::M4ToN2() : N2 {
  att2 := self.att4;
}
```
5.3 Syntax of Essential QVT-OM

The abstract syntax of the QVT language are specified using UML in the official OMG specification [10], and is reproduced in Appendix B. For our purpose, we adopt a subset of the language comprising its most essential features, and we opt for a traditional EBNF style for establishing textual abstract syntax necessary for the denotation of other rules later on.

5.3.1 Abstract Syntax

```
Trans ::= transformation ID (( Kind ID : Type ) *) Property * Entry Operation *
Kind ::= in | out | inout
Property ::= property ID Type Exp
Entry ::= main Block
Operation ::= Helper | Mapping
Helper ::= helper ID (( ID : Type ) *) : Type Block
Mapping ::= mapping Type ::= ID (( ID : Type ) *) : Type Init Population
Init ::= init Block
Population ::= population Block
Block ::= () | { Exp (; Exp ) *}
Exp ::= Literal | Var | Def | Assign | Literal | Type | Call | Map
     | new Type | object Type Exp
     | if Exp then Exp else Exp | while Exp do Exp
     | Exp -> forEach (Var) Block
     | Exp -> ( select | collect ) (Var|Exp)
     | Block | return Exp | break
Literal ::= true | false | 0 | 1 . . . | 'string'
     | ( OrderedSet | Set ) {Exp*}
     | Dict{ (Exp = Exp ) *}
Var ::= this | self | x | y | . . .
Call ::= Exp . ( Property | Op(Exp*) )
Map ::= Exp . map Op(Exp*)
Def ::= var Var ::= Exp
Assign ::= Exp ::= Exp
Iter ::= Exp -> ( select | collect ) (Var|Exp)
```
The top-level production in the EBNF grammar for Essential QVT-OM is the \texttt{Trans} rule, which defines a transformation. Most of the excluded features can be readily emulated using the selected ones. Therefore, there is not much loss of expressive power in Essential QVT-OM.

As it was motivated in the previous chapter, to perform effective partial evaluation on model transformations, QvtMix needs to be conscious of the semantic meanings of several built-in library functions, in order to correctly glue them with static values. Therefore, we treat a number of the standard OCL library as built-in constructs of the language and provide semantics definitions for them. The chosen library functions are presented in the following.

\begin{Verbatim}
\texttt{Op ::= =|<>|+| - |*|/|>|<|<=|>=|+|=|size | sum | at | indexOf | objectsOfType | rootObjects | deepclone | removeElement | union | intersection | isEmpty | asSet | asOrderedSet | get | put | hasKey | subOrderedSet | exists | forAll | includes | including | excluding
\end{Verbatim}

\section{5.4 Overall Process and System Architecture}

QvtMix is a partial evaluator for the model transformation language QVT-OM. It is, in the full spirit of MDE, a model transformation in its own right, developed and implemented as a QVT-OM program. QvtMix exceedingly leverages the paramount feature of QVT-OM transformations that allow for their treatment as MOF compliant models to create higher-order transformations. It is capable of evaluating and specializing a comprehensive subset of QVT-OM that incorporates its most essential features and has the full expressive power of QVT-OM (the excluded features can be simulated by the chosen one as syntactic sugars).

The partial evaluator is, in essence, a higher order transformation written in QVT; it is implemented as a visitor that uses a certain morphism strategy to evaluate static expressions and mix them with dynamic expressions in the form of residual code. The general algorithm, thus, dispatches these specific transformations for each AST node type. The AST is also represented as models compliant to the QVT metamodel depicted in
QvtMix has a hierarchically heterogeneous architecture underpinning the pipeline depicted in Figure 5.5. The Partial Evaluator pipeline accepts as input a QVT program and a set of static input data, and produces a residual program. Below, we discuss in more detail the major components of QvtMix’s architecture:

- **OCL Parser**: This component allows for the parsing and linking of OCL expressions and emits their abstract syntax. As our partial evaluator operates on Ecore elements, the generated abstract syntax is represented as EMF compliant model.

- **QVT Evaluator**: This subsystem realizes a *meta-circular* interpreter for the QVT language—that is, the interpreter is implemented as QVT program itself. The evaluator is capable of wholesome execution of QVT-OM transformations, as well as providing a function called `eval` for the reification of individual OCL and Imperative-OCL expressions. The evaluator stores the values of variables in a lexically scoped environment that is passed to each invocation of the `eval` function. The standard library of the QVT-OM specification is re-exposed (and in a few cases re-implemented). The evaluator is consulted by other components during specialization to evaluate the static values of QVT expressions.

- **Binding Time Analyzer**: QVT uses a hybrid strategy for partial-evaluation; that is, it involves an offline Binding Time Analysis (BTA) phase followed by an online binding time refinement and expression reduction. The main goal of the offline BTA is to determine a tentative binding-time for variables and expressions without evaluating them. It traverses each block of the program and propagates the input division to subsequent expressions. The BTA stores the binding-time of each variable in a binding-time environment. Possible values for binding times are `S` for definitely available values, `D` for definitely dynamic values and `M` for values whose precise binding-time cannot be determined during the offline BTA. The offline BTA is for the most part a value agnostic phase, that is, it does not take into account the actual values of inputs and other variable. It makes a judgement about their binding-times based on the binding-time of other expressions. This is among the reasons for the relatively conservative division that the offline BTA computes. For example, a side-effect free `if` expression both of whose else branch is a static value can be tagged as `S` if the
Figure 5.5: System Architecture Diagram
value of its condition expression is statically available (and it is false). Nonetheless, the offline BTA strategy marks it as $M$, for it does not have access to the actual value of the condition expression. To adapt the general partial evaluation process for the case of model transformations, we partition the input model elements into fixed and variable, by annotating their corresponding elements in the input metamodel with a change specification. More specifically the change specification designates for each class, container and attribute the possibility of modification (by tagging them as $VAR$) or lack thereof (by tagging them as $FIXED$). The BTA propagates the $VAR$ and $FIXED$ tags through the expressions using a special set of inference rules that translates the $VAR$ and $FIXED$ annotations to $S$, $D$ and $M$ divisions. The specific details of applying partial evaluation on model transformations and models are discussed in more detail in the following subsection.

• **Specializer**: This is the core component of QvtMix. It essentially is a hybrid interpreter/compiler, in the sense that it evaluates the expressions of the program as well as transforms its abstract syntax tree by a number of production rules driven based on the resulting static values. The transformation produces an AST that represents the partially evaluated program. The specializer uses the result of the offline BTA to pinpoint the possibilities of static evaluation. It, however, performs a second online binding-time analysis as it traverses the AST nodes to make final judgement about those expressions whose binding-times could not be resolved during the offline BTA phase. The specializer operates on top of the evaluator and maintains an lexically scope evaluation environment, just like an ordinary interpreter. As such, it can utilize the actual values of static inputs, variables and expressions to recognize more expressions as static, thereby providing opportunities for more aggressive specialization. The result of the online BTA is either $S$ or $D$, implying that a final judgement for expressions with uncertain binding-time (i.e., $M$) is due at end of this phase.

Based on the result of online BTA for each expression, one of the three following strategies are pursued:

1. **Reducer** for static expressions.
2. **Mixer** for dynamic expressions in fixed contexts.
3. **Polyvariant Mixer** for dynamic expressions in variable contexts.
The *reduce* strategy evaluates the result of a static expression, evaluates its side-effects as residual expressions, and returns a list of expressions to replace the value of the expression in the AST and also replicate its side-effects if there is any. On the contrary, the *mix* and *polyvariant mix* strategy is applied on dynamic expressions. These two strategies try to reduce the static subexpressions and *mix* them with the dynamic part. This often involves various *ad hoc* strategies that depend on, among other things, the AST type of the expression, the static type of the values and the values of static subexpressions.

The two types of mixing strategies pertain to the two different kind of contexts that an expression can reside in. The plain (or mono variant) mix is for dynamic expressions that have no context, or have a single, fixed context (e.g., expressions inside the `main` function). Polyvariant mix is, on the other hand, used for dynamic expressions which can be executed in multiple contexts (e.g., inside a body of a mapping operation, which can be invoked on various instances).

- **Pretty Printer**: This component accepts the abstract syntax tree of a QVT-OM transformation and produces source code text executable by the QVT-OM engine.

### 5.4.1 Partial Evaluation of Model Transformations

The quintessential form of partial evaluation as introduced in the previous section requires the subject program to take multiple inputs. A program with a singular input cannot be readily specialized using the scheme laid out in Figure 5.1. The archetypal interface of model transformations appears, however, to be one such program; it takes a single model as input and produce another model as output. This aggregate view is misleading. The truth is models are eclectic collections of multitude of elements, which can be queried and accessed in variety of ways. Therefore, the facade of holism of the model transformation interface does in fact lend itself to partial evaluation. The more important issue to solve is the mechanism to devise a binding-time division, whereby raising the arty of the transformation program.

For the case of model synchronization, there exists a natural division based on the concept of model change in the general synchronization scheme introduced in Chapter 1. Figure 5.6 illustrates how the partial evaluation process is streamlined with respect to this
change-driven division. The objective of the process is to provide an equivalent transformation for the given one, that produces the same result on the changed model. Thus, the input model $M$ can, in its entirety, considered as static input. The change specification $\Delta$ then constitutes the dynamic part of the model.

To provide better guidelines for the specialization process, the input metamodel are annotated by VAR and FIXED annotations. Along with the transformation, the framework also loads the input and output metamodels, and the change annotation information—that is, the annotations that tell the user which elements of the source models are fixed, and which ones are variable. The BTA framework accepts user annotations of the input metamodel to guide the partial evaluator about the possible classes of changes and locality thereof, according to which the AST of the input transformation is specialized. In the implementation, we use the ‘eAnnotation’ fragment of Ecore to augment the meta-models with the FIXED and VAR meta-attributes, which are accessible to the QVT-OM partial evaluator.

This explicit annotation is by no means a serious limitation. On the one hand, it is
always possible to provide multiple specialization for the set of applicable changes. This has to do with the fact that the specialization process is performed for each class of change on metamodel types (and not each specific change operation on the corresponding model instance). On the other hand, we believe that the implication of this approach for IDEs utilizing model synchronizations based on this technique is congruent with the practice of software development. Just like when programming, the developers’ interaction with models adheres to some pattern of locality and temporarily, that is, the most immediate changes are most likely to occur in the most recently modified part of the model and are likely to be of the same type of the most recent modification. In programming, a developer is most likely to continue modifying a class after its inception, by adding several methods to it for example. Likewise, the model developer will likely to add operations to a UML class in a sequential manner. Therefore, the kind of guidance required can be semi-automatically inferred from the user’s behavior and the strategy adopted for specialization can be carried out in a tractable fashion.

To facilitate model transformations, QVT-OM and OCL offer various language features to easily manipulate collections in the forms of library functions as well as ingrained language features far beyond what exists in typical general purpose imperative programming languages. These features are widely used for model transformations and provide opportunities for optimization with respect to partially static structures. In particular the general theme for specializing operations that operate on collections is to treat dynamic collections as partially static structures. For the case of ▲ for example, these collections’ available elements at specialization time are all considered as the known part of a partially static structure. Those that will be added in the future can be considered to be appended to the collection. The partial evaluator can memoize the values of the result of the operation on the static part of the structure and merge it with the dynamic part. The static caching step involves encoding the values of the expression table created in the previous step in the appropriate points in the AST. This is typically in the form of local variables embedded in each context. More specifically, for each variable (or interim value) in a local dictionary variable corresponding to its statically evaluated result is created and populated with the static result of the expressions. Embedding these values sometimes requires delicate manipulation of the AST, to, among other things, ensure the correctness and type safety of the resulting expressions. Also, the expressions need to be transformed into a new form that operate only on the modified parts of the input. This is generally done by utilizing the
information stored in the static cache. The exact details of these operations are dependent on the type of the expression and are discussed formally in Section 5.5.

5.5 Partial Evaluation

QVT-OM has an elaborate syntax posing certain challenges for the task of partial evaluation. To motivate various strategies developed for QvtMix, we present in this section a series of example specialization scenarios that progressively advance towards a typical full-blown model transformation. After each set of examples, we present a formalized specification of the reduction strategy involved in the production of the examples. In each subsection we characterize the steps and intricacies involved in deriving the residual program. We use the following definitions throughout the reset of this chapter.

\[
\begin{align*}
\text{BT} & := S \mid D \mid M \\
\text{Annot} & := \text{VAR} \mid \text{VAR}^\bullet \mid \text{VAR}^\circ \mid \text{VAR}^\# \mid \text{FIXED} \\
\tau & : \text{Exp} \rightarrow \text{BT} \\
\alpha & : \text{Exp} \rightarrow \text{Annot} \\
\Gamma & : \text{Exp} \rightarrow \text{Type} \\
\sigma & : \text{Var} \rightarrow \text{Val}
\end{align*}
\]

In the set of definitions above, BT is the domain of values for binding time of expressions. Static values are designated with S, dynamic ones with D and values binding time of which cannot be definitely determined during the offline phase are designated with M. Annot describes the domain of values for annotating metamodel elements to give guidelines about the possible ways in which they can change. VAR is the general annotation for variable elements, which has the three specialized form for each kind of change. Invariant elements are designated with FIXED. Offline BTA is symbolized by binding-time environment \(\tau\), which is basically a function from expressions to binding-time values. Similarly, \(\alpha\) signifies the annotations of metamodel elements. Expression types are accessed through typing environment \(\Gamma\). Variable store \(\sigma\) is the evaluation environment that stores the state of the program, and is maintained by the evaluator.

The following operations are similar to boolean disjunctive and conjunctive combinators that aid us to express the binding time inference equations in a concise manner.
5.5.1 Offline Binding Time Analysis Rules

The following set of rules specify the most important inference rules used for deducting the binding-time of each expression during the offline BTA phase. Each deduction returns the binding-time of the expression as well as the new BTA. The first definition is a notation simplification that allows us to omit explicitly mentioning \( \tau \) on the right hand side of the deductions whenever it remains unchanged.

\[
\begin{align*}
  t_1 \cap t_2 &= t_2 \cap t_1 \quad D \cap D = D \\
  t_1 \cup t_2 &= t_2 \cup t_1 \quad D \cup D = D \\
  M \cap D &= D \\
  S \cap D &= D \\
  S \cap M &= M \\
  S \cap S &= S \\
  M \cup D &= M \\
  S \cup D &= S \\
  S \cup M &= S \\
  S \cup S &= S
\end{align*}
\]
The first rule, BTA-Seq uses the aforementioned combinator \( \cap \) to infer the binding-time of a sequence of two statements, which is used repetitively to infer the binding time of blocks of code, e.g., bodies of loop expressions. Constant values obviously have static binding-time, as indicated by rule BTA-Lit. The binding-time of assignment and variable
definition expressions are the same as that of their right hand-side expressions (rules BTA-Init and BTA-Var). These two rules modify the BT A environment $\tau$ by adding a mapping from the variable name to its associated binding-time. If the binding-time of the condition of an If expression is dynamic, then the whole expression is recognized as dynamic (BTA-If-D). However, if the condition is static, the binding-time of the If expression is inferred to be the conjunction of its then and else expressions’ binding-time. Rules BTA-Model and BTA-Container link the binding-time value to the annotation values; the former states that the binding-time of a model element $m$ which is an instance of meta-model type $M$ is dynamic, if $M$ is annotated as Var, whereas the latter assigns $m$ to $D$ if its container has VAR annotation. Finally, the binding-time of call expressions depend on that of the source of the call and each argument of the call, as indicated by the BTA-Call rule.
5.5.2 Online Binding Time Analysis Rules

\[ \mathcal{B}[] : \text{Exp} \times \text{Env} \times (\text{Exp} \rightarrow \text{BT}) \rightarrow \{S, D\} \]

\[ \mathcal{B}[\text{Exp}] = \begin{cases} S & \tau \vdash e : S \\ D & \text{otherwise} \end{cases} \]

\[ \mathcal{B}[\text{Var}] = \begin{cases} D \sqcup \tau[x] & \sigma(x) = \bot \\ S \sqcup \tau[x] & \text{otherwise} \end{cases} \]

\[ \mathcal{B}[\text{var} \ x := e] = \mathcal{B}[e] \]

\[ \mathcal{B}[\text{if} \ e_{\text{cond}} \ \text{then} \ e_{\text{then}} \ \text{else} \ e_{\text{else}}] = \]

\[ \begin{cases} \mathcal{B}[e_{\text{then}}] \ \varepsilon[e_{\text{cond}}] = \text{True} \\ \mathcal{B}[e_{\text{else}}] \ \varepsilon[e_{\text{cond}}] = \text{False} \end{cases} \]

\[ \mathcal{B}[e_{\text{then}}] \ \varepsilon[e_{\text{then}}] = \mathcal{B}[e_{\text{else}}] \ \varepsilon[e_{\text{else}}] \]

\[ \mathcal{B}[e_{\text{else}}] \cap \mathcal{B}[e_{\text{then}}] = S \]

\[ \text{otherwise} \]

As explained in Section 5.4, QvtMix uses a hybrid approach for the final determination of binding-time values of expressions. The offline phase’s rules were denoted in the previous subsection. The above set of rules specify the online BTA phase. Function \( \mathcal{B}[] \) computes the online binding-time of its argument based on an environment and according to the offline binding time information passed to it. A notable rule is the one used for refining the binding-time of If expressions. Because the static values are now available to the partial evaluator, the value of the condition expression can be consulted and a decision based on that be made. If the binding time of the condition is static and its value is \text{True}, the binding-time of the If-expression is equivalent to that of its then-condition. Also, if the binding time of the condition is dynamic, but both branches are static and evaluate to the same value, the binding time of the If-expression is static.
5.5.3 Constant Propagation and Static Expression Evaluation

The first example demonstrate partial evaluation for expressions whose operands are constant or static. The variables holding the results of these expressions are also determined to be static. Consequently, the succeeding expressions dereferencing these variables are also statically pre-evaluated. In the following example variable \( x \) is initialized with an expression consisting of two constant sub-expression. The offline BTA evaluates each operand as static. Therefore, the initialization expression is also tagged as static, and so is the variable \( x \). The binding time for the variable is then stored in the binding-time environment which the offline BTA maintains. Proceeding to the definition of variable \( y \), the offline BTA resolves operand 2 as static, and fetches from the binding-time environment its prior resolution of the binding-time of the variable \( x \) (which was also static); hence variable \( y \)'s binding time is also determined as static and the binding-time assignment is added to the environment.

```plaintext
main() {
    var x := 1 + 1;
    var y := x + 2;
    var z := y * 2;
    var t := x + z;
}
```

The `reduce` strategy is employed for rewriting all of the expressions above. Due to the fact that the expressions are all side-effect free, the statically evaluated values are simply in lieu of the original initialization expressions. It is intuitively evident that the two pieces of code above behave exactly the same, but the specialized one on the left performs considerably fewer number of computations than the one on the right.

5.5.4 Loops

Loop bodies are assessed for the existence of side-effects. If the side-effects of the body of a loop only affect the value of some variables and those values depend only on static expressions, then the loop is replaced by the reduce strategy with a number of assignments to variables whose values are modified as a result of executing the loop. This is exemplified in the following piece of code.

```plaintext
main() {
    var x : Integer := 2;
    var y : Integer := 4;
    var z : Integer := 8;
    var t : Integer := 10;
}
```
However, if the loop body depends on a dynamic value, or if it has side-effects, the mixReduce strategy is invoked. The following example remains unchanged due to variable size having dynamic binding time.

The following piece of code also remains unchanged due to the side effect of the log expression.
5.5.5 Conditional Expressions

The following piece of code however remains unchanged because the condition expression is cannot be statically reduced.

In the following example, even though the condition expression is dynamic, both branches have side-effect free expressions. QvtMix eliminates the if-expression, replacing it by the value of the branches. Note that, QvtMix is able to recognize side-effects; if each branch were a `log` statement, albeit identical, the if-expression would remain intact.
5.5.6 Symbolic Expression Simplification

The partial evaluator performs various forms of algebraic manipulation of expressions, so as to simplify the generated code and also eliminate local redundant sub-expressions. The following snippets, illustrate this:

```plaintext
main() {
    var books := bm.objectsOfType(Book)->asOrderedSet();
    var x := books->size();
    var y := (2 * x + 1) + (3 * x - 4);
}
```

→

```plaintext
main() {
    var books : OrderedSet(Book) := bm.objectsOfType(Book)->asOrderedSet();
    var x : Integer := books->size();
    var y : Integer := 5 * x - 3;
}
```
5.5.7 Reduce Rules

\[ R \exp e \sigma = \langle e \rangle \sigma \quad R e e' = R e \uplus R e' \]

where \[ \uplus : \langle \exp \rangle \times \env \times (\env \rightarrow \langle \exp \rangle \times \env) \rightarrow \langle \exp \rangle \times \env \]

\[ (c, \sigma) \uplus f = (c \cup \pi_1(f(\sigma)), \pi_2(f(\sigma))) \]

\[ R \while \cond \do body = SE \body \]

\[ R \if \cond \then \then \else \else = \begin{cases} R \then \exp \cond = \true & \text{if } \cond \exp \then \exp \cond = \false & \text{else} \end{cases} \]

\[ R \var x := e \sigma = (SE e \cup \langle \var x := \exp \rangle) \sigma(x \mapsto \exp e) \]

\[ R x := e \sigma = (SE e \cup \langle x := \exp e \rangle) \sigma(x \mapsto \exp e) \]

\[ R \return e = SE e \cup \langle \return \exp e \rangle \]

The example introduced in this subsection are all reduced using a collection of reduction rules, some of which are presented above. The reduce function, \( R \exp \), takes as input an expression and an environment, and returns a sequence of expression re-writing the original one along with a modified environment. In the rules above, the side-effect evaluation function, \( SE \), is a function that besides evaluating expressions, also computes their possible side-effects and returns a list of expressions that mimic those side-effects, e.g., assignment to modified variables in the reduction of while loops as discussed earlier.

The rule for reducing sequence of statements uses a special operator that denotes the plumbing required to pass modified store from one statement to another, and also amalgamate the results of the reduction of each expression with those of its predecessor. Its definition basically states that the returning expression list is the concatenation of the expression list given as its first argument with what its second argument, a function, cal-
culates. The returning environment is the result of applying the second argument to the passed environment (often processed by the earlier statement). The signature for this operator is devised so as to incorporate that of the reduce function as its arguments.

The reduction of variable initialization and assignment, as expected, modify the environment, mapping the variable referenced by the expression to the value of the right hand side of the expression. Conditional expressions are reduced to one of their branches depending on the value of their condition expression.

### 5.5.8 Contexts and Function Calls

The examples in this section exemplify several of the strategies used for partially evaluating function calls to ordinary functions (i.e., helper functions in QVT-OM). Due to the special semantics and peculiarities associated with mapping calls, they are discussed in a separate section. The first example is series of call to a unary pure function that calculates the square of its argument. QvtMix is able to evaluate the function with the static argument and substitute the function call with it. As the example demonstrates, QvtMix properly handles the case of calls with variables (with static binding times) as well as nested calls.

```plaintext
main() {
    var x := sqr(2);
    var y := x;
    var z := sqr(x);
}

helper sqr(i : Integer) : Integer
    { return i * i; }
```

Calls to recursive functions with no side-effects with arguments having static binding time are also similarly reduced to their statically evaluated values.

```plaintext
main()
    { var x : Integer := 4;
    var y : Integer := 16;
    var z : Integer := 256;
    }
```
In general, function calls cannot be always reduced to a singular value. The following example deals with the case when a binary function is invoked with a static and a dynamic argument. In this case, QvtMix specializes the called function, namely mut, with respect to the static value. The specialized function is a unary function that only takes the dynamic value, and whose constituting expressions are reduced according to the static value bound to its first argument (i.e., i is bound to two). Finally, a new call to the specialized function is inserted in the calling expression in lieu of the original call.

If the called function has side-effects then the call is not reduced. The following thus example remains intact.
If the function has side-effects and multiple arguments, then partial reduction is performed. In the following example, the `hello` function has a statement with side-effect (i.e., `log`) and is called with one static argument, viz., `s`. A specialized version of this function is created that inlines the static value.

A peculiar case is the partial evaluation of recursive functions that have side-effects. QvtMix is able to reduce such calls by creating a specialized function for each iteration of the call as it descends through the recursive calls of the function and chaining them together to produce the same computational and side effects. In the following example, the specialization of the call to the `fact` function yields three different specializations cascaded to one-another to emulate the original recursive chain.
5.5.9 Reducing Iterate Expressions

As discussed in Chapter 5.2, QVT-OM and OCL offer a number of collection iteration constructs aiming to facilitate the inquiry and manipulation of model element containers. More specifically, \texttt{collect} and \texttt{select} (and their imperative counterparts \texttt{xcollect} and \texttt{xselect}) provide succinct mechanisms to apply respectively mapping or query operations on collections. QvtMix is capable of partially evaluating these constructs under a variety of conditions. The first example illustrates the effect of partial evaluation when the collection over which the iterate expression is applied. On the left hand-side, variable \texttt{xs} is assigned to a static value of type \texttt{OrderedSet(Integer)}. The next statement is a \texttt{xselect} query that runs over \texttt{xs}, returning a collection comprising those elements of \texttt{xs} that are greater than two. This expression can hence be fully evaluated at specialization-time. In contrast, the first part of the last statement—i.e., \texttt{b.chapters->xcollect(nbPages)}—creates a collection from a dynamic source expression (i.e., \texttt{b.chapters}). As such, it can only be evaluated at compile time. The argument of the \texttt{union} function call in the second part of the last statement, however, only depends on \texttt{ys}, a static variable. It, therefore, is fully reduced to a static value and is inlined in place of the argument of the \texttt{union} function call. The example assumes that the annotations of the input meta-model are somehow that the expression \texttt{b.chapters->collect(nbPages)} is determined to be fully-dynamic (e.g, property \texttt{nbPages}.}
is declared as `VAR`).

```plaintext
main() {
    var b := bm.objectsOfType(Book)->asOrderedSet()->first();
    var xs := OrderedSet{1,2,3,4};
    var ys := xs->xselect(x|x > 2);
    var zs := b.chapters->collect(nbPages)->
              union(ys->xcollect(y|y *y));
}
```

5.5.10 Mixed Reduction of Partially Static Collections

Insertion of new elements into containers of model elements is effectively tantamount to appending new values at the end of collections in the domain of QVT-OM expressions. Therefore, several strategies of mix reduction of dynamic collections deal with the recurring pattern where collection related operations are performed over a sequence that can have extra elements in the future. Given an ordered collection such as \( C \), if we can somehow divide it into two sub-sequence of old and new elements, then for certain operations we can reuse the result of the previous run of the transformation over the old part, perform the operation on the new elements and finally merge the two, thereby saving redundant computations that would otherwise be performed processing the old elements. Figure 5.7 summarizes the steps required to reduce a collection iterate expression under a fixed context.

In the following example, assume that:

\[
b = \langle \langle \emptyset, \langle "ch1", "20" \rangle, \emptyset, \text{Chapter} \rangle, \langle \emptyset, \langle "ch2", "10" \rangle, \emptyset, \text{Chapter} \rangle \rangle, \langle "lib1", "book1" \rangle, \emptyset, \text{Book} \rangle
\]
... var ys := xs->xselect(xlp(x));

Iterate SubCollection

__temp_1Sub->xselect(xlp(x))

Coalesce with Static Part

var ys := __temp_1Sub->xselect(xlp(x))-
       union(OrderedSet{x1, x2,…})

Figure 5.7: Monovariant Mix Reduction

main() {
    var b := bm.objectsOfType(Book)->
        asOrderedSet()->
        first();

    var __temp_1 := b.chapters;
    var __temp_1Last := 2;
    var __temp_1Size := __temp_1->size();
    var __temp_1Sub :=
        if (__temp_1Size > __temp_1Last) then
            __temp_1->subOrderedSet(__temp_1Last +
            1, __temp_1Size)
        else
            OrderedSet()
        endif;

    var zsSub := __temp_1Sub->xcollect(temp1 :
        Chapter|temp1.nbPages);

    var zs : Sequence(Integer) := zsSub->union(
        Sequence(20, 10)->asBag());
}

129
The right hand side piece of code has three auxiliary variables:

1. __temp_1Last: a static variable which holds the position of the last static element of the collection
2. __temp_1Size: a variable assigned to the dynamic size of the collection
3. __temp_1Sub: the dynamic sub-collection of the source

Variable __temp_1Sub extracts the dynamic part of this partially static structure, i.e., those elements that are added to the model after the partial evaluation of the transformation. They are available during the execution time. This separation is done by the aid of the __temp_1Last variable, which holds the size of the collection at the time of partial evaluation.

The last two lines is where the operation is performed and merged with the static result. First, the xcollect operation is invoked on the dynamic fragment of the collection stored in __temp_1Sub, the result of which is stored in variable zSub. The initialization expression of the original variable (i.e., z) is assigned to an expression which is the merge of zSub with the result of the static evaluation of the expression on the existing collection, which for our example is Sequence{20,10}. The merge of these two collections are done using the union function.

There are often cases where embedding the result of the static evaluation into the host expression requires adjustment to the type of the computed expression to satisfy the QVT-OM type-checker. The example listed above is one such case, and as it demonstrates, QvtMix is able to properly augment residual expressions with the required glueing type-cast operations.
5.5.11 Mix Rules

\[ \mathcal{M}[\cdot] : \text{Exp} \times \text{Env} \rightarrow \text{Exp} \times \langle \text{Exp} \rangle \times \text{Env} \]

\[ \mathcal{M}_{\text{Exp}}[e] = (e, \emptyset) \quad \mathcal{M}[e ; e'] = \mathcal{M}[e] \oplus \mathcal{M}[e'] \]

let \((T_s, T_r, \langle (a_i, T_i) \rangle, e_{\text{body}}) \leftarrow \zeta(f)\)

\[ \mathcal{M}[e_s . f(e_1, \ldots, e_n)] = \begin{cases} \langle \pi_1(\mathcal{M}[e_s]), f(e_1, \ldots, e_n), \pi_2(\mathcal{M}[e_s]) \rangle & \text{bt} = S \\ \langle \pi_1(\mathcal{M}[e_s]), \map{f_{\text{mix}}}(e_1', \ldots, e_n'), E \rangle & \text{otherwise} \end{cases} \]

\[ \mathcal{M}[e_s . \text{map } f(e_1, \ldots, e_n)] = \begin{cases} \langle \pi_1(\mathcal{M}[e_s]), \map{f}(e_1, \ldots, e_n), \pi_2(\mathcal{M}[e_s]) \rangle & \text{bt} = S \\ \langle \pi_1(\mathcal{M}[e_s]), \map{f_{\text{mix}}}(e_1', \ldots, e_n'), E \rangle & \text{otherwise} \end{cases} \]

where \(\text{bt} = \mathcal{B}[e_s] \cap \left( \bigcap_{i=1}^n \mathcal{B}[e_i] \right)\)

\(\tau' = \tau[a_i \mapsto \mathcal{B}[e_i]]_{i=1..n}, \text{self} \mapsto \mathcal{B}[e_s]\)

\(\sigma' = \sigma[a_i \mapsto \mathcal{E}[e_i]]_{\mathcal{B}[e_i] = S}, \text{self} \mapsto \mathcal{E}[e_s]\)

\(f_{\text{mix}} = \mathcal{P} \mathcal{E}[f] \sigma' \tau'\)

\(e_i' = \begin{cases} \pi_1(\mathcal{M}[e_i]), \mathcal{B}[e_i] = D \\ \mathcal{E}[e_i], \mathcal{B}[e_i] = S \end{cases} \)

\(E = \pi_2(\mathcal{M}[e_s]) \cup \bigcup_{\mathcal{B}[e_i] = D} \pi_2(\mathcal{M}[e_i])\)

\[ \mathcal{M}[\text{var } x := e] = (\text{var } x := e_{\text{mix}}, E_{\text{mix}}) \]

\[ \mathcal{M}[x := e] = (x := e_{\text{mix}}, E_{\text{mix}}) \]

where \((e_{\text{mix}}, E_{\text{mix}}) = \mathcal{M}[e] \setminus \mathcal{E}[e]\)

\[ \mathcal{M}[\text{if } e_{\text{cond}} \text{ then } e_{\text{then}} \text{ else } e_{\text{else}}] = \begin{cases} \{ \mathcal{M}[e_{\text{then}}] \} & \mathcal{E}[e_{\text{cond}}] = \text{True} \\ \mathcal{M}[e_{\text{else}}] & \mathcal{E}[e_{\text{cond}}] = \text{False} \end{cases} \quad \mathcal{B}[e_{\text{cond}}] = S \]

\[ \mathcal{M}[\text{return } e] = (\text{return } \pi_1(\mathcal{M}[e]), \pi_2(\mathcal{M}[e])) \]

The rules for mix reduction strategies used by QvtMix are presented above. Similar
to the reduce function, the mix function takes an expression and an environment. It, however, returns a surrogate expression, directly replacing the reduced one, as well as a list of auxiliary expressions that are inserted before the expression being reduced. These auxiliary expressions are often variables used to store intermediate results, or glue code for, among other things, satisfying the type-checker. It also returns a modified environment capturing changes made to variables to enable the correct static evaluation of succeeding expressions that reference those variables.

The same combinator used for the reduce strategy is also used here for combining the results and the effects of sequences of expressions. Of particular importance are the two rules pertaining to calls to helper functions and mapping operations. In both cases, a specialized function called $f_{\text{mix}}$ is created by applying the partial evaluator to an environment that corresponds to the context of the function. The static arguments are evaluated and bound to their corresponding parameters. Variable `self` is also bound to the value of the source expression. The partial evaluator produces a version of the function that takes only the arguments whose binding-time are dynamic. The expressions in the body of the function are mix-reduced based on the static values bound to static parameters.

For assignment and variable initialization expressions, QvtMix uses the merge operator (denoted in the rules by $\gamma$) for coalescing the static sub-expressions with the residual ones. Merge is a polymorphic operator that dispatches different strategies for merging based on the type of its operands. Several interesting cases of merging are presented in the following.

<table>
<thead>
<tr>
<th>$\gamma$: $(\text{Exp} \times \langle\text{Exp}\rangle) \times \text{Val} \times \text{Env} \rightarrow \text{Exp} \times \langle\text{Exp}\rangle$</th>
<th>$(e, E) \gamma \nu = (e, E)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(e_s \rightarrow \text{sum()}, E) \gamma \nu = ([e_s \rightarrow \text{sum()} + [\nu]], E)$</td>
<td></td>
</tr>
<tr>
<td>$(e_s \rightarrow \text{select}(x</td>
<td>e_p), E) \gamma \nu = (e_s \rightarrow \text{select}(x</td>
</tr>
<tr>
<td>$(e_s \rightarrow \text{collect}(x</td>
<td>e_c), E) \gamma \nu = (e_s \rightarrow \text{collect}(x</td>
</tr>
</tbody>
</table>

The following two rules elaborate the discussed methodology used for the mix reduction of `select` and `collect` expressions.
These two rules are based on the fact that these two lemmas for combining the result of these operations on new values in a collection with those obtained from the previous evaluation of the expression on the old collection.
Lemma 5.5.1

\[ \mathcal{E}[c_1 \uplus c_2 \rightarrow \text{select}(x|e_p)] = \mathcal{E}[c_1 \rightarrow \text{select}(x|e_p)] \uplus \mathcal{E}[c_2 \rightarrow \text{select}(x|e_p)] \]

and

\[ \mathcal{E}[c_1 \uplus c_2 \rightarrow \text{collect}(x|e_c)] \sigma = \mathcal{E}[c_1 \rightarrow \text{collect}(x|e_c)] \sigma \uplus \mathcal{E}[c_2 \rightarrow \text{collect}(x|e_p)] \sigma \]

**Proof.** Straightforward induction on \( c_2 \) using the \( \text{E - select} \) and \( \text{E - select - Empty} \) big-step operational semantics rules presented in Appendix C. Base case \( c_2; \sigma \Downarrow \langle \nu_1 \rangle \). Induction step: if for \( c_2; \sigma \Downarrow \langle \nu_1, .., \nu_n \rangle \) the lemma holds, then it also holds for \( c_2; \sigma \Downarrow \langle \nu_1, .., \nu_n, \nu_{n+1} \rangle \). The proof of the second equation is similar to this one.

In the residual code emanated for mixed reduction of \( \text{select} \) and \( \text{collect} \), there are a number auxiliary variables. More specifically, variable \( s \) is the mix of the source expression, variable \( l \) holds the position of the static part of the collection and variable \( z \) holds the dynamic size. Variable \( u \) is the new source for the operation call, which picks the dynamic part of the collection, i.e., the elements appended at the end of the static collection of size \( l \).

For the case of \( \text{select} \) expression, the query is run over the dynamic fragment of collection (referenced as \( u \)). In contrast, the reduced code for \( \text{collect} \) applies the mixed version of the collect expression on the static portion of the collection, which should be separated at runtime using the call to \text{subOrderedset} \ library function. The reason is that, the \( \text{collect} \) expressions can be used for applying, among other things, mapping operations on collections. A residual version of these operations should still be applied on the static part of the collection. This has to do with the need to support the situations where imperative mapping operations may access values from model elements other than those they are directly applied on. The transformation scenario presented in Section 5.6 exemplifies such a situation. In both cases, when the expression is assigned to a variable (or used to initialize one) the residue expression is merged with the literal value obtained from the static evaluation of the expression during the time of partial evaluation (see previous rules for variable assignment and initialization).
5.5.12 Partial Evaluation of Mapping Operations

Mapping operations are the chief constructs of QVT-OM for model transformations. The call to a chain of mapping operation is distinguished from calling helper functions by the property that the returning call effectively instantiates a new object and serializes the returning objects to the output model. A mapping operation is a of a variable context and is typically applied on model elements of different values. It is not practical to produce a specialized version of the mapping operation for each single model element in the input. Thus, the mapping operations need to be specialized in a way that they can correctly transform the existing model elements as well as the new ones. The general theme of polyvariat-mix strategies employed for mapping operations is to place the static values for each invocation of the mapping operation in a static tables rather than inline them as the vanilla mix does.

For the examples in this section, consider the metamodels depicted in Figure 5.8. The input meta-model on the right hand side is a domain model containing a simple containment hierarchy. The containing class, Book, has two attributes and one container. The contained types, Chapter, has also two properties: nbPages and title that denote the number of pages and the title of the chapter, respectively. The metamodel on the right-hand side contains the type Publication that is an abstraction of the two classes on the left.

The following transformation creates a Publication class for each Book instance. It assigns the name attribute of the created Publication instance to the title attribute of the corresponding Book, and aggregates the nbPages of all chapters of the book into the nbPages attribute of the target model element, i.e., Chapter instance.
Figure 5.8: PUB and BOOK Metamodels

```plaintext
transformation Book2Pub(in bm : BOOK, out lm : PUB);

main() {
  var b := bm.objectsOfType(Book)->asOrderedSet()->last();
  var x := b.map M();
}

mapping Book::M() : Publication {
  title := self.name.toUpper();
  nbPages := self.chapters->xcollect(nbPages)->sum();
}
```

The mapping operation M is an example of a mapping that does not create a one-to-one correspondence between the elements of the source and target. It is in fact what is referred to in the technical literature of function programming as catamorphism[12]. These are the type of mappings that collapse the structure of their input into a smaller, aggregate unit, thereby loosing some information from the input but projecting a summary representation thereof. The white-box approach is particularly apt for successfully handling these kinds of transformations, as other approaches, including the black-box synchronization scheme introduced in Chapter 4, have several shortcomings in establishing traces for non-monotonic transformations.

136
The transformation above exemplifies some of the common characteristics of model transformations specified in such hybrid languages as QVT-OM. In this language, primarily due to its OCL heritage, several transformation rules operate on collections of model elements that are selected based on context-dependent criteria. The set of input elements in the source model is divided into a set of FIXED elements whose values and relationships are known and will not change, and variable elements, labeled as VAR. Such information about model elements are annotated in the source metamodel. Consider the model transformation above to be applied routinely on a source model to which progressively more elements of type Chapter are added. The metamodel annotations for this change is presented in Figure 5.9.

![Figure 5.9: Annotations for static analysis](image)

This particular annotation labels Book class as FIXED, which indicates that neither any new instances of this class will be added, nor will any of the existing ones be removed from the input model. In contrast, chapters composition association is labeled as VAR to declare that the future iterations of this model will have new instances of Chapter (For simplicity, we only consider addition here). All attributes of classes have FIXED annotations which means their values are invariant.

The entry operation of Book2Pub transformation, i.e., main, is transformed according to the mix reduction strategy. When the mapping operation call on line 5 is encountered, QvtMix creates a specialization of the called mapping, to wit, M_mix and creates a call expression with the same source to the new mapping:
The body of the mapping is specialized according to the polyvariant mix reduce strategy. What distinguishes this strategy from the simple mix strategy is its caching of different contextual intermediate values in global static tables rather than embedding them in the code. For mapping $M$ three such global tables are created. Each table is basically a dictionary that maps the context of the mapping operation to the value of the cached variable. The first cached value pertains to the first statement in $M$ on line 8. The computed static value of $title$ is held in a dictionary (line 3 below). The two other caches correspond to the $xcollect$ statement in the mapping operation. As discussed before, one keeps track of the position of static values in the collection. The third one caches the static result computed for variable $nbPages$, i.e., the value of the built-in operation $sum$.

The listing in the following presents the mixed mapping operation. The statement on line 7 sets the value of the variable that keeps the last position of the static part of the $self.chapters$ collection, which is read from the corresponding table by the $get$ operation. Since the archetype of contexts of mapping operations are model elements and they may be large structures, they are quite possibly not very efficient if directly used as keys for hashing the static values in tables. Instead, the address of the model elements in the input models are used as keys for the dictionaries; hence the use of $path$ operation on line 7 to get the path of the object in the model.

```
property __M__mix__result_nbPagesCache : Dict(String, Integer) = Dict {
  '/allBooks.4' = 70
};

property __M__mix__temp_1LastCache : Dict(String, Integer) = Dict {
  '/allBooks.4' = 2
};

property __M__mix__result_titleCache : Dict(String, String) = Dict {
  '/allBooks.4' = 'ODYSSEY'
};
```

The $VAR$ annotation on $chapters$ container entails that the value of $Chapter::nbPages$ attribute depends on the chapters that will be added later on. However, by specializing this
transformation for addition of input elements, the partial evaluator can infer the result for the existing chapters. In line 11, the result of the \texttt{sum} operation performed on the dynamic fragment of \texttt{self.chapters} is merged with the static value which is similarly fetched from the first cache.

```plaintext
mapping Book::M__mix() : Publication {
    object result : Publication@lm {
        result.title := self.name.toUpper();

        var __M__mix__temp_1 := self.chapters;
        var __M__mix__temp_1Last := __M__mix__temp_1LastCache->get(self.oclAsType(EObject ).path());
        var __M__mix__temp_1Size := __M__mix__temp_1->size();
        var __M__mix__temp_1Sub := if (__M__mix__temp_1Size > __M__mix__temp_1Last) then __M__mix__temp_1->subOrderedSet(__M__mix__temp_1Last + 1, __M__mix__temp_1Size) else OrderedSet{} endif;
        var __result_nbPagesSub := __M__mix__temp_1Sub->xcollect(temp1 : Chapter|temp1.nbPages)->sum();
        result.nbPages := __result_nbPagesSub + __M__mix__result_nbPagesCache->get(self.oclAsType(EObject).path());
    }
}
```
5.5.13 Polyvariant Mix Rules

This section presents the formal definition of the polyvariant reduction rules, used for the partial evaluation of expressions in that can be executed in multiple contexts, e.g., those in the body of mapping operations. Similar to other reduction rules discussed earlier, the polyvariant mix reduce function, $PM$, takes an expression as its argument and returns a primary replacement expression, along with a list of auxiliary expressions.

$PM : Exp \times Context \times Env \times \Psi \rightarrow Exp \times \langle Exp \rangle \times Env \times \Psi$

$\Psi = Context \times Var \rightarrow (Addr \times Val)$

$PM[e; e'] = PM[e] \oplus PM[e']$

$PM[\text{var } x := e \mid c \sigma \Psi] = (\text{var } x := e_{mix}, E_{mix}, \sigma'', \Psi[c :: x \mapsto (Addr_M(\sigma(self)), \mathcal{E}[e])])$

$PM[x := e \mid c \sigma \Psi] = (x := e_{mix}, E_{mix}, \sigma'', \Psi[c :: x \mapsto (Addr_M(\sigma(self)), \mathcal{E}[e])])$

where

$\sigma' = \sigma[x \mapsto \mathcal{E}[e]]$

$l = [\Psi(c :: x)]->\text{get}(Addr_M(\sigma(self)))$

$(e', E') = M[e] \setminus l$

\[
(e_{mix}, E_{mix}, \sigma'') = \begin{cases} 
(u->f(e_1, .., e_n), \ E' \psi \langle \text{var } u := e_s \rangle, \ \sigma(u) = \bot \\
\sigma'[u \mapsto \mathcal{E}[e_s])
\end{cases}
\]

where $e \neq e' = e_s->f(e_1, .., e_n)$

Besides the usual variable store environment, the polyvariant mix function takes two additional argument. The first one signifies the context in which the expression is being reduced. The second one is the collection of the static caches created by QvtMix to memoize the intermediate results of expressions for each context. This table, represented
by $\Psi$, associates to each variable and context a static cache that maps the address of each model element to the value that the variable assumes when the context is executed over said model element. More specifically, if the mapping operation $\text{Context::op()}$ is invoked over a collection of model elements (e.g., $\text{OrderedSet\{m1, m2, m3\}}$--$\text{map op()}$), then each cached result in the body of $\text{op}$ is a mapping from the addresses of $m1$, $m2$ and $m3$ to the value of the expression when $\text{op}$ is invoked on the respective element.

The rules for polyvariant reduction of assignment and variable initialization expressions are essentially the same as their respective mix reduction rules with two important difference. A static cache is added to hold the static value for current context. For expressions other than calls, a new initialization expression is formed by merging the mix reduced version of the original initialization expression with a lookup expression that fetches the contextual static value from the cache (represented by $l$). For call expressions, an auxiliary variable, $u$, is also created, which is initialized to the static value of the source of the call expression. The original source expression for the call, viz. $e_s$, is replaced by this variable.

5.5.14 Polyvariant Reduction of Iterate Expressions

The following two rules present the polyvariant mix reduction of collection iterate expressions. Again, the most prominent difference with the vanilla mix rules are the use of static caches, in lieu of inline literals, to represent static values. In both cases, the variable holding the position of the last static element, namely $l$, is initialized with a lookup expression to read the value for the context from its pertinent cache. The modified $\Psi$ states caching of the static value of $l$. 

141
\[ \text{PM}[\langle e_s \rightarrow \text{select}(x \mid e_p) \rangle] \circ \psi = \]
\[
( u \rightarrow \text{select}(x \mid e_p),
\pi_2(\text{PM}[\langle e_s \rangle]) \circ
\langle
\text{var } s := \pi_1(\text{PM}[\langle e_s \rangle]),
\text{var } l := \prod_{\psi(c : x)} \rightarrow \text{get(Addr}_M(\sigma(\text{self}))),
\text{var } z := s.\text{size}(),
\text{var } u := \text{if } s > l \text{ then } s \rightarrow \text{suborderedSet}(l + 1, z) \text{ else } \text{OrderedSet}\{\}\rangle,\sigma,
\psi[c : l \mapsto (\text{Addr}_M(\sigma(\text{self})), |E[\langle e_s \rangle]|)]
) \]

\[ \text{PM}[\langle e_s \rightarrow \text{collect}(x \mid e_c) \rangle] \circ \psi = \]
\[
( u \rightarrow \text{collect}(x \mid e_c) \rightarrow \text{union}(e_s \rightarrow \text{suborderedSet}(1, l) \rightarrow \text{collect}(x \mid \pi_1(\text{M}[\langle e_c \rangle])),
\pi_2(\text{PM}[\langle e_s \rangle]) \circ
\langle
\text{var } s := \pi_1(\text{PM}[\langle e_s \rangle]),
\text{var } l := \prod_{\psi(c : x)} \rightarrow \text{get(Addr}_M(\sigma(\text{self}))),
\text{var } z := s.\text{size}(),
\text{var } u := \text{if } s > l \text{ then } s \rightarrow \text{suborderedSet}(l + 1, z) \text{ else } \text{OrderedSet}\{\}\rangle \circ
\bigcup_{\gamma_i \in E[\langle e_s \rangle]} \pi_2(\text{M}[\langle e_c \rangle] \sigma[x \mapsto \gamma_i]),\sigma,
\psi[c : l \mapsto (\text{Addr}_M(\sigma(\text{self})), |E[\langle e_s \rangle]|)]
) \]
5.6 A Full Example

To further introduce the capabilities QvtMix we extend the previous example with a few other subtleties that require the expressive power of QVT-OM to be effectively specified. The following is the listing of the transformation. Lines 11, 12 and 13 are the difference between this transformation and the previous example. What these do is to query the source model for other books with the same name, and then make the title of the Publication instance to the form of \texttt{<bookName>_<i>_of_<n>}, where \( n \) represents the total number of books with the same name in the source model, and \( i \) is the index of this particular book in that list.

```plaintext
modeltype PUB uses 'http://pub/1.0';
modeltype bookModel uses 'http://book/1.0';
transformation Book2Pub(in bm : bookModel, out lm : PUB);

main() {
    var books := bm.objectsOfType(Book)->asOrderedSet();
    var x := books->map Book2Pub();
}

mapping Book::Book2Pub() : Publication {
    var books := bm.objectsOfType(Book)->asOrderedSet()->xselect(b|b.name = self.name);
    var index := books->indexOf(self);
    title := self.name + '__' + index.toString() + '_of_' + books->size().toString();
    nbPages := self.chapters->xcollect(nbPages)->sum();
}
```

The full code of the partially evaluated transformation specialized for the input model of Figure 5.10 is reproduced below:

```
QVT Code
```

```plaintext
1 import qvt.mix.util;
2 modeltype ECORE uses.ecore('http://www.eclipse.org/emf/2002/Ecore');
3 modeltype PUB uses pub('http://pub/1.0');
5 transformation Book2Pub(in bm : bookModel, out lm : PUB);
```
Figure 5.10: The Source and Target Instance Models

```plaintext
11 property __Book2Pub__mix__result_nbPagesCache : Dict(String, Integer) = Dict {'/allBooks.2' = 20, '/allBooks.3' = 10, '/allBooks.1' = 20};
12 property __Book2Pub__mix__booksCache : Dict(String, OrderedSet(ObjectPath)) = Dict {'/allBooks.2' = OrderedSet(object ObjectPath {
13 path := '/allBooks.2';
14 }, object ObjectPath {
15 path := '/allBooks.1';
16 }, object ObjectPath {
17 path := '/allBooks.3';
18 }), '/allBooks.3' = OrderedSet(object ObjectPath {
19 path := '/allBooks.1';
20 }, object ObjectPath {
21 path := '/allBooks.3';
22 });
23 property __Book2Pub__mix__indexCache : Dict(String, Integer) = Dict {'/allBooks.3' = 2, '/allBooks.2' = 1, '/allBooks.1' = 1};
24 property __Book2Pub__mix__temp_LastCache : Dict(String, Integer) = Dict {'/allBooks.3' = 1, '/allBooks.1' = 2, '/allBooks.2' = 2};
25 property __Book2Pub__mix__result_titleCache : Dict(String, String) = Dict {'/allBooks.2' = 'book2_1_of_1', '/allBooks.3' = 'book1_1_of_2', '/allBooks.1' = 'book1_1_of_2'};
26 property __Book2Pub__mix__temp_LastCache : Dict(String, Integer) = Dict {'/allBooks.2' = 3, '/allBooks.3' = 3, '/allBooks.1' = 3};
```
```plaintext
main() {
  var books : OrderedSet(Book) := bm.objectsOfType(Book)->asOrderedSet();
  var __temp_1 := books;
  var __temp_1Size := __temp_1->size();
  var __temp_1Sub := if (__temp_1Size > __temp_1Last) then __temp_1->subOrderedSet(__temp_1Last + 1, __temp_1Size) else OrderedSet() endif;
  var x : Sequence(Publication) := __temp_1Sub->xselect(b : Book|b.name = self.name)->union(books->subOrderedSet(1, __temp_1Last)->xcollect(temp1 : Book|temp1.map(Book2Pub()))->union(books->subOrderedSet(1, __temp_1Last)->xcollect(temp1 : Book|temp1.map(Book2Pub__mix())));
}

mapping Book::Book2Pub() : Publication
{
  object result : Publication@lm {
    var books : OrderedSet(Book) := bm.objectsOfType(Book)->asOrderedSet()->xselect(b : Book|b.name = self.name);
    var index : Integer := books->indexOf(self);
    result.title := self.name + '__' + index.toString() + '_of_' + books->size().toString();
    result.nbPages := self.chapters->xcollect(temp1 : Chapter|temp1.nbPages)->sum();
  }
}

mapping Book::Book2Pub__mix() : Publication
{
  object result : Publication@lm {
    var __Book2Pub__mix__temp_1 := bm.objectsOfType(Book)->asOrderedSet();
    var __Book2Pub__mix__temp_1Last := __Book2Pub__mix__temp_1LastCache->get(obj.oclAsType(EObject).path());
    var __Book2Pub__mix__temp_1Size := __Book2Pub__mix__temp_1->size();
    var __Book2Pub__mix__temp_1Sub := if (__Book2Pub__mix__temp_1Size > __Book2Pub__mix__temp_1Last) then __Book2Pub__mix__temp_1->subOrderedSet(__Book2Pub__mix__temp_1Last + 1, __Book2Pub__mix__temp_1Size) else OrderedSet() endif;
    var __booksSub := __Book2Pub__mix__temp_1Sub->xselect(b : Book|b.name = self.name);
    var books := __booksSub->union(__Book2Pub__mix__booksCache->get(obj.oclAsType(EObject).path())->getObject(bm)->oclAsType(Book)->asBag());
    var __indexSub := __Book2Pub__mix__indexCache;
    var index := __indexSub->get(obj.oclAsType(EObject).path());
    result.title := self.name + '__' + index.toString() + '_of_' + books->size().toString();
    result.nbPages := __Book2Pub__mix__temp_2Sub->xcollect(temp1 : Chapter|temp1.nbPages)->sum();
  }
}
```

From Line 69 onward, there are addenda helper functions that implement the path scheme used for addressing objects inside models and storing values in static tables. As these functionalities are missing from the standard library of QVT, the code for them are patched to the end of each specialized transformation that requires them. Now if we add a new book with name book1 to the source model, both original and specialized transformations create an instance of Publication on the target model. Figure 5.11 exemplifies a
Figure 5.11: Applying the Specialized Transformation to the Changed Source Model

scenario wherein a new book named book1 is added to the source model. This change entails the insertion of a new instance of Publication to the target model as well as updating the title attribute of those instances that correspond with older books named book1 to book1<i>_of_3, for there exist three such books now.

5.7 Experiments and Discussion

In this section we present the results of our experiments with the partial evaluation framework. We have used the same transformation, i.e., BookToLibrary described in the previous section over a range of models of various sizes. Our experiments were performed on a Win-
dows XP (Service pack 3) based PC featuring Intel Core™2 CPU clocked at 2.1Ghz, 2GB of physical memory, running Eclipse M2M project’s QVT Operational mapping implementation on top of Eclipse 3.5.

Our first set of experiments involved applying the transformation on models progressively growing in number of elements, and thus in size. More specifically, we used an instance of the Book Figure 5.12 compares the performance of the original transformation, and its specialized version. The As it is evident from the graphs, the difference is negligible for small input models. This has to do with I/O being the dominant factor during the transformation of these models, which is comparable for both transformations. However, as we apply partial evaluation on larger models where the processing is the most time-consuming part and the I/O effect is amortized (i.e., models with more than 100 elements), the performance advantage of partial evaluation becomes conspicuous. The details of this experiment is reported in Table 5.1. The input models were generated by a QVT transformation. The reported elements are the ones for which the BookToLibrary transformation was partially evaluated. In this set of experiments, we consider the change to be the addition of just one chapter to the first book of the first library. The original transformation requires to transform all other non-affected elements, whereas the partially evaluated one exploits its static cache to expedite the reconciliation of the source and target models. Although at first this minimal size of change for empirical analysis can be called into question, it is in fact representative of a common practical scenario. Time and again, developers have to manipulate bits and pieces of colossal models, and no matter how small the change, the full re-transformation of these models are, more often than not, the only possibilities in many existing modeling tools. Partial evaluation provides a viable solution for these scenarios.

<table>
<thead>
<tr>
<th>N_T</th>
<th>N_L</th>
<th>N_B</th>
<th>N_C</th>
<th>T_B2L</th>
<th>T_mixB2L</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.12ms</td>
<td>0.11ms</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>0.12ms</td>
<td>0.11ms</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>0.27ms</td>
<td>0.25ms</td>
</tr>
<tr>
<td>1111</td>
<td>1</td>
<td>100</td>
<td>10</td>
<td>135.6ms</td>
<td>1.75ms</td>
</tr>
<tr>
<td>11011</td>
<td>1</td>
<td>1000</td>
<td>10</td>
<td>10602ms</td>
<td>47ms</td>
</tr>
</tbody>
</table>

Table 5.1: The results of the first set of experiments. N_T: total elements, N_L: no. of Lib, N_B: no. of Book per Lib, N_C: no. of Chapter per Book, T_B2L: Exec. time of BookToLibrary, and T_mixB2L: Exec. time of mixBookToLibrary.
In our second set of experiments, we applied a fixed input model to both transformations, each time instructing the partial evaluator to use a fraction of input for specializing the transformation. This is in effect the same as having the rest of the elements added on the second execution of the transformation. What this experiment assess is whether specializing more expressions inside the program leads to better performance of the specialized program. In particular, we focused in this experiments on the summation of \texttt{nbPages} in the program which was reduced by the specializer. The multiple-valued variables were completely cached for each invocation of the transformation. The results of these experiments are projected in Figure 5.13. We triggered the transformation with a model comprised of 10 libraries, 100 books and 10 chapters, the transformation of which took 10703ms by the original \texttt{BookToLibrary} transformation. We then varied the percentage of input elements used as \texttt{FIXED} and calculated the time of execution of the transformation specialized for those model elements. As expected, the more input elements were being involved in the partial evaluation, the fewer dynamic computations was need to be performed during re-execution, and thus, the transformation took less time. We started from treating all \texttt{VAR} elements (i.e., instances of Chapter) as dynamic (they can be considered as new elements added after the initial transformation), and reduced this by 10% in each step, which re-
sulted in more reduction of transformation time. The full partial evaluation performs more than twice as fast as having no partial evaluation.
Chapter 6

Conclusion

This thesis presented an inquiry into various aspects of change management in the context of model driven software engineering. To this end, we laid the foundation for a formalism to denote models and change operations. An algebraic methodology for computing the impact of change operations on models was introduced. We tackled the problem of factorizing the differences of two models into a number of atomic change operators, devising two different algorithms based on two different sets of atomic change operators. We used this foundation to provide a precise characterization for various notions that we discuss in the later sections of the dissertation. More specifically, the classes of monotonic, homomorphic and uniform transformations were introduced and their properties were studied. These properties, i.e., uniformity and locality, allow for generic change translation between source and target sides of a transformation, which lays the foundation for the black-box synchronization approach. We also provided definitions for incrementality and bi-directionality in the context of model transformations.

After introducing the foundation material, the thesis focused on the important problem of model synchronization, for which it proposed a two-fold solution. First, a novel methodology to build synchronization around existing uni-directional and non-incremental transformations was introduced. This technique differs from the previous undertakings primarily in the fact that it uses the original artifact generators as black boxes. As such, other than some generic assumptions about the type of the transformations, it needs no detailed knowledge of the consistency rules. In contrast to the approaches based on incremental transformation engines, our proposed black-box model synchronization framework neither
needs denotation of transformations in a new language, nor it requires the transformations to be re-executed after they are used for the initial creation of target models. The framework, even when used in conjunction with unidirectional artifact generators, is capable of propagating updates in the opposite direction of that of the used artifact generator. The synchronization scheme results in models that comply with the original transformation. The proposed approach is based on a process we refer to as Conceptualization. This process extracts the mutual information of two or more inter-related artifacts and stores them in a central concept pool. Shadow Models are used as input to the transformations for providing effective traceability between concepts and model elements in interlinked models. We utilize the technique to provide instant and incremental propagation of Update changes between models. To support incremental synchronization of Insertion changes, we propose the notion of μ-templates, some localized place holders in shadow models. Treating transformations as black-boxes has the advantage of eliminating the cost associated with reverse engineering of consistency rules between software artifacts. However, the proposed solution, even though covering a wide spectrum of practical model transformations, is limited to a certain class of model transformations.

To extend the gamut of supported transformations, we introduced a complementary, white-box approach to the problem of model synchronization. In this methodology, we analyze the source code of existing model transformations and transform them to semantically equivalent programs that operate in an incremental fashion. In particular, by leveraging a technique called partial evaluation, we avoid crude re-computation of model transformation rules. We presented QvtMix, a partial evaluator for an essential subset of the QVT Operational Mappings language. Through several examples, we demonstrated the use of QvtMix in different specialization scenarios, during which redundant computations were omitted. Our experiments exhibited significant reduction in re-transformation time as the percentage of the input model elements utilized for partial evaluation increases.

6.1 Future Work

6.1.1 Information Content of Models and Model Transformations

An interesting area for future work is to investigate the notion of Kolmogorov Complexity as a representation for the information content of models and that of transformations.
Informally, the Kolmogorov Complexity of a message is defined as the minimum length of a program that can produce the message as its output. It is shown that the choice of language affects this complexity up to a constant order. In MDE, the notion of representing transformations, which generate models, as model instances themselves has been ubiquitously championed (and in fact is a basis for the white-box methodology presented in this dissertation, too). The Kolmogorov Complexity metric can potentially be a useful metric to categorize various kinds of model transformations based on how much of the information in their source models they propagate to their target side, and to what extent they lose information. Another intriguing investigation we propose is to use this to formulate the relationship of higher-order transformations and the meta-meta hierarchies popular in modeling notations such as MOF.

6.1.2 Improved Conceptualization Schemes

The conceptualization scheme proposed here can be improved using information retrieval methods such as Latent Semantics Indexing and Formal Concept Analysis. The idea is to use these methodologies based on existing transformation scenarios to gain insight into the semantic bridges established by the transformation across its domain and co-domain metamodels, and use this knowledge to refine the concepts associated to values in the domain model.

6.1.3 Automatic Generation of Abstractors/DeAbstractors

Model based synchronization techniques are applicable to all types of software artifacts, insofar as they can be represented in a canonical modeling format. The conversion step to create the representation of an artifact, denoted in an arbitrary format, to a canonical representation such as MOF is conducted by so called Abstractors. DeAbstractors do the reverse conversion. A system that can generate these programs automatically (or semi-automatically) from a formal grammar can have immense utility in facilitating the use of model transformations for establishing consistency among myriad of existing formats for software artifacts.
6.1.4 Embedded Rule Language for Composite Concepts

The concept-pool, presented as part of the black-box framework in this dissertation, allows for mappings between concept identifiers and simple values. It, however, can be extended to accommodate expressions that can reference and combine the values of other concepts to increase the expressive power of the framework and provide support for a wider range of model transformations. These embedded rules form a mini-language whose expressions are evaluated by the deShadow algorithm. Adding the ability to reference other concepts can, however, create termination issues. We need to investigate methodologies to ensure that the framework terminates, e.g., to provide guarnatees that concept dependencies are cycle-free.

6.1.5 Fine-Grained Version Management Using Concepts

The ideal of a centralized concept-pool and shadow models can have other applications besides model synchronization. One interesting venue is to use this framework to build a configuration management system for maintaining versions of models. A potentially useful advantage of this approach over the existing file-based solutions such as CVS or SVN would be the availability of independent, fine-grained versions down to the level of concepts. This allows us to exploit concepts in order to pick and choose different versions of each element or attribute to form a view of the history of the evolution of a model in the workspace. We envisage that such a feature should have several practical uses.

6.1.6 Enhancements to QvtMix

QvtMix incorporates many important specialization algorithms and in many cases goes beyond what conventional partial evaluators for general purpose programming languages support. Nevertheless, QVT-OM is a language with extensive grammar and several features, which makes the task of extending QvtMix a continuous endeavor. Furthermore, several improvements to the existing specialization scheme are envisioned. Incorporating data-flow analysis algorithms and inter-procedural optimizations common for optimizing compilers can yield more efficient residual programs.
6.1.7 Self Applicability of QvtMix

QvtMix is written for and in the QVT Operational Mappings language. This is by design: so as to allow for self-applicability. Self-application of partial evaluators, the so called Futamura projections \(29\), have several interesting theoretical and practical properties. The idea is that partial evaluators can be applied on themselves to generate compilers and compiler generators. This requires, at the very least, that the partial-evaluator be implemented in the same language that it processes. Concocting self-applicable partial evaluators, nonetheless, is known to be a non-trivial task and requires multitude of tweaking and tinkering to make sure certain kinds of symmetries in the code of the partial evaluator are observed.
Appendix A

Haskell Implementation of Change and Sync

Haskell Code

```haskell
module Ecore where

data EObject = EPkg EPackage | ECl EClassifier

data EPackage = EPackage {
    pkgName :: EString,
    nsURI :: EString,
    nsPrefix :: EString
} deriving Show

data EClassifier = EClass {
    clName :: EString,
    eStructuralFeatures :: [EStructuralFeature],
    eOperations :: [EOperation],
    eSuperType :: EString,
    instanceTypeName :: EString,
    abstract :: EBool,
    interface :: EBool
} deriving Show

| EDataType {
    clName :: EString,
    cTypeParameters :: [ETypeParameter]
} | EEnum {
    clName :: EString,
    eLiterals :: [EENumLiteral]
} deriving Show

data EENumLiteral = EENumLiteral {
    elitName :: EString,
    value :: EInt
} deriving Show
```
data EStructuralFeature = EAttribute {
    attName :: EString, attType :: EString, defaultValue :: EString, esLowerBound :: EInt, esUpperBound :: EInt, changeable :: EBool, esOrdered :: EBool, volatile :: EBool, id :: EBool, esUnique :: EBool, transient :: EBool
} |
    EReference {
        reName :: EString, refType :: EString, defaultValue :: EString, containment :: EBool, derived :: EBool, eOpposite :: EString, esLowerBound :: EInt, esUpperBound :: EInt, changeable :: EBool, esOrdered :: EBool, resolveProxies :: EBool, transient :: EBool, unsettable :: EBool, volatile :: EBool, esUnique :: EBool
    }
} deriving Show

data EOperation = EOperation {
    opName :: EString, opType :: EString, opLowerBound :: EInt, opUpperBound :: EInt, opOrdered :: EBool, opUnique :: EBool, eGenericType :: EString, opTypeParameters :: [ETypeParameter], eParameters :: [EParameter]
} deriving Show

data EParameter = EParameter {
    pName :: EString, pType :: EString, pLowerBound :: EInt, pUpperBound :: EInt, pOrdered :: EBool, pUnique :: EBool
} deriving Show

newtype ETypeParameter = ETypeParameter EString deriving Show

type EString = String

type EBool = EBoolean

type EBoolean = Bool

type EInt = Int
class ENamedElement a where
  name :: a -> EString

class ETypedElement a where
  eType :: a -> EString

class EMultiplicityElement a where
  ordered :: a -> EBool
  unique :: a -> EBool
  lowerBound :: a -> EInt
  upperBound :: a -> EInt

class HasETypeParameters a where
  eTypeParameters :: a -> [ETypeParameter]

instance ENamedElement EPackage where
  name = pkgName

instance ENamedElement EClassifier where
  name = clName

instance HasETypeParameters EClassifier where
  eTypeParameters = clTypeParameters

instance ENamedElement EEnumLiteral where
  name = elitName

instance ENamedElement EStructuralFeature where
  name (EAttribute {attName = an}) = an
  name (EReference {refName = rn}) = rn

instance ETypedElement EStructuralFeature where
  eType (EAttribute {attType = at}) = at
  eType (EReference {refType = rt}) = rt

instance EMultiplicityElement EStructuralFeature where
  ordered = esOrdered
  unique = esUnique
  lowerBound = esLowerBound
  upperBound = esUpperBound

instance ENamedElement EOperation where
  name = opName

instance ETypedElement EOperation where
  eType = opType

instance EMultiplicityElement EOperation where
  ordered = opOrdered
  unique = opUnique
  lowerBound = opLowerBound
  upperBound = opUpperBound

instance HasETypeParameters EOperation where
  eTypeParameters = opTypeParameters

instance ENamedElement EParameter where
  name = pName

instance ETypedElement EParameter where
  eType = pType

instance EMultiplicityElement EParameter where
  ordered = pOrdered
  unique = pUnique
  lowerBound = pLowerBound
  upperBound = pUpperBound
155 isEAttribute :: EStructuralFeature -> Bool
156 isEAttribute EAttribute {} = True
157 isEAttribute _ = False
159 isEContainer :: EStructuralFeature -> Bool
160 isEContainer EReference {containment = True} = True
161 isEContainer _ = False
163 isEReference :: EStructuralFeature -> Bool
164 isEReference EReference {containment = False} = True
165 isEReference _ = False

-- ---------------------------------------------------------------------------
-- EcoreParser
-- ---------------------------------------------------------------------------
module EcoreParser where

import Data.Maybe(fromMaybe, fromJust)
import Data.List(isPrefixOf, find)
import Data.Char(toLower)
import Text.XML.Light
import Debug.Trace
import Ecore
import Util

getEPackage :: Element -> Maybe EPackage
getEPackage e = case e of
  (Element {elName = QName {qName = "EPackage"}}) ->
    Just EPackage pkgName = getAttr "name" e
      , nsURI = getAttr "nsURI" e
      , nsPrefix = getAttr "nsPrefix" e
    otherwise -> Nothing

getEClass :: Maybe String -> Element -> Maybe EClassifier
getEClass xsi e = if (elName e == (QName "eClassifiers" Nothing Nothing)) &&
  (getXSIType xsi e == Just "ecore:EClass")
  then Just EClass{ cName = getAttr "name" e
    , eStructuralFeatures = getEStructuralFeatures xsi e
    , eOperations = getEOperations xsi e
    , eSuperType = cutPrefix '/' $ getAttr "eSuperTypes" e
    , instanceTypeName = getAttr "instanceTypeName" e
    , abstract = case getAttr "abstract" e of
        "true" -> True; _ -> False
    , interface = case getAttr "interface" e of
        "true" -> True; _ -> False
    }
  else Nothing

getNameSpaces :: Element -> [(String, String)]
getNameSpaces = (map \a->((qName . attrKey) a, attrVal a)) .
  (filter isNameSpace) .
  eAttribs
  where isNameSpace (Attr (QName _ _ (Just "xmlns"})) _ = True
    isNameSpace _ = False

xsiURI :: Element -> Maybe String
xsiURI e = findXSI (getNameSpaces e) >>= Just . snd
  where findXSI = find ((\(k,v) -> k == "xsi"))
getXSIType :: Maybe String -> Element -> Maybe String
getXSIType xsi = findAttr (QName "type" xsi) (Just "xsi")

getAttr at =
  (fromMaybe "") . (findAttr (unqual at))

getEStructuralFeatures :: Maybe String -> Element -> [EStructuralFeature]
getEStructuralFeatures xsi =
  let featElems = findElements (unqual "eStructuralFeatures") in
    map (getEStructuralFeature xsi) . featElems

getEStructuralFeature :: Maybe String -> Element -> EStructuralFeature
getEStructuralFeature xsi e =
  case getXSIType xsi e of
    Just "ecore:EAttribute" -> makeAttribute e
    Just "ecore:EReference" -> makeReference e
    where
      makeAttribute e =
        (EAttribute
          { attName = getAttr "name" e,
            attType = cutPrefix '/' $ getAttr "eType" e,
            defaultValue = getAttr "defaultValue" e,
            esLowerBound = toInt (getAttr "lowerBound" e),
            esUpperBound = toInt (getAttr "upperBound" e),
            changeable = toBool (getAttr "changeable" e),
            esOrdered = toBool (getAttr "ordered" e),
            volatile = toBool (getAttr "volatile" e),
            unsettable = toBool (getAttr "unsettable" e),
            derived = toBool (getAttr "derived" e),
            iD = toBool (getAttr "iD" e),
            esUnique = toBool (getAttr "unique" e),
            transient = toBool (getAttr "transient" e)
        })
      makeReference e =
        (EReference
          { refName = getAttr "name" e,
            refType = cutPrefix '/' $ getAttr "eType" e,
            defaultValue = getAttr "defaultValue" e,
            containment = toBool (getAttr "containment" e),
            derived = toBool (getAttr "derived" e),
            eOpposite = getAttr "eOpposite" e,
            esLowerBound = toInt (getAttr "lowerBound" e),
            esUpperBound = toInt (getAttr "upperBound" e),
            changeable = toBool (getAttr "changeable" e),
            esOrdered = toBool (getAttr "ordered" e),
            resolveProxies = toBool (getAttr "resolveProxies" e),
            transient = toBool (getAttr "transient" e),
            unsettable = toBool (getAttr "unsettable" e),
            volatile = toBool (getAttr "volatile" e),
            esUnique = toBool (getAttr "unique" e)
        })

getEOperations :: Maybe String -> Element -> [EOperation]
getEOperations xsi =
  let opElems = findElements (unqual "eOperations") in
    map (getEOperation xsi) . opElems
getEOperation :: Maybe String -> Element -> EOperation
getEOperation xsi e = EOperation {
  opName = getAttr "name" e,
  opType = cutPrefix '/' $ getAttr "eType" e,
  opLowerBound = toInt (getAttr "lowerBound" e),
  opUpperBound = toInt (getAttr "upperBound" e),
  opOrdered = toBool (getAttr "ordered" e),
  opUnique = toBool (getAttr "unique" e),
  eGenericType = getAttr "eGenericType" e,
  eTypeParameters = getETypeParameters e,
  eParameters = getEParameters e
}

getETypeParameters :: Element -> [ETypeParameter]
getETypeParameters e = let tparElem = findElements (unqual "eTypeParameters") in
  map getETypeParameter . tparElem

getETypeParameter :: Element -> ETypeParameter
getETypeParameter e = ETypeParameter {
  pName = getAttr "name" e,
  pType = cutPrefix '/' $ getAttr "eType" e,
  pOrdered = toBool (getAttr "ordered" e),
  pUnique = toBool (getAttr "unique" e),
  pLowerBound = toInt (getAttr "lowerBound" e),
  pUpperBound = toInt (getAttr "upperBound" e)
}

module Model where

import Ecore
import EcoreParser
import Text.XML.Light
import Data.Maybe
import Data.List
import Util
import Debug.Trace
import Control.Monad.Reader
import System.IO

test = do
  input <- readFile "JavaSrvImpl.ecore.xml"
  xmi <- readFile "JavaClass.xmi"
  let elems = onlyElems (parseXML input)
      pkgEl = head $ tail elems
      epkg = fromJust $ getEPackage pkgEl
      children = elChildren pkgEl
      ecla = map fromJust $
        filter ('\n'-> case n of
Just _ -> True; Nothing -> False) $ map (getEClass (xsiURI pkgEl)) children
metaModel = getMetaModel "JavaSrvImp" ecla
topElem = head $ tail $ onlyElems $ parseXML xmi

print topElem
putStrLn "-------------"
print metaModel
putStrLn "-------------"
let mod = xmiJMMToModel metaModel (getElementType topElem) topElem
    addr = (Root mod) </> 0 <..> 0

putStrLn "-------------"
print mod
print addr
print $ getModel addr
return mod

-- MetaModel

data MetaModel = MetaModelElement {
mmeName :: String
    , mmeContainers :: [(Container, Type)]
    , mmeAttributes :: [(Attribute, Type)]
    , mmeReferences :: [(Reference, Type)]
    , mmeSuper :: String
    } | MetaModel {
mmeName::String
    , mmeElems::[MetaModel]
    } deriving (Show, Read)
-- Model

data Model = Model {
mName :: String
    , mContainers :: [[Model]]
    , mAttributes :: [Attribute]
    , mReferences :: [Reference]
    , mType :: Type
    } deriving (Show, Read, Eq)

-- Addr

data Addr = Root Model | Cont Addr Int | El Addr Int | At Addr Int

model m = Root $ Model {mName = m}

-- Container Selector

(</>) :: Addr -> Int -> Addr
a </> i = Cont a i

-- Element Selector
401 (<.>) :: Addr -> Int -> Addr
402 a <.> i = El a i

404 (-- Attribute Selector
406

408 (<@>) :: Addr -> Int -> Addr
409 a <@> i = At a i

411 getAt :: Addr -> Maybe Attribute
412 getAt (At addr i) = do
413 m <- getModel addr
414 let attribs = mAttributes m
415 if i < length attribs then Just $ attribs !! i else Nothing
416 getAt _ = Nothing

418 getCont :: Addr -> Maybe [Model]
419 getCont (Cont addr i) = do
420 m <- getModel addr
421 let conts = mContainers m
422 if i < length conts then Just $ conts !! i else Nothing
423 getCont _ = Nothing

425 getEl :: Addr -> Maybe Model
426 getEl e@El _ _ = getModel e
427 getEl _ = Nothing

429 getModel :: Addr -> Maybe Model
430 getModel (Root m) = Just m
431 getModel (Cont a i) = getModel a
432 getModel (At a i) = getModel a
433 getModel (El (Cont a c) e) = getModel a >>= getEl c e
434 where
435     getEl :: Int -> Int -> Model -> Maybe Model
436     getEl c e m = let conts = mContainers m
437                     elems = conts !! c
438                     in
439                     if length conts > c && length elems > e then
440                         Just $ elems !! e
441                     else
442                         Nothing
444
445     instance Show Addr where
446     show (Root m) = "/" ++ mName m
447     show (Cont addr c) = (show addr) ++ '/'(show c)
448     show (El addr e) = (show addr) ++ '.'(show e)
449     show (At addr a) = (show addr) ++ '@'(show a)

450     type Container = String
451     type Attribute = String
452     type Reference = String
453     type Type = String

455     emptyModel :: Model
456     emptyModel = Model "" [] [] "Nil_"

458     getMetaModel :: String -> [EClassifier] -> MetaModel
459     getMetaModel name ecla =
MetaModel name $ map fromEcore ecla

fromEcore :: EClassifier -> MetaModel
fromEcore (EClass name features super _ _) =
  MetaModelElement name c a r super
  where
c = map \f-> (refName f, refType f) $ filter isEContainer features
a = map \f-> (attName f, attType f) $ filter isEAttribute features
r = map \f-> (refName f, refType f) $ filter isEReference features

xmiToModel :: Element -> Model
xmiToModel e = Model {
  mName = getElementName e
  , mContainers = getContainers e
  , mAttributes = getAttributes e
  , mReferences = getReferences e
  , mType = getElementType e
}

xmiMMToModel :: MetaModel -> String -> Element -> Model
xmiMMToModel mm ty e =
  let mme = find \(m -> mmeName m == ty) \(mmElems mm)
in case mme of
  Just mme' ->
    Model {
      mName = getElementName e
      , mContainers = getMMContainers mm mme' e
      , mAttributes = getMMAttributes mme' e
      , mReferences = getMMReferences mme' e
      , mType = ty
    }
  otherwise -> emptyModel

getElementName :: Element -> String
getAttributes = getAttr "name"
getAttributes = (map attrVal) . (filter sansPrefix) . elAttribs
  where
    sansPrefix (Attr (QName _ _ Nothing) _) = True
    sansPrefix _ = False

getMMAttributes :: MetaModel -> Element -> [Attribute]
getMMAttributes mme e = [attrVal a | (k,t) <- mmeAttributes mme,
  a <- elAttribs e, qName (attrKey a) == k]

gETElementType :: Element -> Type
gETElementType = qName . eName

gETReferences :: Element -> [Reference]
gETReferences _ = []

gETMMReferences :: MetaModel -> Element -> [Reference]
gETMMReferences mme e = [attrVal a | (k,t) <- mmeReferences mme,
  a <- elAttribs e, qName (attrKey a) == k]
getContainers = (map (map xmToModel)) .
  (groupBy (\el e2 -> elName el == elName e2)).
elChildren

getMMContainers :: MetaModel -> MetaModel -> Element -> [Model]
getMMContainers mm mme e = [xmiMMToModel mm t child] | (cont,t) <- mmeContainers mme, child <- elChildren e, cont == qName(elName child)]

--mapModel f (Nil a t) = Nil (map f a) t
--mapModel f (Model ns a t) = Model (map (mapModel f)) ns (map f a) t

storeLazyModel :: Model -> String -> IO ()
storeLazyModel model path = writeFile path $ show model

loadLazyModel :: String -> IO Model
loadLazyModel model path = readFile path >>= return.read

storeModel :: Model -> String -> IO ()
storeModel model path =
do
  handle <- openFile path WriteMode
  hPutStr handle $ show model
  hClose handle

loadModel :: String -> IO Model
loadModel path =
do
  handle <- openFile path ReadMode
  content <- loadContent handle ""
  hClose handle
  return $ read content
where loadContent h str =
do
  eof <- hIsEOF h
  if eof
    then return str
    else do line <- hGetLine h
           let newS = str++line
           n

import Model
import Concept

m = (Model "M" [[(Model "C1" [] ["clattr1","clattr2"] [] "C")]] ["m1a1"] [] "MC")

upd :: Attribute -> Addr -> Maybe Model
upd v (At addr i) = getModel addr >>= return.updElem i v
upd _ _ = Nothing

updElem :: Int -> Attribute -> Model -> Model
updElem i v m = 
  if i >= 0 && i < length attrs && v /= oldVal then 
    m{mAttributes = newAttrs}
  else 
    m
  where attrs = mAttributes m 
    oldVal = attrs !! i
    newAttrs = let (xs, v':ys) = splitAt i attrs in 
                  xs++v:ys

--- Insert 
---
--- Insert 
---

ins :: Type -> Addr -> Maybe Model 
ins t (Cont addr i) = getModel addr >>= return . insElem t i
ins _ _ = Nothing

insElem :: Type -> Int -> Model -> Model 
insElem t i m = 
  if i >= 0 && i < length containers then 
    m (mContainers = newContainers) 
  else if i == 0 && length containers == 0 then 
    insElem t 0 m (mContainers = [[]])
  else 
    m
  where containers = mContainers m
        newContainers = let (cs, ci:cs') = splitAt i containers in 
                       cs++(cs++(init ci++[emptyModel{mType = t}]):cs')

--- Delete 
---

del :: Addr -> Maybe Model 
del (Cont addr i) = getModel addr >>= return . delElem i 
del _ _ = Nothing

delElem :: Int -> Model -> Model 
delElem i m = 
  if i >= 0 && i < length containers && not (null $ containers !! i) then 
    m(mContainers = newContainers)
  else 
    m
  where containers = mContainers m
        newContainers = let (cs, ci:cs') = splitAt i containers in 
                       cs++(init ci):cs'

--- Concept 
---

module Concept where 

import Control.Monad.State 
import System.Time 
import Data.Map
data Concept = Concept {
    cID :: ConceptID,
    cVal :: String,
    -- , refs :: [Addr]
    -- , history :: [(TimeStamp, String)]
    -- , newMu :: Maybe Concept
    -- |Mu { id :: ConceptID } | Nil { id::ConceptID}
    newMu :: Maybe
}

deriving (Show)

type ConceptID = String

type TimeStamp = ClockTime

data ConceptPool = Map ConceptID Concept

type ConState = StateT ConceptPool IO

newID :: String

newID = undefined

updateCon :: ConceptID -> String -> ConState ()

updateCon cid v =
    get >>= put.update (\c -> Just c{cVal = v}) cid

addCon :: Concept -> ConState ()

addCon c = get >>= put.insert (cID c) c

getch :: ConceptID -> ConState (Maybe Concept)

getch cid = get >>= return . Data.Map.lookup cid

module Shadow where

import Ecore
import EcoreParser
import Model
import Concept
import Data.UUID.V1
import Control.Monad.Trans
import Control.Monad.State
import Data.Maybe
import Debug.Trace
import qualified Data.Map as Map

data ShadowState = ShadowState {
    shMemMap :: ShadowMap
    , shFileMap :: ShadowFileMap
} deriving (Show)

type ShadowMap = [(Model, Model)]


type ShadowFileMap = [(String, String)]

data ShadowState = ShadowState {
    shMemMap :: ShadowMap
    , shFileMap :: ShadowFileMap
} deriving (Show)
type ShadowStateT = StateT ShadowState ConState

mapModel f (Model n c a r t) =
  Model n (map (map (mapModel f)) c) (map f a) r t

mapModelM :: (Monad m) => (String -> m String) -> Model -> m Model
mapModelM f (Model n c a r t) =
do
  fa <- mapM f a
  fc <- mapM (mapM (mapModelM f)) c
  return $ Model n fc fa r t

shadowAttr v = do
  cuuid <- liftIO nextUUID
  let cid = show (fmap cuuid)
  unless (cid == Nothing) $ addCon (Concept (fromJust cid) v)
  return cid

shadow m = mapModelM ($a -> shadowAttr a >>= return . fromJust) m

deShadow s = mapModelM ($cid -> (getCon cid) >>= return . cVal . fromJust) s

createShadowModel :: String -> ShadowStateT ()
createShadowModel modelPath =
do
  model <- liftIO $ loadLazyModel modelPath
  shadowModel <- lift $ shadow model
  shMap <- get
  let shadowPath = (takeWhile (/='.') modelPath) ++ "Shadow.mod"
  let newShadowState = shMap {
    shMemMap = (model,shadowModel):(shMemMap shMap ),
    shFileMap = (modelPath,shadowPath):(shFileMap shMap )
  }
  put newShadowState
  liftIO $ storeLazyModel shadowModel shadowPath
  state <- get
  liftIO $ putStrLn (show state)

testShadow = runStateT (runStateT testScript (ShadowState [] [])) Map.empty

where testScript =
do
  createShadowModel "m.mod"
  m <- liftIO $ loadLazyModel "mShadow.mod"
  liftIO $ putStrLn $ "Shadow Model: " ++ (show m)
  dm <- lift $ deShadow m
  liftIO $ putStrLn $ "DeShadow Model: " ++ (show dm)

module Util where
import Data.Char

splitWith :: (Char -> Bool) -> String -> [String]
splitWith _ [] = []
splitWith p xs = let (ts, fs) = break p xs
                 in ts:splitWith p (tail' fs)
                 where tail' [] = []
tail' (x:xs) = xs

cutPrefix c = last' . (splitWith (\x->x==c))
            where last' [] = []
               last' xs = last xs

toBool s = case map toLower s of
            "true" -> True
            otherwise -> False

toInt s = case reads s::[(Int, String)] of
           [(n, [])] -> n
           otherwise -> 0
Appendix B

Abstract Syntax of QVT Operational Mappings
Figure B.1: Simplified Metamodel of QVT Operational Mappings
Figure B.2: Simplified Metamodel of Imperative OCL
Figure B.3: Simplified Metamodel of OCL
Appendix C

Big-Step Operational Semantics of QVT Operational Mappings

C.1 Big-Step Operational Semantics

In this section we present a formal semantics for Essential QVT-OM. The method chosen for denoting the semantics of the language is Big-Step Operational Semantics. Further information on specifying formal semantics of programming languages can be found in standard graduate level references on programming languages theory [79, 65, 59]. QVT-OM has a number of unique constructs that are not present in other languages. In that respect, a formal semantics for the language can be immensely useful to analyze profoundly the effect of these unfamiliar features. The semantics rules that deal with models and change operations use the formalism presented in Chapter 3.

The first set of rules spell out the meaning of literal values, constant collections, model element attribute, container and references access and basic arithmetic operations.
\[
\begin{align*}
\sigma; e \Downarrow v & \quad \sigma(x) = \bot \quad \sigma; e \Downarrow v \quad \sigma(x) \neq \bot \\
\langle \sigma, \text{var } x := e \rangle \Downarrow \langle \sigma[x \mapsto v], v \rangle & \quad \text{E -- Init} \\
\langle \sigma, x := e \rangle \Downarrow \langle \sigma[x \mapsto v], v \rangle & \quad \text{E -- Assign} \\
\sigma; e_{\text{init}} \Downarrow v & \\
\langle \sigma, \text{property } p : T = e_{\text{init}} \rangle \Downarrow \langle \sigma[p \mapsto v], _\_ \rangle & \quad \text{E -- Property}
\end{align*}
\]

\[
\begin{align*}
\sigma; e_{\text{cond}} \Downarrow \text{True} & \quad \sigma; e_{\text{then}} \Downarrow v_{\text{then}} \\
\sigma; \text{if } e_{\text{cond}} \text{ then } e_{\text{then}} \text{ else } e_{\text{else}} \Downarrow v_{\text{then}} & \quad \text{E -- If -- True} \\
\sigma; e_{\text{cond}} \Downarrow \text{False} & \quad \sigma; e_{\text{else}} \Downarrow v_{\text{else}} \\
\sigma; \text{if } e_{\text{cond}} \text{ then } e_{\text{then}} \text{ else } e_{\text{else}} \Downarrow v_{\text{else}} & \quad \text{E -- If -- False} \\
\sigma; \text{new } T() \Downarrow (\varnothing, \varnothing, \varnothing, T) & \quad \text{E -- New} \\
\langle \sigma_{i-1}, e_i \rangle \Downarrow \langle \sigma_i, v_i \rangle & \quad \text{E -- Object} \\
\sigma, \text{object } T \{ a_1 := e_1; \ldots; a_n := e_n; \} \Downarrow (\varnothing, \langle v_1, \ldots, v_n \rangle, \varnothing, T) & \quad \text{E -- Object} \\
\sigma; e_{\text{cond}} \Downarrow \text{False} & \\
\langle \sigma, \text{while } e_{\text{cond}} \text{ do } e_{\text{body}} \rangle \Downarrow \langle \sigma, _\_ \rangle & \quad \text{E -- While -- False} \\
\sigma; e_{\text{cond}} \Downarrow \text{True} \\
\langle \sigma, \varnothing, e_{\text{body}} \rangle \Downarrow \langle \sigma', \varnothing, _\_ \rangle \\
\langle \sigma', \varnothing, \text{while } e_{\text{cond}} \text{ do } e_{\text{body}} \rangle \Downarrow \langle \sigma'', _\_ \rangle \\
\langle \sigma, \varnothing, \text{while } e_{\text{cond}} \text{ do } e_{\text{body}} \rangle \Downarrow \langle \sigma'', _\_ \rangle & \quad \text{E -- While -- True} \\
\langle \sigma, \varnothing, \text{break}; e \rangle \Downarrow \langle \sigma, \varnothing, _\_ \rangle & \quad \text{E -- Break} \\
\langle \sigma, \varnothing, \text{while } e_{\text{cond}} \text{ do } e_{\text{body}} \rangle \Downarrow \langle \sigma, _\_ \rangle & \quad \text{E -- While -- Break} \\
\sigma; e_s \Downarrow \varnothing & \\
\langle \sigma, e_s \rightarrow \text{forEach}(x) \ e_{\text{body}} \rangle \Downarrow \langle \sigma, _\_ \rangle & \quad \text{E -- For -- Empty} \\
\langle \sigma, e_s \rangle \Downarrow \langle \sigma_0, (a_1, \ldots, a_n) \rangle & \langle \sigma_{i-1}[x \mapsto a_i], e_{\text{body}} \rangle \Downarrow \langle \sigma_i, _\_ \rangle |_{1 \leq i \leq n} \\
\langle \sigma, e_s \rightarrow \text{forEach}(x) \ e_{\text{body}} \rangle \Downarrow \langle \sigma_n, _\_ \rangle & \quad \text{E -- For}
\end{align*}
\]
\[
\langle \zeta, \text{helper } T_s :: f(a_1 : T_1, \ldots, a_n : T_n) : T_r \ e_{\text{body}} \rangle \\
\downarrow
\langle \zeta[f \mapsto \zeta(f)] \cup \{(T_s, T_r, \langle(a_i, T_i)\rangle, e_{\text{body}})\} \rangle, \_ \rangle
E - Helper
\]

\[
\langle \xi, \text{mapping } T_s :: m(a_1 : T_1, \ldots, a_n : T_n) : T_r e_i e_p e_c \rangle \\
\downarrow
\langle \xi[m \mapsto \xi(m)] \cup \{(T_s, T_r, \langle(a_i, T_i)\rangle, e_i, e_p, e_c)\} \rangle, \_ \rangle
E - Mapping
\]

\[
\langle (\sigma_p, \sigma), e \rangle \downarrow \nu
\langle (\sigma_p, \sigma), \downarrow, \text{return } e \rangle \downarrow \langle (\sigma_p, \sigma), \_\_\_ \rangle \nu \rangle
E - Return
\]

\[
\langle (\sigma_p, \sigma), \downarrow, e \rangle \downarrow \langle (\sigma'_p, \sigma'), \downarrow, \nu \rangle \downarrow \langle (\sigma''_p, \sigma), \downarrow, e' \rangle \downarrow \langle (\sigma''_p, \sigma), \downarrow, \nu' \rangle
E - Seq
\]

\[
\langle (\sigma_p, \sigma), \_\_\_, e \rangle \downarrow \langle (\sigma'_p, \sigma'), \_\_\_, \nu \rangle \downarrow \langle (\sigma'_p, \sigma'), \_\_\_, e' \rangle \downarrow \langle (\sigma'_p, \sigma'), \_\_\_, \nu' \rangle \rangle
E - Skip
\]

\[
\sigma; e_s \downarrow \nu_s \quad \sigma; a_i \downarrow \nu_{i \mid 1 \leq i \leq n} \quad \Gamma \vdash \nu_s : T_{v_s}
\exists T_s. (T_s, T_r, \langle(a_i, T_i)\rangle, e_{\text{body}}) \in \zeta(f) \land T_{v_s} \leftarrow T_s \land \forall T_b \ T_{v_s} \leftarrow T_b \Rightarrow T_s \leftarrow T_b
\langle \Gamma, \zeta, (\sigma_p, \sigma[\text{self } \mapsto \nu_s, a_i \mapsto \nu_{i \mid 1 \leq i \leq n}]), l, e_{\text{body}} \rangle \downarrow \langle (\sigma''_p, \sigma), \_\_\_, \nu_r \rangle
\]

\[
\langle \Gamma, \zeta, (\sigma_p, \sigma), e_s.f(e_1, \ldots, e_n) \rangle \downarrow \langle \zeta, (\sigma''_p, \sigma), \_\_\_, \nu_r \rangle \rangle \quad E - Call
\]

\[
\sigma; e_s \downarrow \nu_s \quad \sigma; a_i \downarrow \nu_{i \mid 1 \leq i \leq n} \quad \Gamma \vdash \nu_s : T_{v_s}
\exists T_s. (T_s, T_r, \langle(a_i, T_i)\rangle, e_i, e_p, e_c) \in \xi(m) \land T_{v_s} \leftarrow T_s \land \forall T_b \ T_{v_s} \leftarrow T_b \Rightarrow T_s \leftarrow T_b
\langle \Gamma, \xi, (\sigma_p, \sigma[\text{self } \mapsto \nu_s, a_i \mapsto \nu_{i \mid 1 \leq i \leq n}]), e_i \rangle \downarrow \langle (\sigma'_p, \sigma'), \_\_\_ \rangle 
\langle \Gamma, \xi, (\sigma'_p, \sigma')[\text{result } \mapsto (\emptyset, \emptyset, T_r)], e_p \rangle \downarrow \langle (\sigma''_p, \sigma'''), \_\_\_ \rangle
\langle \Gamma, \xi(\sigma''_p, \sigma'''), e_c \rangle \downarrow \langle (\sigma'''_p, \sigma''), \emptyset \rangle \downarrow \langle (\sigma'''_p, \sigma''), \_\_\_ \rangle \rangle \quad E - Map
\]

177
<table>
<thead>
<tr>
<th>Rule</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma; e \downarrow \text{True}$</td>
<td>$\sigma; e$.not() $\downarrow \text{False}$ $\sigma; e_1$.and(e_2) $\downarrow \text{False}$ $\sigma; e_1$.or(e_2) $\downarrow \text{True}$ $\sigma; e_1$.not() $\downarrow \text{True}$ $\sigma; e_1$.and(e_2) $\downarrow \beta$ $\sigma; e_1$.or(e_2) $\downarrow \beta$</td>
</tr>
<tr>
<td>$\sigma; e_1 \downarrow \text{False}$</td>
<td>$\sigma; e_1$.and(e_2) $\downarrow \beta$ $\sigma; e_1$.or(e_2) $\downarrow \beta$</td>
</tr>
<tr>
<td>$\sigma; e \downarrow \text{Or}$</td>
<td>$\sigma; e \downarrow \text{M}$ $\sigma; e$.objectsOfType(T) $\downarrow { m \in \sigma; e \downarrow \text{objectsOfType} }$ $\sigma; e \downarrow (C, A, R, T)$ $\sigma; e$.rootObjects() $\downarrow \cup C$ $\sigma; e \downarrow (C, A, R, T)$ $\sigma; e$.deepclone() $\downarrow (C, A, R, T)$</td>
</tr>
</tbody>
</table>
\[
\begin{array}{ll}
\sigma; e \downarrow \langle a_1, \ldots, a_n \rangle & \sigma; e \downarrow A \\
\sigma; e \rightarrow \mathit{asSet()} \downarrow \bigcup_{i=1}^{n} \{a_i\} & \text{E} \rightarrow \mathit{asSet} - \mathit{Ord} \\
\sigma; e \rightarrow \mathit{asSet()} \downarrow A & \sigma; e \rightarrow \mathit{asSet()} \downarrow \mathit{A} \text{ E} \rightarrow \mathit{asSet} \\
\sigma; e \downarrow \{a_1, \ldots, a_n\} \mid a_{i-1} \leq a_i & \sigma; e \rightarrow \mathit{asOrderedSet()} \downarrow \langle a_1, \ldots, a_n \rangle \text{ E} \rightarrow \mathit{asOrderedSet} \\
\sigma; e \rightarrow \mathit{asOrderedSet()} \downarrow \langle a_1, \ldots, a_n \rangle & \text{E} \rightarrow \mathit{asOrderedSet} \\
\frac{\sigma; e_1 \downarrow \phi \quad \sigma; e_2 \downarrow \mu \quad \phi(\mu) \neq \bot}{\sigma; e_1 \rightarrow \mathit{hasKey}(e_2) \downarrow \mathbf{True}} & \text{E} \rightarrow \mathit{hasKey} \\
\frac{\sigma; e_1 \downarrow \phi \quad \sigma; e_2 \downarrow \mu \quad \phi(\mu) = \bot}{\sigma; e_1 \rightarrow \mathit{hasKey}(e_2) \downarrow \mathbf{False}} & \text{E} \rightarrow \mathit{hasKey} - \mathit{not} \\
\frac{\sigma; e \downarrow \phi \quad \sigma; e_k \downarrow \mu \quad \sigma; e_v \downarrow \nu}{\sigma; e \rightarrow \mathit{put}(e_k, e_v) \downarrow \lambda x. \begin{cases} 
\nu & \text{x} = \mu \\
\phi(x) & \text{otherwise}
\end{cases} \text{ E} \rightarrow \mathit{put} \\
\sigma; e \downarrow \phi \quad \sigma; e_k \downarrow \mu & \sigma; e \downarrow \varnothing \\
\frac{\sigma; e \rightarrow \mathit{get}(e_k) \downarrow \phi(\mu) \text{ E} \rightarrow \mathit{get}}{\sigma; e \rightarrow \mathit{get}(e_k) \downarrow \bot \text{ E} \rightarrow \mathit{get} - \mathit{bot}} \\
\sigma; e \downarrow \varnothing & \sigma; e \downarrow \langle \nu_1, \ldots, \nu_n \rangle \\
\frac{\sigma; e \rightarrow \mathit{size()} \downarrow 0 \text{ E} \rightarrow \mathit{size} - \mathit{Empty}}{\sigma; e \rightarrow \mathit{size()} \downarrow n \text{ E} \rightarrow \mathit{size} - \mathit{Ord}} \\
\frac{\sigma; e \downarrow A}{\sigma; e \rightarrow \mathit{size()} \downarrow |A| \text{ E} \rightarrow \mathit{size} - \mathit{Set}} \\
\frac{\sigma; e \downarrow \langle \nu_1, \ldots, \nu_n \rangle}{\sigma; e \rightarrow \mathit{sum()} \downarrow \sum_{i=1}^{n} \nu_i \text{ E} \rightarrow \mathit{sum}} \\
\end{array}
\]
\[
\begin{array}{llll}
\sigma; e \downarrow A & \forall a \in A. \ \sigma[x \mapsto a]; e_p \downarrow False \\
\sigma; e \rightarrow \exists (x \in e_p) \downarrow False & E - \text{exists - not} \\
\sigma; e \downarrow A & \exists a \in A. \ \sigma[x \mapsto a]; e_p \downarrow True \\
\sigma; e \rightarrow \exists (x \in e_p) \downarrow True & E - \text{exists} \\
\sigma; e \downarrow \emptyset & \sigma; e \rightarrow \exists (x \in e_p) \downarrow False \quad E - \text{exists - Empty} \\
\sigma; e \downarrow A & \exists a \in A. \ \sigma[x \mapsto a]; e_p \downarrow False \\
\sigma; e \rightarrow \forall (x \in e_p) \downarrow False & E - \text{forall - not} \\
\sigma; e \downarrow A & \forall a \in A. \ \sigma[x \mapsto a]; e_p \downarrow True \\
\sigma; e \rightarrow \forall (x \in e_p) \downarrow True & E - \text{forall} \\
\sigma; e \downarrow \emptyset & \sigma; e \rightarrow \forall (x \in e_p) \downarrow True \quad E - \text{forall - Empty} \\
\sigma; e_s \downarrow \emptyset & \sigma; e_s \rightarrow \text{select}(x \mid e_p) \downarrow \emptyset \quad E - \text{select - Empty} \\
\sigma; e_s \downarrow \{v_1, \ldots, v_n\} & \sigma[x \mapsto v_i]; e_p \downarrow p_i \mid 1 \leq i \leq n \\
\sigma; e_s \rightarrow \text{select}(x \mid e_p) \downarrow \{v_i \mid p_i = \text{True}\} & E - \text{select} \\
\sigma; e_s \downarrow \emptyset & \sigma; e_s \rightarrow \text{collect}(x \mid e_c) \downarrow \emptyset \quad E - \text{collect - Empty} \\
\sigma; e_s \downarrow \langle v_1, \ldots, v_n \rangle & \sigma[x \mapsto v_i]; e \downarrow c_i \mid 1 \leq i \leq n \\
\sigma; e_s \rightarrow \text{collect}(x \mid e_c) \downarrow \langle c_1, \ldots, c_n \rangle & E - \text{collect}
\end{array}
\]
Appendix D

QvtMix Implementation

QVT Code

```java
import qvt.mix.util;

modeltype QVT uses qvtoperational::expressions('http://www.eclipse.org/QVT/1.0.0/Operational');
modeltype ImpOCL uses ImperativeOCL('http://www.eclipse.org/qvt/1.0/ImperativeOCL');
modeltype OCL uses ocl::utilities('http://www.eclipse.org/ocl/1.1.0/ocl');
modeltype OCLECORE uses ocl::ecore('http://www.eclipse.org/ocl/1.1.0/ocl');
modeltype OCLTYPE uses ocl::types('http://www.eclipse.org/ocl/1.1.0/OCL');
modeltype ECORE uses ecore('http://www.eclipse.org/emf/2002/ECore');
modeltype BTA uses 'http://qvt.bta/1.0';
modeltype BOOK uses 'http://book/1.0';

transformation QvtMix{
  in inputMetaModel : ECORE,
  in inputModel : ECORE,
  in qvt : QVT,
  in outputModel : ECORE,
  in outputMetaModel : ECORE,
  out residue : QVT
};
```

182
36 ///////////////////////////////////////////////////////////////////////////////

39 main() {
40

var trans := qvt.rootObjects()[OperationalTransformation]->asOrderedSet()->first();

41

trans.bta(bindingTimes, context);

45

var resQvt := trans.mix();

46

log(’import qvt.mix.util;’);

47

log(’modeltype ECORE uses ecore(\’http://www.eclipse.org/emf/2002/Ecore\’);’);

49

log(resQvt.print(0));

51

log(’

52 ///////////////////////// ModelUtils ///////////////////////////////////////////////////
53 intermediate class ObjectPath {
54

path : String;

55 }
57 helper EObject::path() : String {
58

var parents : OrderedSet(Integer) := OrderedSet{};

59

var containers : OrderedSet(Integer) := OrderedSet{};

60

var path := \’\’;

61

var obj := self;

62

var parentObj := obj.eContainer();

63

while (parentObj <> null) {

64

var container := obj.eContainingFeature().oclAsType(EReference);

65

var col := Sequence{};

66

getMultiFeature(parentObj, container.name, col);

67

var position := col->indexOf(obj);

68

path := \’/\’ + container.name + \’.\’ + position.repr() + path;

69

obj := parentObj;

70

parentObj := obj.eContainer();

71

};

72

if path = \’\’ then {

73

path := \’/\’;

74

}

75

endif;

76

return path;

77 }
79 helper Model::getObject(p : ObjectPath) : EObject {
80

var str := p.path;

81

var obj := self.rootObjects()->asOrderedSet()->first().oclAsType(EObject);

82

str := str.substringAfter(\’/\’);

83

while (str <> null) {

84

var segment := str.substringBefore(\’/\’);

85

var pos : String;

86

var cont : String;

87

if segment <> null then {

88

pos := segment.substringAfter(\’.\’);

89

cont := segment.substringBefore(\’.\’);

90

}

91

else {

92

pos := str.substringAfter(\’.\’);

93

cont := str.substringBefore(\’.\’);

94

}

95

endif;

96

obj := obj.getObject(cont, pos.asInteger());

183


97   str := str.substringAfter('/\');
98 
99   return obj;
100 }

101 helper ObjectPath::getObject(model : Model) : EObject {
102   return model.getObject(self);
103 }

104 helper EObject::getObject(cont : String, pos : Integer) : EObject {
105   var col := Sequence();
106   getMultiFeature(self, cont, col);
107   return col->at(pos).oclAsType(EObject);
108 }
109 ');
110 }

111 ///////////////////////////////////////////////////////// Path /////////////////////////////////////////////////////////

112 intermediate class ObjectPath {
113   path : String;
114   rootObject : EObject;
115 }

116 constructor ObjectPath::ObjectPath(root : EObject, p : String) {
117   path := p;
118   rootObject := root;
119 }

120 //////////////////////////////////////////////////////////
121 // ObjectPath::getObject
122 //////////////////////////////////////////////////////////

123 helper ObjectPath::getObject(model : Model) : EObject {
124   return model.getObject(self);
125 }

126 //////////////////////////////////////////////////////////
127 // EObject::path
128 //////////////////////////////////////////////////////////

129 helper EObject::path() : ObjectPath {
130   var parents : OrderedSet(Integer) := OrderedSet();
131   var containers : OrderedSet(Integer) := OrderedSet();
132   var path := '';
133   var obj := self;
134   var parentObj := obj.eContainer();
135   while (parentObj <> null) {
136     var container := obj.eContainingFeature().oclAsType(EReference);
137     var col := Sequence();
138     getMultiFeature(parentObj, container.name, col);
139     var position := col->indexOf(obj);
140     path := '/' + container.name + '.' + position.print() + path;
141     obj := parentObj;
142   }
158    parentObj := obj.eContainer();
159    );
160    if path = '' then {
161       path := '/';
162    }
163    endif;
164    return new ObjectPath(obj, path);
165 }

168 //////////////////////////////////////////////////////////////////////////////////////////
169 //
170 // Model::getObject
171 //
172 //////////////////////////////////////////////////////////////////////////////////////////

174 helper Model::getObject(path : ObjectPath) : EObject {
175    var str := path.path;
176    var obj := self.rootObjects()->asOrderedSet()->first().oclAsType(EObject);
177    str := str.substringAfter('/');
178    while (str <> null) {
179       var segment := str.substringBefore('/');
180       var pos : String;
181       var cont : String;
182       if segment <> null then {
183          pos := segment.substringAfter('.');
184          cont := segment.substringBefore('.');
185       }
186       else {
187          pos := str.substringAfter('.');
188          cont := str.substringBefore('.');
189       }
190    }
191    obj := obj.getObject(cont, pos.asInteger());
192    str := str.substringAfter('/');
193 }
194    return obj;
195 }

199 //////////////////////////////////////////////////////////////////////////////////////////
200 //
201 // ObjectPath::makeExp
202 //
203 //////////////////////////////////////////////////////////////////////////////////////////

205 helper ObjectPath::makeExp() : OCLExpression {
206    return self.path.makeExp();
207 }

210 //////////////////////////////////////////////////////////////////////////////////////////
211 //
212 // EObject::getObject
213 //
214 //////////////////////////////////////////////////////////////////////////////////////////

216 helper EObject::getObject(cont : String, pos : Integer) : EObject {
217    var col := Sequence();
218    getMultiFeature(self, cont, col);
return col->at(pos).oclAsType(EObject); }

//arium

helper memoizeVariable(variable : String, path : ObjectPath, val : OclAny) {
  var newpart := object DictLiteralPart {
    key := path.makeExp();
    value := val.makeExp();
  };
  var prop : EAttribute;
  if memoizeTables->hasKey(variable) then {
    prop := memoizeTables->get(variable);
    var dict := prop.eAnnotations.contents->first().oclAsType(DictLiteralExp);
    var parts := dict.part->asOrderedSet();
    parts += newpart;
    dict.part := parts->asSet();
    var annot := object EAnnotation {
      contents := OrderedSet{dict.oclAsType(EOBJECT)};
    };
    setMultiFeature(prop, 'eAnnotations', Sequence{annot});
  } else {
    var dict := object DictLiteralExp {
      part := Set{newpart};
    };
    prop := object EAttribute {
      name := variable;
      eType := object DictionaryType {
        keyType := object PrimitiveType {
          name := 'String';
        };
        elementType := newpart.value.eType.oclAsType(EOBJECT);
        name := 'Dict(String,' + newpart.value.typeName() + ')';
      };
      eAnnotations := OrderedSet{
        object EAnnotation {
          contents := OrderedSet{dict.oclAsType(EOBJECT)};
        }
      };
    };
    memoizeTables->put(variable, prop);
  }
}

//arium

helper getCacheType(cache : String) : String {
  if memoizeTables->hasKey(cache) then {
    var table := memoizeTables->get(cache);
return table.eTypeoclAsType(DictionaryType).name;
"
endif;
return null;
"

// EObject::getElementType

helper EObject::getElementType() : EObject {
  return null;
}

// EClassifier::getElementType

helper EClassifier::getElementType() : EObject {
  return self.oclAsType(EObject);
}

// CollectionType::getElementType

helper CollectionType::getElementType() : EObject {
  return self.elementType;
}

// DictionaryType::getElementType

helper DictionaryType::getElementType() : EObject {
  return self.elementType;
}

// OclAny::typeName

helper OclAny::typeName() : String {
  if self.oclIsKindOf(EObject) then {
    return self.oclAsType(EObject).typeName();
  }
341 }  
342 endif;  
343 return 'OclAny';  
344 }

347 //////////////////////////////////////////////////////////////////////////////////////////
348 //  
349 // Integer::typeName  
350 //  
351 //////////////////////////////////////////////////////////////////////////////////////////

353 helper Integer::typeName() : String {  
354 return 'Integer';  
355 }

358 //////////////////////////////////////////////////////////////////////////////////////////
359 //  
360 // IntegerLiteralExp::typeName  
361 //  
362 //////////////////////////////////////////////////////////////////////////////////////////

364 helper IntegerLiteralExp::typeName() : String {  
365 return 'Integer';  
366 }

370 //////////////////////////////////////////////////////////////////////////////////////////
371 //  
372 // RealLiteralExp::typeName  
373 //  
374 //////////////////////////////////////////////////////////////////////////////////////////

376 helper RealLiteralExp::typeName() : String {  
377 return 'Real';  
378 }

381 //////////////////////////////////////////////////////////////////////////////////////////
382 //  
383 // String::typeName  
384 //  
385 //////////////////////////////////////////////////////////////////////////////////////////

387 helper String::typeName() : String {  
388 return 'String';  
389 }

392 //////////////////////////////////////////////////////////////////////////////////////////
393 //  
394 // StringLiteralExp::typeName  
395 //  
396 //////////////////////////////////////////////////////////////////////////////////////////

398 helper StringLiteralExp::typeName() : String {  
399 return 'String';  
400 }
helper EObject::typeName() : String {
return self.eClass().name;
}

helper CollectionWrapper::typeName() :

  String {
var type : String := self.type;
if not self.collection()->isEmpty() then {
type := type + '(' + self.collection()->selectOne(true).typeName() + ')';
}
endif;
return type;
}

helper CollectionLiteralExp::typeName() :

  String {
var type : String := self.eType.name.substringBefore('(');
type := type + '(' + self.part->first().typeName() + ')';
return type;
}

helper CollectionItem::typeName() :

  String {
return self.item.typeName();
}

helper ObjectExp::typeName() :

  String {
return self.instantiatedClass.name;
}
intermediate class ArithmaticExp extends OCLExpression {
    left : OCLExpression;
    right : OCLExpression;
    op : String;
}

helper OCLExpression::toArithmaticExp() : OCLExpression {
    return self;
}

helper ocl::ecore::OperationCallExp::toArithmaticExp() : OCLExpression {
    var oper := self.referredOperation.oclAsType(EOperation);
    var lexp := self.source.oclAsType(OCLExpression);
    var rexp := self.argument->first().oclAsType(OCLExpression);
    var res := object ArithmaticExp {
        left := lexp.toArithmeticExp();
        right := rexp.toArithmeticExp();
    };
    switch {
        case (oper.name = '+') {
            res.op := 'ADD';
        }
        case (oper.name = '-') {
            res.op := 'SUB';
        }
        case (oper.name = '*') {
            res.op := 'MUL';
        }
        case (oper.name = '/') {
            res.op := 'DIV';
        }
        else {
            return self;
        }
    }
    return res;
}
helper ocl::expressions::OCLExpression::toOperationCall() : ocl::expressions::OCLExpression {
    return self;
}

helper ArithmaticExp::toOperationCall() : ocl::expressions::OCLExpression {
    var lexp := self.left.toOperationCall();
    var rexp := self.right.toOperationCall();
    var res := object OperationCallExp {
        source := lexp;
    };
    res.argument += rexp;
    var opName : String;
    switch {
        case (self.op = 'ADD') {opName := '+'}
        case (self.op = 'SUB') {opName := '-'}
        case (self.op = 'MUL') {opName := '*'}
        case (self.op = 'DIV') {opName := '/'}
    };
    res.referredOperation := object EOperation {
        name := opName;
    }.oclAsType(EObject);
    return res;
}

helper makeVariableExp(s : String) : VariableExp {
    return object VariableExp {
        referredVariable := object Variable {
            name := s;
        };
        name := referredVariable.oclAsType(Variable).name;
    };
}

}
helper OCLExpression::simplify() : OCLExpression {
  return self;
}

helper OCLExpression::isSame(r : OCLExpression) : Boolean {
  return self = r;
}

helper VariableExp::isSame(r : OCLExpression) : Boolean {
  return r.oclsKindOf(VariableExp) and
          self.referredVariable.oclAsType(Variable).name = r.oclAsType(VariableExp).referredVariable.oclAsType(Variable).name
  ;
}

helper IntegerLiteralExp::isSame(r : OCLExpression) : Boolean {
  return r.oclsKindOf(IntegerLiteralExp) and
          (self.integerSymbol = r.oclAsType(IntegerLiteralExp).integerSymbol)
  ;
}

helper ArithmaticExp::isSame(r : OCLExpression) : Boolean {
  if r.oclsKindOf(ArithmaticExp) then {
    var exp := r.oclAsType(ArithmaticExp);
    switch {
      case (self.op <> exp.op)
        {
          return false;
        }
      case (self.op = 'MUL' or self.op = 'ADD')
        {
          return
        }
      case (self.op = 'DIV' or self.op = 'SUB')
        {
          return
        }
      case (self.op = 'ADD' or self.op = 'SUB')
        {
          return
        }
    }
  }
(self.left.isSame(rexp.left)) and (self.right.isSame(rexp.right))

} or

{ (self.left.isSame(rexp.right)) and (self.right.isSame(rexp.left))
}

;

else

{ return (self.left.isSame(rexp.left)) and (self.right.isSame(rexp.right));
}

}

endif;

return false;

}

} // ArithmaticExp::print

//

(helper ArithmaticExp::print(tabs : Integer) : String {

var code := "";

switch {

case (self.op = 'ADD')

{

return ('' + self.left.print(0) + ' + ' + self.right.print(0) + '');
}

case (self.op = 'SUB')

{

return ('' + self.left.print(0) + ' - ' + self.right.print(0) + '');
}

case (self.op = 'MUL')

{

return ('' + self.left.print(0) + ' * ' + self.right.print(0) + '');
}

case (self.op = 'DIV')

{

return ('' + self.left.print(0) + ' / ' + self.right.print(0) + '');
}

case (self.op = 'NEG')

{

return '-' + self.left.print(0);
}

}

return code;

} // ArithmaticExp::simplify

(helper ArithmaticExp::simplify() : OCLExpression {

}
var lexp := self.left.simplify();
var rexp := self.right.simplify();
switch {
  case (self.op = 'ADD')
  {
    switch {
      case (lexp.oclIsKindOf(IntegerLiteralExp) and rexp.oclIsKindOf(IntegerLiteralExp))
      {
        return object IntegerLiteralExp {
          integerSymbol := lexp.oclAsType(IntegerLiteralExp).integerSymbol + rexp.oclAsType(IntegerLiteralExp).integerSymbol;
        };
      }
      case (rexp.oclIsKindOf(IntegerLiteralExp) and rexp.oclAsType(IntegerLiteralExp).integerSymbol = 0)
      {
        return lexp;
      }
      case (lexp.oclIsKindOf(IntegerLiteralExp))
      {
        return object ArithmeticExp {
          left := rexp;
          right := lexp;
          op := 'ADD';
        };
      }
      case (rexp.oclIsKindOf(IntegerLiteralExp) and rexp.oclAsType(IntegerLiteralExp).integerSymbol < 0)
      {
        var rval := rexp.oclAsType(IntegerLiteralExp).integerSymbol;
        return object ArithmeticExp {
          left := lexp;
          right := (-rval).makeExp();
          op := 'SUB';
        }.simplify();
      }
      case (lexp.oclIsKindOf(ArithmeticExp) and lexp.oclAsType(ArithmeticExp).op = 'ADD')
      {
        return object ArithmeticExp {
          left := lexp.oclAsType(ArithmeticExp).left.simplify();
          right := object ArithmeticExp {
            left := lexp.oclAsType(ArithmeticExp).right;
            right := rexp;
            op := 'ADD';
          }.simplify();
        }.simplify();
      }
      case (lexp.oclIsKindOf(ArithmeticExp) and lexp.oclAsType(ArithmeticExp).op = 'SUB')
      {
        return object ArithmeticExp {
          left := lexp.oclAsType(ArithmeticExp).left.simplify();
        };
      }
      case (rexp.oclIsKindOf(ArithmeticExp) and rexp.oclAsType(ArithmeticExp).op = 'ADD')
      {
        return object ArithmeticExp {
          left := rexp.oclAsType(ArithmeticExp).left.simplify();
          right := lexp;
          op := 'ADD';
        }.simplify();
      }
      case (rexp.oclIsKindOf(ArithmeticExp) and rexp.oclAsType(ArithmeticExp).op = 'SUB')
      {
        return object ArithmeticExp {
          left := rexp.oclAsType(ArithmeticExp).left.simplify();
          right := lexp;
          op := 'ADD';
        }.simplify();
      }
    }
  }
}
right := object ArithmeticExp {
  left := object ArithmeticExp {
    left := lexp.oclAsType(ArithmeticExp).right;
    op := 'NEG';
    }.simplify();
  right := rexp;
  op := 'ADD';
};
right := rexp;
op := 'ADD';
}

op := 'ADD';
}

case (lexp.oclIsKindOf(VariableExp) and rexp.oclIsKindOf(ArithmeticExp)) // x + (y + x)
{
  var r := rexp.oclAsType(ArithmeticExp);
  if lexp.isSame(r.left) then ( // x + (x + y)
    return object ArithmeticExp {
      left := object ArithmeticExp {
        left := 2.makeExp();
        right := r.left;
        op := 'MUL';
      };
      right := r.right;
      op := 'ADD';
    };
  }
  endif;
  if lexp.isSame(r.right) then ( // x + (y + x)
    return object ArithmeticExp {
      left := object ArithmeticExp {
        left := 2.makeExp();
        right := r.right;
        op := 'MUL';
      };
      right := r.left;
      op := 'ADD';
    };
  }
  endif;
  return object ArithmeticExp {
    left := 2.makeExp();
    right := lexp;
    op := 'MUL';
  };
}

case (lexp.oclIsKindOf(ArithmeticExp))
{
  return object ArithmeticExp {
    left := 2.makeExp();
    right := rexp;
    op := 'MUL';
  };
}

case ((lexp.oclIsKindOf(ArithmeticExp) and lexp.oclAsType(ArithmeticExp).op = 'MUL') or (rexp.oclIsKindOf(ArithmeticExp) and rexp.oclAsType(ArithmeticExp).op = 'MUL'))
{
  var l : ArithmeticExp;
  if lexp.oclIsKindOf(ArithmeticExp) then {
    l := lexp.oclAsType(ArithmeticExp);
  }
  else {
    l := object ArithmeticExp {
      left := 1.makeExp();
      right := lexp;
      op := 'MUL';
    };
  }
}
829
}
830
endif;
831
var r : ArithmaticExp;
832
if (rexp.oclIsKindOf(ArithmaticExp)) then {
833      r := rexp.oclAsType(ArithmaticExp);
834  }
835
else {
836      r := object ArithmaticExp {
837          left := l.makeExp();
838          right := rexp;
839          op := 'MUL';
840      };
841  }
842
endif;
843
var factor : OCLExpression;
844
var lpart : OCLExpression;
845
var rpart : OCLExpression;
847
if l.op = 'MUL' and r.op = 'MUL' then {
848      switch {
849          case {l.left.isSame(r.left)}
850          {
851              factor := l.left.simplify();
852              lpart := l.right.simplify();
853              rpart := r.right.simplify();
854          };
855          case {l.right.isSame(r.right)}
856          {
857              factor := l.right.simplify();
858              lpart := l.left.simplify();
859              rpart := r.left.simplify();
860          };
861          case {l.left.isSame(r.right)}
862          {
863              factor := l.left.simplify();
864              lpart := l.right.simplify();
865              rpart := r.left.simplify();
866          };
867          else
868          {
869              return self;
870          };
871      }
872  }
873
else {
874      return object ArithmaticExp {
875          left := object ArithmaticExp {
876              left := l;
877              right := r.left;
878          }.simplify();
879          op := 'ADD';
880          right := r.right;
881      }.simplify();
882  }
883
endif;
885
return object ArithmaticExp {
886      left := factor.simplify();
887      right := object ArithmaticExp {
888          left := lpart;
889          right := rpart;
890  };
891
}
op := 'ADD';
}.simplify();
op := 'MUL';
}.simplify();
}

```java
switch {
case (self.op = 'SUB')
{
switch {
case (lexp.oclIsKindOf(IntegerLiteralExp) and
rexp.oclIsKindOf(IntegerLiteralExp))
{
return object IntegerLiteralExp {
integerSymbol :=
lexp.oclAsType(IntegerLiteralExp).integerSymbol -
rexp.oclAsType(IntegerLiteralExp).integerSymbol
};
}
case (rexp.oclIsKindOf(IntegerLiteralExp) and
rexp.oclAsType(IntegerLiteralExp).integerSymbol = 0)
{
return lexp;
}
case (lexp.oclIsKindOf(IntegerLiteralExp) and
lexp.oclAsType(IntegerLiteralExp).integerSymbol = 0)
{
return object ArithmeticExp {
left := lexp;
right := null;
op := 'NEG';
}.simplify();
}
case (lexp.isSame(rexp))
{
return 0.makeExp();
}
case (rexp.oclIsKindOf(IntegerLiteralExp))
{
var rval := rexp.oclAsType(IntegerLiteralExp).integerSymbol;
if rval < 0 then {
return object ArithmeticExp {
left := lexp;
right := (-rval).makeExp();
op := 'ADD';
};
}
return object ArithmeticExp {
left := lexp;
right := rexp;
op := 'SUB';
};
}
}
case (self.op = 'MUL')
{
switch {
```
case {{\texttt{rexp.oclIsKindOf}({\texttt{IntegerLiteralExp}}) \textbf{and} \\
\texttt{rexp.oclAsType}({\texttt{IntegerLiteralExp}}).integerSymbol = 0} \textbf{or} \\
{{\texttt{lexp.oclIsKindOf}({\texttt{IntegerLiteralExp}}) \textbf{and} \\
\texttt{lexp.oclAsType}({\texttt{IntegerLiteralExp}}).integerSymbol = 0}}
  
  
  { 
    \texttt{return object} IntegerLiteralExp { 
      integerSymbol := 0;
      \}
  } 
  
  case {\texttt{rexp.oclIsKindOf}({\texttt{IntegerLiteralExp}}) \textbf{and} \\
  \texttt{rexp.oclAsType}({\texttt{IntegerLiteralExp}}).integerSymbol = 1}
  
  { 
    \texttt{return lexp;}
  } 
  
  case {\texttt{lexp.oclIsKindOf}({\texttt{IntegerLiteralExp}}) \textbf{and} \\
  \texttt{lexp.oclAsType}({\texttt{IntegerLiteralExp}}).integerSymbol = 1}
  
  { 
    \texttt{return rexp;}
  } 
  
  case {\texttt{lexp.oclIsKindOf}({\texttt{ArithmaticExp}}) \textbf{and} \\
  \texttt{lexp.oclAsType}({\texttt{ArithmaticExp}}).op = 'MUL'
  
  { 
    \texttt{return object} ArithmaticExp { 
      left := lexp;
      right := lexp;
      op := 'MUL';
      \}.simplify();
    } 
  } 
  
  case {\texttt{lexp.oclIsKindOf}({\texttt{VariableExp}}) \textbf{and} \\
  \texttt{rexp.oclIsKindOf}({\texttt{ArithmaticExp}})}
  
  { 
    \texttt{var l := object} ArithmaticExp { 
      left := lexp;
      right := rexp;
      op := 'MUL';
    }
  }
return object ArithmaticExp {
    left := l.simplify();
    right := r.right;
    op := 'MUL';
}.simplify();
}
} 

else {
    return l.simplify();
}
}
}

switch {
    case (self.op = 'DIV') {
        switch {
            case (rexp.oclIsKindOf(IntegerLiteralExp) and
                rexp.oclAsType(IntegerLiteralExp).integerSymbol = 0) {
                return null;
            } 
            case (lexp.oclIsKindOf(IntegerLiteralExp) and
                lexp.oclAsType(IntegerLiteralExp).integerSymbol = 0) {
                return lexp;
            } 
            case (lexp.isSame(rexp)) {
                return 1.makeExp();
            } 
            case (rexp.oclIsKindOf(IntegerLiteralExp) and
                lexp.oclIsKindOf(ArithmaticExp)) {
                var l := lexp.oclAsType(ArithmaticExp);
                var r := rexp;
                op := 'DIV';
                return object ArithmaticExp {
                    left := l.right;
                    right := object ArithmaticExp {
                        left := l.right;
                        right := r;
                        op := 'DIV';
                    }
                }.simplify();
            } 
            case (rexp.oclIsKindOf(ArithmaticExp)) {
                var l := rexp.oclAsType(ArithmaticExp);
                return object ArithmaticExp {
                    left := l.left;
                    right := object ArithmaticExp {
                        left := l.left;
                        right := r;
                        op := 'DIV';
                    }
                }.simplify();
            } 
            case (lexp.isSame(rexp)) {
                return 1.makeExp();
            } 
            case (lexp.oclIsKindOf(ArithmaticExp) and
                lexp.oclAsType(ArithmaticExp).op = 'MUL') {
                var l := lexp.oclAsType(ArithmaticExp);
                return object ArithmaticExp {
                    left := l.left;
                    right := object ArithmaticExp {
                        left := l.left;
                        right := r;
                        op := 'DIV';
                    }
                }.simplify();
            } 
            case (lexp.oclIsKindOf(ArithmaticExp)) {
                var l := lexp.oclAsType(ArithmaticExp);

switch {
    case (l.op == 'NEG') {
        return l.left.simplify();
    }
    case (l.op == 'ADD') {
        return object ArithmaticExp {
            left := object ArithmaticExp {
                left := l.left.simplify();
                op := 'NEG';
            };
            right := object ArithmaticExp {
                left := l.right.simplify();
                op := 'NEG';
            };
            op := 'SUB';
        }.simplify();
    };
    case (l.op == 'SUB') {
        return object ArithmaticExp {
            left := object ArithmaticExp {
                left := l.left.simplify();
                op := 'NEG';
            };
            right := object ArithmaticExp {
                left := l.right.simplify();
                op := 'NEG';
            };
            op := 'ADD';
        }.simplify();
    };
    return object ArithmaticExp {
        left := lexp;
        right := rexp;
        op := self.op
    };
}

property helpers : OrderedSet(Helper) = null;
property mappings : OrderedSet(MappingOperation) = null;
property properties : OrderedSet(EAttribute) = null;
property configProperties : OrderedSet(EAttribute) = null;
property usedModelTypes : OrderedSet(ModelType) = null;
property mappingCallLevel : Integer = 0;
intermediate class Environment {
  localEnv : Dict(String, Frame);
  parentEnv : Environment;
}

constructor Environment::Environment() {
  localEnv := Dict();
  parentEnv := null;
}

createEnvironment() : Environment {
  return new Environment();
}

constructor Environment::Environment(parent : Environment) {
  localEnv := Dict();
  parentEnv := parent;
}

createEnvironment(parent : Environment) : Environment {
  return new Environment(parent);
}

constructor Environment::Environment(name : String, val : OclAny, parent : Environment ) {
  parentEnv := parent;
  localEnv := Dict();
  localEnv-&gt;put(name, createFrame(val));
}

createEnvironment(name : String, val : OclAny, parent : Environment) : Environment {
  return new Environment(name, val, parent);
}

Environment::hasKey

Environment::hasKey
helper Environment::hasKey(name : String) : Boolean {
    if (self.localEnv->hasKey(name)) then {
        return true;
    }
    endif;
    return self.parentEnv.hasKey(name);
}

helper Environment::put(name : String, val : Frame) : Environment {
    if (not self.localEnv->hasKey(name)) and self.parentEnv.hasKey(name) then {
        self.parentEnv->put(name, val);
    } else {
        self.localEnv->put(name, val);
    }
    endif;
    return self;
}

helper Environment::get(name : String) : Frame{
    if self.localEnv->hasKey(name) then {
        return self.localEnv->get(name);
    } endif;
    return self.parentEnv.get(name);
}

helper getEnvironment(e : OclAny) : Environment {
    if (eoclIsTypeOf(e, Environment)) then {
        return eoclAsType(e, Environment);
    }
    endif;
    return null;
}
1256  // Environment::copy
1257  //
1258  ///////////////////////////////////////////////////////////////////////////////

1259  helper Environment::copy(): Environment {
1260    var env : Environment := new Environment();
1261    var local := self.localEnv;
1262    var goUp := true;
1263    while (goUp) {
1264      local->keys()->forEach(k) {
1265        env.put(k, local->get(k));
1266      };
1267      if (self.parentEnv <> null) then {
1268        env := object Environment {
1269          parentEnv := env;
1270        };
1271        local := self.parentEnv.localEnv;
1272      } else {
1273        break;
1274      }
1275    } endif;
1276  }
1277  return env;
1278 }

1279  intermediate class Function {};

1280  ///////////////////////////////////////////////////////////////////////////////
1281  //
1282  // Function::func
1283  //
1284  ///////////////////////////////////////////////////////////////////////////////

1285  helper Function::func(arg : OclAny) : OclAny {
1286    return null;
1287 }
1288
1289  intermediate class Eq extends Function {};

1290  ///////////////////////////////////////////////////////////////////////////////
1291  //
1292  // Eq::func
1293  //
1294  ///////////////////////////////////////////////////////////////////////////////

1295  helper Eq::func(arg0 : OclAny, arg1 : OclAny) : OclAny {
1296    return arg0 = arg1;
1297 }
1298
1299  ///////////////////////////////////////////////////////////////////////////////
1300  //
1301  // Function::func
1302  //
1303  ///////////////////////////////////////////////////////////////////////////////

1304  helper Function::func(arg0 : OclAny, arg1 : OclAny) : OclAny {
1305    return null;
1306 }
1307
1308  ///////////////////////////////////////////////////////////////////////////////
ImperativeIterateExp::makeCondition

helper ImperativeIterateExp::makeCondition(env : Environment) : Function {

    var opCall := self.condition.oclAsType(OperationCallExp);
    var opName := opCall.referredOperation.oclAsType(EOperation).name;
    var cond : Function := null;
    if opName = '=' then {
        cond := new Eq();
    }
    endif;
    return cond;
}

helper ImperativeIterateExp::makeCollector(env : Environment) : Function {

    return null;
}

intermediate class CollectionWrapper extends EObject {

    set : Set(OclAny);
    seq : Sequence(OclAny);
    ord : OrderedSet(OclAny);
    list : List(OclAny);
    bag : Bag(OclAny);
    type : String;
}

constructor CollectionWrapper::CollectionWrapper(c : Sequence(OclAny)) {
    type := "Sequence";
    seq := c;
}

constructor CollectionWrapper::CollectionWrapper(c : OrderedSet(OclAny)) {
    type := "OrderedSet";
    ord := c;
}

constructor CollectionWrapper::CollectionWrapper(c : Set(OclAny)) {
    type := "Set";
    set := c;
}

constructor CollectionWrapper::CollectionWrapper(c : List(OclAny)) {
    type := "List";
    list := c;
}
constructor CollectionWrapper::CollectionWrapper(c : Bag(oclAny)) {
    type := 'Bag';
    bag := c;
}

constructor CollectionWrapper::CollectionWrapper(c : Collection(oclAny)) {
    type := 'Collection';
}

// ///////////////////////////////////////////////////////////////////////////
// CollectionWrapper
// //////////////////////////////////////////////////////////////////////////

helper CollectionWrapper::collection() : Collection(oclAny) {
    switch {
    case (self.type = 'Bag')
        { return self.bag }
    case (self.type = 'List')
        { return self.list }
    case (self.type = 'OrderedSet')
        { return self.ord }
    case (self.type = 'Sequence')
        { return self.seq }
    case (self.type = 'Set')
        { return self.set }
    }
    return null;
}

// ///////////////////////////////////////////////////////////////////////////
// createCollectionWrapper
// //////////////////////////////////////////////////////////////////////////

helper createCollectionWrapper(c : Bag(oclAny)) : CollectionWrapper {
    return new CollectionWrapper(c);
}

// ///////////////////////////////////////////////////////////////////////////
// createCollectionWrapper
// //////////////////////////////////////////////////////////////////////////

helper createCollectionWrapper(c : Set(oclAny)) : CollectionWrapper {
    return new CollectionWrapper(c);
}
helper createCollectionWrapper(c : OrderedSet(OclAny)) : CollectionWrapper {
    return new CollectionWrapper(c);
}

// createCollectionWrapper

helper createCollectionWrapper(c : Sequence(OclAny)) : CollectionWrapper {
    return new CollectionWrapper(c);
}

// createCollectionWrapper

helper createCollectionWrapper(c : List(OclAny)) : CollectionWrapper {
    return new CollectionWrapper(c);
}

// createCollectionWrapper

helper getWrappedCollection(c : OclAny) : Collection(OclAny) {
    if (coclIsTypeOf(CollectionWrapper)) then {
        return coclAsType(CollectionWrapper).collection();
    }
    return null;
}

// getWrappedCollection

helper getCollectionWrapper(c : OclAny) : CollectionWrapper {
    if (coclIsTypeOf(CollectionWrapper)) then {
        return coclAsType(CollectionWrapper);
    }
    return null;
}

// getCollectionWrapper

helper setCollectionWrapper(c : OclAny) : CollectionWrapper {
    // setCollectionWrapper
}
helper setCollectionWrapper(inout c : EObject, col : OrderedSet(OclAny)) {
  if (c.oclIsTypeOf(CollectionWrapper)) then {
    c.oclAsType(CollectionWrapper).ord := col;
  }
  endif;
}

helper OclAny::asSet() : Set(OclAny) {
  if (self.oclIsKindOf(CollectionWrapper)) then {
    self.oclAsType(CollectionWrapper).asSet();
  }
  endif;
  return Set{self};
}

helper CollectionWrapper::asSet() : Set(OclAny) {
  switch {
    case (self.type = 'Bag')
      {return self.bag->asSet();}
    case (self.type = 'List')
      {
        var res := Sequence();
        self.list->forEach(element) {
          res->append(element);
        };
        return res->asSet();
      }
    case (self.type = 'OrderedSet')
      {return self.ord->asSet();}
    case (self.type = 'Sequence')
      {return self.seq->asSet();}
    case (self.type = 'Set')
      {return self.set}
  };
  return null;
}

helper CollectionWrapper::asOrderedSet() : Set(OclAny) {
  switch {
    case (self.type = 'Bag')
      {return self.bag->asSet();}
    case (self.type = 'List')
      {
        var res := Sequence();
        self.list->forEach(element) {
          res->append(element);
        };
        return res->asSet();
      }
    case (self.type = 'OrderedSet')
      {return self.ord->asSet();}
    case (self.type = 'Sequence')
      {return self.seq->asSet();}
    case (self.type = 'Set')
      {return self.set}
  };
  return null;
}
helper CollectionWrapper::asOrderedSet() : OrderedSet(OclAny) {
    switch {
    case (self.type = 'Bag') {
        return self.bag->asOrderedSet();
    case (self.type = 'List') {
        var res := OrderedSet();
        self.list->forEach(element) {
            res->append(element);
        }
        return res;
    case (self.type = 'OrderedSet') {
        return self.ord->asOrderedSet();
    case (self.type = 'Sequence') {
        return self.seq->asOrderedSet();
    case (self.type = 'Set') {
        return self.set->asOrderedSet();
    }
    return null;
    }
    return null;
    }
    }

    helper CollectionWrapper::asBag() : Bag(OclAny) {
    switch {
    case (self.type = 'Bag') {
        return self.bag;
    case (self.type = 'List') {
        var res := OrderedSet();
        self.list->forEach(element) {
            res->append(element);
        }
        return res->asBag();
    case (self.type = 'OrderedSet') {
        return self.ord->asBag();
    case (self.type = 'Sequence') {
        return self.seq->asBag();
    case (self.type = 'Set') {
        return self.set->asBag();
    }
    return null;
    }
    return null;
    }

    helper CollectionWrapper::asList() : List(OclAny) {
    switch {
    case (self.type = 'Bag') {
        return self.bag->asList();
    case (self.type = 'List') {
        return self.list->asList();
    case (self.type = 'OrderedSet') {
        return self.ord->asList();
    case (self.type = 'Sequence') {
        return self.seq->asList();
    case (self.type = 'Set') {
        return self.set->asList();
    }
    return null;
    }
    return null;
    }

    //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    // CollectionWrapper::asBag
    //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    // CollectionWrapper::asList
    //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
case (self.type = 'List')
    return self.list

case (self.type = 'OrderedSet')
    return self.ord->asList()

case (self.type = 'Sequence')
    return self.seq->asList()

case (self.type = 'Set')
    return self.set->asList()
};

return null;

helper CollectionWrapper::print() : String {
    var output : String = '[
    if (self.collection()->notEmpty()) then {
        var firstObj := self.collection()->any(true);
        output := output + firstObj.repr();
        //self.collection->
        self.collection()->forEach(e| e <> firstObj){
            output := output + ', ' + e.repr();
        }
    } endif;
    return output + ']';
}

intermediate class FrameFactory {
    static instance : FrameFactory;
};

helper createFrame(val : OclAny) : Frame {
    switch {
        case (val.oclIsKindOf(EObject))
            return new ObjectFrame(val.oclAsType(EObject));
        case (val.oclIsKindOf(Integer))
            return new IntegerFrame(val.oclAsType(Integer));
        case (val.oclIsKindOf(String))
            return new StringFrame(val.oclAsType(String));
        case (val.oclIsKindOf(Boolean))
            return new BooleanFrame(val.oclAsType(Boolean));
        else
            return new Frame(val);
    }
    return null;
}
helper updateFrame(frame : ObjectFrame, val : EObject) : Frame {
    return new ObjectFrame(val);
}

helper updateFrame(frame : IntegerFrame, val : Integer) : Frame {
    return new IntegerFrame(val);
}

intermediate class Frame {
    type : AnyType;
    scope : Integer;
    binding : String;
    anyValue : OclAny;
}

constructor Frame::Frame(val : OclAny) {
    anyValue := val;
}

helper Frame::value() : OclAny {
    return self.anyValue;
}

intermediate class ObjectFrame extends Frame{
    objValue : EObject;
}

constructor ObjectFrame::ObjectFrame(val : EObject) {
    objValue := val;
}

helper ObjectFrame::value() : OclAny {
    return self.objValue;
}
1745  intermediate class IntegerFrame extends Frame {
1746    intValue : Integer;
1747  }

1749  constructor IntegerFrame::IntegerFrame(val : Integer) {
1750    intValue := val;
1751  }

1754  ///////////////////////////////////////////////////////////////////////////
1755  // IntegerFrame::value
1756  ///////////////////////////////////////////////////////////////////////////

1760  helper IntegerFrame::value() : OclAny {
1761    return self.intValue;
1762  }

1764  intermediate class StringFrame extends Frame {
1765    strValue : String;
1766  }

1768  constructor StringFrame::StringFrame(val : String) {
1769    strValue := val;
1770  }

1773  ///////////////////////////////////////////////////////////////////////////
1774  // StringFrame::value
1775  ///////////////////////////////////////////////////////////////////////////

1779  helper StringFrame::value() : OclAny {
1780    return self.strValue;
1781  }

1783  intermediate class BooleanFrame extends Frame {
1784    boolValue : Boolean;
1785  }

1787  constructor BooleanFrame::BooleanFrame(val : Boolean) {
1788    boolValue := val;
1789  }

1792  ///////////////////////////////////////////////////////////////////////////
1793  // BooleanFrame::value
1794  ///////////////////////////////////////////////////////////////////////////

1798  helper BooleanFrame::value() : OclAny {
1799    return self.boolValue;
1800  }

1803  ///////////////////////////////////////////////////////////////////////////
1804  //
// OclAny::invoke

helper OclAny::invoke(op : EOperation, args : Sequence(OclAny)) : OclAny {
  var opName := op.name;
  if (not self.oclIsKindOf(CollectionWrapper)) then
    switch {
      case (opName = '=') { return self = args->first();}
      case (opName = '<>') { return self <> args->first();}
      case (opName = 'asSequence') { return new CollectionWrapper(self->asSequence());}
      case (opName = 'asSet') { return new CollectionWrapper(self->asSet());}
      case (opName = 'asBag') { return new CollectionWrapper(self->asBag());}
      case (opName = 'asOrderedSet') { return new CollectionWrapper(self->asOrderedSet());}
      case (opName = 'asList') { return new CollectionWrapper(self->asList());}
    }
  endif;
  var builtin := self.invoke(opName, args);
  if builtin = null then
    return nonBuiltinOperationInvoke(self, op, args)
  endif;
  return builtin;
}

// OclAny::invoke

helper OclAny::invoke(opName : String, args : Sequence(OclAny)) : OclAny {
  if self.oclIsKindOf(EOBJECT) then
    return self.oclAsType(EOBJECT).invoke(opName, args)
  endif;
  return null;
}

helper nonBuiltinOperationInvoke(source : OclAny, op : EOperation, args : Sequence(OclAny)) : OclAny {
  // look-up defined operations
  // evaluate the operation body for the source and args
  var srcObj := source.oclAsType(EOBJECT);
  var srcClass := srcObj.eClass();
  var res := srcObj.eInvoke(op, args->oclAsType(EOBJECT));
  return null;
}

// String::invoke

//
helper String::invoke(opName : String, args : Sequence(OclAny)) : OclAny {

switch {
    case (opName = '=')
        {return self = args->first().oclAsType(String)}
    case (opName = '<=>')
        {return self <=> args->first().oclAsType(String)}
    case (opName = 'size')
        {return self.size()}
    case (opName = 'concat')
        {return self.concat(args->first().oclAsType(String))}
    case (opName = 'substring')
        {return self.substring(args->first().oclAsType(String))}
    case (opName = 'toInteger')
        {return self.toInteger()}
    case (opName = 'toReal')
        {return self.toReal()}
    case (opName = 'toLower')
        {return self.toLower()}
    case (opName = 'toUpper')
        {return self.toUpper()}
    case (opName = '+')
        {return self + args->first().oclAsType(String)}
    case (opName = 'addSuffixNumber')
        {return self.addSuffixNumber()}
    case (opName = 'asBoolean')
        {return self.asBoolean()}
    case (opName = 'asFloat')
        {return self.asFloat()}
    case (opName = 'asInteger')
        {return self.asInteger()}
    case (opName = 'endsWith')
        {return self.endsWith(args->first().oclAsType(String))}
    case (opName = 'equalsIgnoreCase')
        {return self.equalsIgnoreCase(args->first().oclAsType(String))}
    case (opName = 'find')
        {return self.find(args->first().oclAsType(String))}
    case (opName = 'firstToUpper')
        {return self.firstToUpper()}
    case (opName = 'format')
        {
            log("format not supported yet");
            assert fatal (true);
        }
    case (opName = 'getStrCounter')
        {return self.getStrCounter(args->first().oclAsType(String))}
    case (opName = 'incrStrCounter')
        {return self.incrStrCounter(args->first().oclAsType(String))}
    case (opName = 'indexOf')
        {return self.indexOf(args->first().oclAsType(String))}
    case (opName = 'isQuoted')
        {return self.isQuoted(args->first().oclAsType(String))}
    case (opName = 'lastToUpper')
        {return self.lastToUpper()}
    case (opName = 'length')
        {return self.length()}
    case (opName = 'match')
        {return self.match(args->first().oclAsType(String))}
    case (opName = 'matchBoolean')
        {return self.matchBoolean(args->first().oclAsType(Boolean))}
    case (opName = 'matchFloat')
        {return self.matchFloat(args->first().oclAsType(Float))}
    }
```java
    {return self.matchFloat(args->first().oclAsType(Real))}
  case (opName = 'matchIdentifier')
    {return self.matchIdentifier(args->first().oclAsType(String))}
  case (opName = 'matchInteger')
    {return self.matchInteger(args->first().oclAsType(Integer))}
  case (opName = 'normalizeSpace')
    {return self.normalizeSpace()}
  case (opName = 'replace')
    {return self.replace(args->at(1).oclAsType(String), args->at(2).oclAsType(String))}
  case (opName = 'restartAllStrCounter')
    {return self.restartAllStrCounter()}
  case (opName = 'rfind')
    {return self.rfind(args->first().oclAsType(String))}
  case (opName = 'startsWith')
    {return self.startsWith(args->first().oclAsType(String))}
  case (opName = 'startStrCounter')
    {return self.startStrCounter(args->first().oclAsType(String))}
  case (opName = 'substringAfter')
    {return self.substringAfter(args->first().oclAsType(String))}
  case (opName = 'substringBefore')
    {return self.substringBefore(args->first().oclAsType(String))}
  case (opName = 'quotify')
    {return self.quotify(args->first().oclAsType(String))}
  case (opName = 'trim')
    {return self.trim()}
  case (opName = 'unquotify')
    {return self.unquotify(args->first().oclAsType(String))}
  else
  {
  }
}
  return null;

  helper Integer::invoke(opName : String, args : Sequence(OclAny)) : OclAny {
    switch {
      case (opName = '<')
        {return self < args->first().oclAsType(Integer)}
      case (opName = '>')
        {return self > args->first().oclAsType(Integer)}
      case (opName = '=')
        {return self == args->first().oclAsType(Integer)}
      case (opName = '<=')
        {return self <= args->first().oclAsType(Integer)}
      case (opName = '>=')
        {return self >= args->first().oclAsType(Integer)}
      case (opName = '==')
        {return self == args->first().oclAsType(Integer)}
      case (opName = '>=')
        {return self >= args->first().oclAsType(Integer)}
      case (opName = '<=')
        {return self <= args->first().oclAsType(Integer)}
    }
```

214
if args->size() == 0 then
    return (-self);
else
    return self args->first().oclAsType(Integer);
endif;

return null;

helper Boolean::invoke(opName : String, args : Sequence(OclAny)) : OclAny {
    switch {
        case (opName == '=')
            {return self args->first().oclAsType(Boolean)}
        case (opName == '<> ')
            {return self args->first().oclAsType(Boolean)}
        case (opName == 'not ')
            {return self.not()}
        case (opName == 'and ')
            {return self args->first().oclAsType(Boolean)}
        case (opName == 'or ')
            {return self or args->first().oclAsType(Boolean)}
        case (opName == 'implies ')
            {return self implies args->first().oclAsType(Boolean)}
        case (opName == 'xor ')
            {return self xor args->first().oclAsType(Boolean)}
        else
            {
            }
    }
return null;
helper Real::invoke(opName : String, args : Sequence(OclAny)) : OclAny {
    switch {
        case (opName = '=') {
            return self = args->first().oclAsType(Real)
        }
        case (opName = '<') {
            return self < args->first().oclAsType(Real)
        }
        case (opName = '>') {
            return self > args->first().oclAsType(Real)
        }
        case (opName = '<=') {
            return self <= args->first().oclAsType(Real)
        }
        case (opName = '>=') {
            return self >= args->first().oclAsType(Real)
        }
        case (opName = '+') {
            return self + args->first().oclAsType(Real)
        }
        case (opName = '-') {
            return self - args->first().oclAsType(Real)
        }
        case (opName = '*') {
            return self * args->first().oclAsType(Real)
        }
        case (opName = '/') {
            return self / args->first().oclAsType(Real)
        }
        case (opName = 'max') {
            return self.max(args->first().oclAsType(Real))
        }
        case (opName = 'min') {
            return self.min(args->first().oclAsType(Real))
        }
        case (opName = 'abs') {
            return self.abs()
        }
        case (opName = 'floor') {
            return self.floor()
        }
        case (opName = 'round') {
            return self.round()
        }
        case (opName = 'toString') {
            return self.toString()
        }
        case (opName = 'toString') {
            return self.toString()
        }
        default {
            return null;
        }
    }
}

helper Model::invoke(opName : String, args : Sequence(OclAny)) : OclAny {
    switch {
        case (opName = 'copy') {
            return self.copy()
        }
        case (opName = 'createEmptyModel') {
            return self.createEmptyModel()
        }
        case (opName = 'objects') {
            return new CollectionWrapper(self.objects())
        }
        default {
            return null;
        }
    }
}
case (opName = 'objectsOfType')
{
  var arg := args->first().oclAsType(EClassifier);
  var res := self.objects()->oclAsType(EObject)->xselect(o|o.eClass().name = arg.name)->asOrderedSet();
  return new CollectionWrapper(res);
}

case (opName = 'removeElement')
{
  return self.removeElement(args->first().oclAsType(Element))
}

case (opName = 'rootObjects')
{
  return new CollectionWrapper(self.rootObjects())
}

case (opName = '=')
{
  return self = args->first()()
}

case (opName = '<>')
{
  return self <> args->first()()
}

};
return null;

/////////////////////////////////////////////////////////////////////////////////
//
// EObject::invoke
//
/////////////////////////////////////////////////////////////////////////////////

helper ecore::EObject::invoke(opName : String, args : Sequence(OclAny)) : OclAny {
  switch {
  case (opName = 'oclIsKindOf')
  {
    var typeArg := args->first().oclAsType(EClass);
    return typeArg.name = self.eClass().name;
  }
  case (opName = 'oclAsType')
  {
    if typeArg.name = self.eClass().name then
      return self
    endif;
    return null;
  }
  }

  return null;
}

/////////////////////////////////////////////////////////////////////////////////
//
// CollectionWrapper::invoke
//
/////////////////////////////////////////////////////////////////////////////////

helper CollectionWrapper::invoke(opName : String, args : Sequence(OclAny)) : OclAny {
  var c := self.collection();
  switch {
  case (opName = 'count') { return c->count(args->first()); } 
  case (opName = 'excludes') { return c->excludes(args->first()); }
  case (opName = 'excludesAll')
  {
    var a := args->first()->asSequence();
  }
return c->excludesAll(a);
}
case (opName = 'includes') {return c->includes(args->first());}
case (opName = 'includesAll') {return c->includesAll(args->first())->asSequence();}
case (opName = 'isEmpty') {return c->isEmpty();}
case (opName = 'notEmpty') {return c->notEmpty();}
case (opName = 'product') {
    return new CollectionWrapper(c->product(args->first()->asSequence()));
}
case (opName = 'sum') {return c->sum();}
case (opName = 'size') {return c->size();}
else {
    switch {
    case (self.type = 'Bag') {
        return bagInvoke(self.bag, opName, args);
    }
    case (self.type = 'Set') {
        return setInvoke(self.set, opName, args);
    }
    case (self.type = 'OrderedSet') {
        return ordInvoke(self.ord, opName, args);
    }
    case (self.type = 'Sequence') {
        return seqInvoke(self.seq, opName, args);
    }
    case (self.type = 'List') {
        return listInvoke(self.list, opName, args);
    }
    }
}

helper listInvoke(list : List<OclAny>, opName : String, args : Sequence<OclAny>) : OclAny {
    switch {
    case (opName = 'append') {
        list->append(args->first());
        return new CollectionWrapper(list);
    }
    case (opName = 'prepend') {
        list->prepend(args->first());
        return new CollectionWrapper(list);
    }
    case (opName = 'insertAt') {
        list->insertAt(}
args->first(),
    args->at(2).oclAsType(Integer)
);  
    return new CollectionWrapper(list);
  }
  }
  
  case (opName = 'joinfields')
  {
    list->joinfields(
    args->first().oclAsType(String),
    args->at(2).oclAsType(String),
    args->at(3).oclAsType(String)
    );
  }
  
  case (opName = 'xselect' or opName = 'select')
  {
    var expr := args->at(2).oclAsType(ImperativeIterateExp);
    var env := args->first().oclAsType(Environment);
    var iteratorname := expr.iterator->first().oclAsType(Variable).name;
    var col :=
    list->xselect(i_|
    expr.condition.eval(
    new Environment(iteratorname, i_, env)
    ).oclAsType(Boolean)
    );
    var resList : List(OclAny) := List();
    col->forEach(e) {
      resList->append(e);
    }
    return new CollectionWrapper(resList);
  }
  }
  
  case (opName = '=' ) {
    return seq = args->first()->asSequence();
  }
  
  case (opName = '<>') {
    return seq <> args->first()->asSequence();
  }
  
  case (opName = 'union') {
    return new CollectionWrapper(seq->union(args->first()->asSequence()));
  }
  
  case (opName = 'append') {
    return new CollectionWrapper(seq->append(args->first()));
  }
  
  case (opName = 'prepend') {
    return new CollectionWrapper(seq->prepend(args->first()));
  }
  
  case (opName = 'insertAt')
  {
    return new CollectionWrapper(
    seq->insertAt(
    args->first().oclAsType(Integer),
    args->at(2)
    )
    );
  }
  
  case (opName = 'subSequence')
  {
    return new CollectionWrapper(
seq->subSequence{
    args->first().oclAsType(Integer),
    args->at(2).oclAsType(Integer)
}.

case (opName = 'at') { return seq->at(args->first().oclAsType(Integer));}
case (opName = 'indexOf') { return seq->indexOf(args->first());}
case (opName = 'including') { return new CollectionWrapper(seq->includes(args->first()));}
case (opName = 'excluding') { return new CollectionWrapper(seq->excluding(args->first()));}
case (opName = 'first') {
    var res := seq->first();
    return res;
}
case (opName = 'last') {
    return seq->last();
}
case (opName = 'flatten') {
    return new CollectionWrapper(seq->flatten());
}
case (opName = 'asSequence') {
    return new CollectionWrapper(seq);
}
case (opName = 'asSet') {
    return new CollectionWrapper(seq->asSet());
}
case (opName = 'asBag') {
    return new CollectionWrapper(seq->asBag());
}
case (opName = 'asOrderedSet') {
    return new CollectionWrapper(seq->asOrderedSet());
}
case (opName = 'asList') {
    return new CollectionWrapper(seq->asList());
}
case (opName = 'xselect' or opName = 'select') {
    var expr := args->at(2).oclAsType(ImperativeIterateExp);
    var env := args->first().oclAsType(Environment);
    var iteratorname := expr.iterator->first().oclAsType(Variable).name;
    return new CollectionWrapper(
        seq->xselect(i_|
            expr.condition.eval(new Environment(iteratorname, i_, env)).oclAsType(Boolean))
    )
}

return null;

(helper ordInvoke(ord : OrderedSet(OclAny), opName : String, args : Sequence(OclAny)) : OclAny { 
    switch {
        case (opName = '=') {
            return ord = args->first() asOrderedSet();
        }
        case (opName = '<>') {
            return ord <> args->first() asOrderedSet();
        }
        case (opName = 'union') {
            return new CollectionWrapper(ord->union(args->first()) asOrderedSet());
        }
        case (opName = 'append') {
            return new CollectionWrapper(ord->append(args->first()));
        }
        case (opName = 'prepend') {
            return new CollectionWrapper(ord->prepend(args->first()));
        }
        case (opName = 'insertAt') {
            return new CollectionWrapper(
                ord->insertAt(
                    args->first().oclAsType(Integer),
                    args->at(2)
                )
            )
        }
    }
})

return null;
case (opName = 'subOrderedSet')
{
  return new CollectionWrapper(
    ord->subOrderedSet(
      args->first().oclAsType(Integer),
      args->at(2).oclAsType(Integer)
    )
  );
}

case (opName = 'at')
{
  return ord->at(args->first().oclAsType(Integer));
}

case (opName = 'including')
{
  return new CollectionWrapper(ord->including(args->first()));
}

case (opName = 'intersection')
{
  return new CollectionWrapper(ord->intersection(args->first()->asSet()));
}

case (opName = 'excluding')
{
  return new CollectionWrapper(ord->excluding(args->first()));
}

case (opName = '-')
{
  return new CollectionWrapper(ord - args->first().oclAsType(Integer));
}

case (opName = 'union')
{
  return new CollectionWrapper(ord->union(args->first()->asSet()));
}

case (opName = 'intersection')
{
  return new CollectionWrapper(ord->intersection(args->first()->asSet()));
}

case (opName = 'symmetricDifference')
{
  return new CollectionWrapper(ord->symmetricDifference(args->first()->asSet()));
}

case (opName = 'first')
{
  return ord->first();
}

case (opName = 'last')
{
  return ord->last();
}

case (opName = 'flatten')
{
  return new CollectionWrapper(ord->flatten());
}

case (opName = 'asSequence')
{
  return new CollectionWrapper(ord->asSequence());
}

case (opName = 'asSet')
{
  return new CollectionWrapper(ord->asSet());
}

case (opName = 'asBag')
{
  return new CollectionWrapper(ord->asBag());
}

case (opName = 'asOrderedSet')
{
  return new CollectionWrapper(ord);
}

case (opName = 'asList')
{
  return new CollectionWrapper(ord->asList());
}

case (opName = 'xselect' or opName = 'select')
{
  var expr := args->at(2).oclAsType(ImperativeIterateExp);
  var env := args->first().oclAsType(Environment);
  var iteratorname := expr.iterator->first().oclAsType(Variable).name;
  return new CollectionWrapper(
    ord->xselect(i_|
      expr.condition.eval(
        new Environment(iteratorname, i_, env)
      ).oclAsType(Boolean)
    )
  );
}

return null;

helper setInvoke(set : Set<OclAny>, opName : String, args : Sequence<OclAny>) : OclAny {
  var firstArg : Set<OclAny>;
  if args->first().oclIsKindOf(CollectionWrapper) then {
    firstArg := args->first()->asSet();
  }
  else {
    firstArg := args->first()->asSet();
  }

  // setInvoke
case (opName = '=' ) { return set = firstArg; }
case (opName = '<=' ) { return set <= firstArg; }
case (opName = 'union' ) { return new CollectionWrapper(set->union(firstArg)); }
case (opName = 'intersection' ) { return new CollectionWrapper(set->intersection(firstArg)); }
case (opName = '-' ) { return new CollectionWrapper(set - firstArg); }
case (opName = 'excluding' ) { return new CollectionWrapper(set->excluding(args->first())); }
case (opName = 'symmetricDifference' ) { return new CollectionWrapper(set->symmetricDifference(firstArg)); }
case (opName = 'flatten' ) { return new CollectionWrapper(set->flatten()); }
case (opName = 'asSequence' ) { return new CollectionWrapper(set->asSequence()); }
case (opName = 'asSet' ) { return new CollectionWrapper(set); }
case (opName = 'asBag' ) { return new CollectionWrapper(set->asBag()); }
case (opName = 'asOrderedSet' ) { return new CollectionWrapper(set->asOrderedSet()); }
case (opName = 'asList' ) { return new CollectionWrapper(set->asList()); }
case (opName = 'xselect' or opName = 'select' ) {
var expr := args->at(2).oclAsType(ImperativeIterateExp);
var env := args->first().oclAsType(Environment);
var iteratorname := expr.iterator->first().oclAsType(Variable).name;
return new CollectionWrapper(set->xselect(iteratorname, i_, env),
oclAsType(Boolean));
}
return null;
}

////////////////////////////////////////////////////////////////////////////////////////////
// bagInvoke
////////////////////////////////////////////////////////////////////////////////////////////

helper bagInvoke(bag : Bag(OclAny), opName : String, args : Sequence(OclAny)) : OclAny {
switch {
case (opName = '=' ) { return bag = args->first()->asBag(); }
case (opName = '<=' ) { return bag <= args->first()->asBag(); }
case (opName = 'union' ) {
return new CollectionWrapper(bag->union,
args->first())->asBag();
}
case (opName = 'intersection' ) { return new CollectionWrapper(bag->intersection(args->first())->asBag()); }
case (opName = 'excluding' ) { return bag->excluding(args->first()); }
case (opName = 'symmetricDifference' ) { return new CollectionWrapper(bag->symmetricDifference()); }
case (opName = 'flatten' ) { return new CollectionWrapper(bag->flatten()); }
case (opName = 'asSequence' ) { return new CollectionWrapper(bag->asSequence()); }
case (opName = 'asSet' ) { return new CollectionWrapper(bag->asSet()); }
}
2475 case (opName = 'asBag') {return new CollectionWrapper(bag);}
2476 case (opName = 'asOrderedSet') {return new CollectionWrapper(bag->asOrderedSet());}
2477 case (opName = 'asList') {return new CollectionWrapper(bag->asList());}
2478 case (opName = 'xselect' or opName = 'select') {
2479     var expr := args->at(2).oclAsType(ImperativeIterateExp);
2480     var env := args->first().oclAsType(Environment);
2481     var iteratorname := expr.iterator->first().oclAsType(Variable).name;
2482     return new CollectionWrapper(bag->xselect(i_| expr.condition.eval(new Environment(iteratorname, i_, env)).oclAsType(Boolean)));
2483 }
2484 }
2485 return null;
2486 }
2487 //////////////////////////////////////////////////////////////////////////////
2488 //
2489 // OclAny::print
2490 //
2491 //////////////////////////////////////////////////////////////////////////////
2492 helper OclAny::print() : String {
2493     return self.repr();
2494 }
2495 //////////////////////////////////////////////////////////////////////////////
2496 //
2497 // OCLExpression::lval
2498 //
2499 //////////////////////////////////////////////////////////////////////////////
2500 //
2501 //////////////////////////////////////////////////////////////////////////////
2502 helper ocl::ecore::OCLExpression::lval(env : Environment) : OclAny {
2503     return null;
2504 }
2505 //////////////////////////////////////////////////////////////////////////////
2506 //
2507 // OCLExpression::lval
2508 //
2509 //////////////////////////////////////////////////////////////////////////////
2510 //
2511 //////////////////////////////////////////////////////////////////////////////
2512 helper ocl::expressions::OCLExpression::lval(env : Environment) : OclAny {
2513     return null;
2514 }
2515 //////////////////////////////////////////////////////////////////////////////
2516 //
2517 // OCLExpression::lval
2518 //
2519 //////////////////////////////////////////////////////////////////////////////
2520 //
2521 //////////////////////////////////////////////////////////////////////////////
2522 //
2523 // OCLExpression::lval
2524 //
2525 //////////////////////////////////////////////////////////////////////////////
2526 //
2527 // OCLExpression::lval
2528 //
2529 //////////////////////////////////////////////////////////////////////////////
2530 //
2531 //////////////////////////////////////////////////////////////////////////////
2532 //
2533 //
helper VariableExp::lval(env : Environment) : OclAny {
  return self.name;
}

helper PropertyCallExp::lval(env : Environment) : OclAny {
  return self.source.lval(env);
}

helper OperationalTransformation::eval(env : Environment) : OclAny {
  return self.entry.eval(env);
}

helper EntryOperation::eval(env : Environment) : OclAny {
  return self.body.eval(env);
}

helper OperationBody::eval(env : Environment) : OclAny {
  var lastVal : OclAny;
  var retVal : OclAny;
  var retEnv := new Environment('%__return', null, env);
  return self.body.eval(env);
}
self.content->forEach(statement) {
  lastVal := statement.eval(retEnv);
  retVal := retEnv.get('$_return').value();
  if (retVal <> null) then {
    return retVal;
  }
}
return lastVal;

helper ocl::expressions::OCLExpression::eval(env : Environment) : OclAny {
return null;
}

helper ReturnExp::eval(env : Environment) : OclAny {
var retVal := self.value.eval(env);
env.put('$_return', createFrame(retVal));
return retVal;
}

helper OclAny::toEObject() : EObject {
return null;
}

helper String::toEObject() : EObject {
return self.oclAsType(EOBject);
}

helper Real::toEObject
helper Real::toEObject() : EObject {
  return selfoclAsType(EObject);
}

helper Integer::toEObject() : EObject {
  return selfoclAsType(EObject);
}

helper Boolean::toEObject() : EObject {
  return selfoclAsType(EObject);
}

helper EObject::toEObject() : EObject {
  return self;
}

helper ocl::ecore::OperationCallExp::eval(env : Environment): OclAny {
  var src := self.source.eval(env);
  if self.referredOperationoclIsKindOf(Helper) then {
    var helpOp := self.referredOperationoclAsType(Helper);
    var context : Environment;
    if srcoclIsInvalid() then
      context := new Environment(env)
    else
      context := new Environment(ulong"self", src, env)
    endif;
    var i := 1;
    while (i <= self.argument->size()) {
      
      
    }
```plaintext
2719  var arg := self.argument->at(i).eval(env);
2720  var argName := helpOp.eParameters->at(i).name;
2721  context.put(argName, createFrame(arg));
2722  i := i + 1;
2723 }
2724  return helpOp.body.eval(context);
2725 }
2726 endif;
2727 var args := self.argument->eval(env);
2728 var res := src.invoke(self.referredOperation.oclAsType(EOperation), args);
2729 return res;
2730 }
2731 }

2734 /////////////////////////////////////////////////////////////////////////////////
2735 //
2736 // MappingBody::eval
2737 //
2738 /////////////////////////////////////////////////////////////////////////////////

2740 helper MappingBody::eval(env : Environment) : OclAny {
2741  var initPart := self.initSection.eval(env);
2742  var contentPart := self.content->eval(env);
2743  var endPart := self.endSection.eval(env);
2744  var variablePart := self.variable;
2745  return contentPart->first();
2746 }

2749 /////////////////////////////////////////////////////////////////////////////////
2750 //
2751 // MappingParameter::eval
2752 //
2753 /////////////////////////////////////////////////////////////////////////////////

2755 helper MappingParameter::eval(env : Environment) : OclAny {
2756  var init_ := self.initExpression.eval(env);
2757  return null;
2758 }

2761 /////////////////////////////////////////////////////////////////////////////////
2762 //
2763 // MappingOperation::eval
2764 //
2765 /////////////////////////////////////////////////////////////////////////////////

2767 helper MappingOperation::eval(env : Environment) : OclAny {
2768  var whenPart := self._when.eval(env)->last().oclAsType(Boolean);
2769  if whenPart = false then {
2770    return null;
2771  }
2772  endif;
2773  var resultVar := self.result.name;
2774  var resultType := self.result.eType;
2775  var resFrame := new ObjectFrame(null);
2776  var resEnv := new Environment(env);
2777  resEnv.localEnv->put('result', resFrame);
2778  var contextPart := self.context.eval(env);
```

227
```java
//
// MappingCallExp::eval
//

helper MappingCallExp::eval(env : Environment) : OclAny {
  mappingCallLevel := mappingCallLevel + 1;
  var src := self.source.eval(env);
  var type := self.eType.name;
  var op := self.referredOperation.oclAsType(MappingOperation);
  var args := self.argument->eval(env).oclAsType(CollectionWrapper).collection();
  var newEnv := new Environment('self', src, env);

  var i := 1;
  while (i <= self.argument->size()) {
    var arg := self.argument->at(i).eval(env);
    var argName := op.eParameters->at(i).name;
    newEnv.put(argName, createFrame(arg));
    i := i + 1;
  }

  var res := op.eval(newEnv);
  mappingCallLevel := mappingCallLevel - 1;
  if mappingCallLevel = 0 and res.oclIsKindOf(EObject) then {
    var resObj := res.oclAsType(EObject);
    resObj.map writeObject();
  }
  endif;
  return res;
}
```

```java
//
// BlockExp::eval
//

helper BlockExp::eval(env : Environment) : OclAny {
  self.body->eval(new Environment(env));
  return null;
}
```
helper LogExp::eval(env : Environment) : OclAny {
    if (self.condition = null or self.condition.eval(env).oclAsType(Boolean) <> false) then {
        var output : String := "";
        self.argument->forEach(a) {
            var res := a.eval(env);
            output := output + a.eval(env).print();
        }
        log(output);
    } else {
        return null;
    }
}

helper AssignExp::eval(env : Environment) : OclAny {
    var lv := self.left.lval(env).oclAsType(String);
    var rv := self.value.first().eval(env);
    if (self.left.oclIsKindOf(PropertyCallExp)) then {
        var propExpr := self.left.oclAsType(PropertyCallExp);
        var srcFrameName := propExpr.source.getName();
        var lFrame := env.get(lv).oclAsType(ObjectFrame);
        var feature := propExpr.referredProperty.oclAsType(E StructuralFeature);
        var obj := lFrame.value().oclAsType(EObject);
        if (rv.oclIsKindOf(CollectionWrapper) and
            (feature.upperBound > 1 or feature.upperBound = -1 or
            feature.eType.oclIsKindOf(CollectionType)) then {
            var col := rv.oclAsType(CollectionWrapper);
            switch {
                case (col.type = 'Sequence')
                    {setMultiFeature(obj, feature.name, col.seq);}
                case (col.type = 'OrderedSet')
                    {setMultiFeature(obj, feature.name, col.ord);}
                case (col.type = 'Bag')
                    {setMultiFeature(obj, feature.name, col.bag);}
                case (col.type = 'Set')
                    {setMultiFeature(obj, feature.name, col.set);}
                case (col.type = 'List')
                    {setMultiFeature(obj, feature.name, col.list);}
            };
        } else {
            obj.eSet(feature, rvValue);
        }
    } else {
        obj.eSet(feature, rvValue);
    }
}

2902  
2903  endif
2904  endif;
2905  //env.put(lValue.)
2906  return rValue;
2907  }

2910  /////////////////////////////////////////////////////////////////////////////
2911  //
2912  // IteratorExp::eval
2913  //
2914  /////////////////////////////////////////////////////////////////////////////

2916  helper IteratorExp::eval(env : Environment) : OclAny {
2917      var sourceCol_ := self.source.eval(env).oclAsType(CollectionWrapper);
2918      var source_ := sourceCol_.collection();
2919      var type_ := sourceCol_.type;
2920      var iterator_ := self.iterator;
2921      var itername := self.iterator->first().oclAsType(Variable).name;
2922      var res : OclAny;
2923      switch {
2924         case (self.name = 'select') {
2925            var condition_ := Function := self.makeCondition(env);
2926            var arg1 := self.condition.oclAsType(OperationCallExp).argument->first().eval(env);
2927            return sourceCol_.invoke(self.name, Sequence{env, self});
2928            if type_ = 'List' then {
2929                var reswrap_ := new CollectionWrapper();
2930                var resList_ := List(OclAny) := List();
2931                var resCol_ := Collection(OclAny);
2932                if self.condition.oclIsKindOf(TypeExp) then {
2933                    resCol_ := sourceCol_.list->xselect(i_|
2934                        self.condition.eval(
2935                            new Environment(itername, i_, env)
2936                        ).oclAsType(Boolean)
2937                    );
2938                } else {
2939                    resCol_ := sourceCol_.list->
2940                        xselect(i_|
2941                            self.condition.eval(
2942                                new Environment(itername, i_, env)
2943                            ).oclAsType(Boolean)
2944                        );
2945                }
2946                resList_->forEach(e) {
2947                    resList_->append(e);
2951            }
2952            } else {
2953                res := new CollectionWrapper {
2954                    source_->xselect(i_|
2955                        self.condition.eval(
2956                            new Environment(itername, i_, env)
2957                        ).oclAsType(Boolean)
2958                }
2959            }
2962            }
230
case (self.name = 'collect') {
    res := new CollectionWrapper{
        source_->xcollect(i_|
            self.body.eval{
                new Environment(itername, i_, env)
            })
    }
}

return res;

helper ImperativeIterateExp::eval(env : Environment) : OclAny {
    var sourceCol_ := self.source.eval(env).oclAsType(CollectionWrapper);
    var source_ := sourceCol_.collection();
    var type_ := sourceCol_.type;
    var iterator_ := self.iterator;
    var itername := self.iterator->first().oclAsType(Variable).name;
    var res : OclAny;
    switch {
    case (self.name = 'xselect') {
        //var condition_ : Function := self.makeCondition(env);
        //var arg1 := self.condition.oclAsType(OperationCallExp).argument->first().eval(env);
        return sourceCol_.invoke(self.name, Sequence{env, self});
    }
    case (self.name = 'List') then {
        var reswrap_ := new CollectionWrapper();
        var resList_ : List(OclAny) := list();
        var resCol_ : Collection(OclAny);
        if self.condition.oclIsKindOf(TypeExp) then {
            resCol_ := sourceCol_.list->xselect(i_|
                self.condition.eval{
                    new Environment(itername, i_, env)
                }).oclAsType(Boolean)
        } 
        return reswrap_.invoke(self.name, Sequence{env, self});
    }
    /*
    if type_ = 'List' then {
        var reswrap_ := new CollectionWrapper();
        var resList_ : List(OclAny) := list();
        var resCol_ : Collection(OclAny);
        if self.condition.oclIsKindOf(TypeExp) then {
            resCol_ := sourceCol_.list->
                xselect(i_|
                    self.condition.eval{
                        new Environment(itername, i_, env)
                    }).oclAsType(Boolean)
        } 
        return reswrap_.invoke(self.name, Sequence{env, self});
    }
    */
    }
resCol_~foreach(e) {  
  resList_~append(e);  
};

reswrap_.type := 'List';
reswrap_.list := resList_;  
res := reswrap_;  
}
else {  
res := new CollectionWrapper (  
  source_~xselect(i_|  
    self.condition.eval(  
      new Environment(itername, i_, env)  
    ).oclAsType(Boolean)  
  )  
);  
}
endif;
*/  
}
  
case (self.name = 'xcollect') {  
if (self.body <> null) then {  
res := new CollectionWrapper(  
  source_~xcollect(i_|  
    self.body.eval(  
      new Environment(itername, i_, env)  
    )  
  )  
};  
}
else ( // if the body is null (e.g., collection type casts)  
res := new CollectionWrapper(source_~xcollect(i_[i_]));  
}
endif;
  
case (self.name = 'selectOne') {  
res := source_~selectOne(i_|  
    self.condition.eval(  
      new Environment(itername, i_, env)  
    ).oclAsType(Boolean)  
  );  
}
  
case (self.name = 'collectOne') {  
res := source_~collectOne(i_|  
    self.body.eval(  
      new Environment(itername, i_, env)  
    ).oclAsType(Boolean)  
  );
}
  
case (self.name = 'collectselect') {  
var target := self.target.oclAsType(Variable).name;
var iterEnv := new Environment(env);
res := new CollectionWrapper(  
  source_~collectselect(i_|  
    t = self.body.eval(  
      iterEnv.put(itername, createFrame(i_))  
    )  
  ).oclAsType(Boolean)  
);
case (self.name = 'collectselectOne') {
  var target := self.target.oclAsType(Variable).name;
  var iterEnv := new Environment(env);
  res := source_->collectselectOne(
    i_,
    t = self.body.eval(
      iterEnv.put(itername, createFrame(i_))
    ).oclAsType(Boolean)
  );
}

return res;

iterEnv.put(
   itername, createFrame(i_)
) .oclAsType(Boolean)
   { self.body.eval(iterEnv);
   }
   }
}
return null;
}

/helper OclAny::makeExp()
helper OclAny::makeExp() : OCLExpression {
   if self.oclIsKindOf(EObject) then {
      var obj := self.oclAsType(EObject);
      var x := obj.makeExp();
      return x;
   } endif;
   return null;
}

/helper OCLExpression::makeExp()
helper OCLExpression::makeExp() : OCLExpression {
   return self;
}

/helper EObject::makeExp()
helper EObject::makeExp() : OCLExpression {
   var path := self.path();
   var objInModel := inputModel.getObject(path);
   var obj := self;
   if objInModel <> null and not objInModel.oclIsInvalid() then {
      obj := path.oclAsType(EObject);
   }
   endif;
   var literalExp : ObjectExp := new ObjectExp();
   literalExp.body := new ConstructorBody();
   obj.eClass().eAllAttributes->forEach(attr) {
      var attrValue : OclAny;
      etc...
if (attr.eType.oclIsKindOf(CollectionType)) then {
    var col := Sequence();
    getMultiFeature(obj, attr.name, col);
    attrValue := new CollectionWrapper(col);
} else {
    attrValue := obj.eGet(attr);
} endif;

literalExp.body.content +=
    object AssignExp {
        left := object VariableExp {
            referredVariable := object Variable {
                name := attr.name;
            };
            name := referredVariable.oclAsType(Variable).name;
        };
        value := attrValue.makeExp();
    };
literalExp.instantiatedClass := obj.eClass();

return literalExp;

   //*******************************************************************************
   // Integer::makeExp
   //*******************************************************************************

helper Integer::makeExp() : OCLExpression {
    return object IntegerLiteralExp {
        integerSymbol := self;
    }
}

   //*******************************************************************************
   // Real::makeExp
   //*******************************************************************************

helper Real::makeExp() : OCLExpression {
    return object RealLiteralExp {
        realSymbol := self;
    }
}

   //*******************************************************************************
   // String::makeExp'
   //*******************************************************************************

helper String::makeExp() : OCLExpression {
    return object StringLiteralExp {
        stringSymbol := self;
    };
}
helper Boolean::makeExp() : OCLExpression {
    return object BooleanLiteralExp {
        booleanSymbol := self;
    }
}

helper CollectionWrapper::makeExp() : OCLExpression {
    var literalExp := new CollectionLiteralExp();
    self.collection()->forEach(element) {
        literalExp.part += object CollectionItem {
            item := element.makeExp();
        };
    }
    switch {
        case (self.type = 'Set') {
            literalExp.kind := ocl::expressions::CollectionKind::Set;
            literalExp.eType := object SetType {
            };
        }
        case (self.type = 'Sequence') {
            literalExp.kind := ocl::expressions::CollectionKind::Sequence;
            literalExp.eType := object SequenceType {
            };
        }
        case (self.type = 'OrderedSet') {
            literalExp.kind := ocl::expressions::CollectionKind::OrderedSet;
            literalExp.eType := object OrderedSetType {
            };
        }
        case (self.type = 'List') {
            literalExp.kind := ocl::expressions::CollectionKind::Sequence;
            literalExp.eType := object ListType {
            };
        }
    }
}
case (self.type == 'Bag')
{
literalExp.kind := ocl::expressions::CollectionKind::Bag;
literalExp.eType := object BagType {
  
};
}
return literalExp;
}

helper CollectionLiteralExp::eval(env : Environment) : OclAny {
  var type := self.eType.name;
  var seq : Sequence(OclAny);
  self.part->forEach(p) {
    seq := seq->append(p.oclAsType(CollectionItem).eval(env));
  }
  var res : CollectionWrapper;
  switch {
    case (type.startsWith('Set'))
      {res := new CollectionWrapper(seq->asSet())}
    case (type.startsWith('Sequence'))
      {res := new CollectionWrapper(seq)}
    case (type.startsWith('OrderedSet'))
      {res := new CollectionWrapper(seq->asOrderedSet())}
    case (type.startsWith('Bag'))
      {res := new CollectionWrapper(seq->asBag())}
    case (type.startsWith('List'))
      {res := new CollectionWrapper(seq->asList())}
  }
  return res;
}

helper StringLiteralExp::eval(env : Environment) : OclAny {
  return self.stringSymbol;
}

helper IntegerLiteralExp::eval(env : Environment) : OclAny {
return self.integerSymbol;
}

// BooleanLiteralExp::eval
//
helper BooleanLiteralExp::eval(env : Environment) : OclAny {
  return self.booleanSymbol;
}

// RealLiteralExp::eval
//
helper RealLiteralExp::eval(env : Environment) : OclAny {
  return self.realSymbol;
}

// NullLiteralExp::eval
//
helper NullLiteralExp::eval(env : Environment) : OclAny {
  return null;
}

// CollectionItem::eval
//
helper CollectionItem::eval(env: Environment) : OclAny {
  return self.item.eval(env);
}

// PropertyCallExp::eval
//
helper PropertyCallExp::eval(env : Environment) : OclAny {
  var srcObject := self.source.eval(env).oclAsType(EObject);
  var srcType := srcObject.eClass();
  var feature := srcType.getEStructuralFeature(self.referredProperty.oclAsType(EStructuralFeature).name);
  if (feature.upperBound = -1) or (feature.upperBound > 1) or feature.eType.oclIsKindOf(CollectionType) then {
    var subobj := srcObject.allSubobjects();
3451 //var val := subobj->xselect(obj : EObject|obj.eContainingFeature() = feature);
3452 var val := Sequence();
3453 getMultiFeature(srcObject, feature.name, val);
3454 var value := new CollectionWrapper(val->asOrderedSet());
3455 // switch {
3456 // case (feature.eType.oclIsKindOf(SetType)) value := new CollectionWrapper(val->asSet());
3457 // case (feature.eType.oclIsKindOf(BagType)) value := new CollectionWrapper(val->asBag());
3458 // case (feature.eType.oclIsKindOf(OrderedSetType)) value := new CollectionWrapper(val->asOrderedSet());
3459 // case (feature.eType.oclIsKindOf(SequenceType)) value := new CollectionWrapper(val->asSequence());
3460 // case (feature.eType.oclIsKindOf(ListType)) value := new CollectionWrapper(val->asList());
3461 // };
3462 return value;
3463 }
3464 endif;
3465 var value := srcObject.eGet(feature);
3466 return value;
3467 }

3472 /////////////////////////////////////////////////////////////////////////////////
3473 //
3474 // VariableExp::eval
3475 //
3476 /////////////////////////////////////////////////////////////////////////////////
3477 helper VariableExp::eval(env : Environment) : OclAny {
3478 return env.get(self.name).value();
3479 }

3484 /////////////////////////////////////////////////////////////////////////////////
3485 //
3486 // VariableInitExp::eval
3487 //
3488 /////////////////////////////////////////////////////////////////////////////////
3489 helper VariableInitExp::eval(env : Environment) : OclAny {
3490 var initValue := self.referredVariable.initExpression.eval(env);
3491 var frame := createFrame(initValue);
3492 env.localEnv->put(self.referredVariable.name, frame);
3493 return initValue;
3494 }

3498 /////////////////////////////////////////////////////////////////////////////////
3499 //
3500 // WhileExp::eval
3501 //
3502 /////////////////////////////////////////////////////////////////////////////////
3503 helper WhileExp::eval(env : Environment) : OclAny {
3504 var res : OclAny := null;
3505 while(self.condition.eval(env).oclAsType(Boolean)) {
3506 res := self.body.eval(env);
3507 }
3509 return null;
3510 }
helper ComputeExp::eval(env : Environment) : OclAny {
    var initValue := self.returnedElement.initExpression.eval(env);
    var context := new Environment(self.returnedElement.name, initValue, env);
    var body := self.body.eval(context);
    var ret := context.get(self.returnedElement.name);
    return ret.value();
}

helper IfExp::eval(env : Environment) : OclAny {
    return if (self.condition.eval(env).oclAsType(Boolean)) then
         self.thenExpression.eval(env)
    else
         self.elseExpression.eval(env)
    endif;
}

helper InstantiationExp::eval(env : Environment) : OclAny {
    var type := self.instantiatedClass;
    var instance := type.ePackage.eFactoryInstance.create(type);
    return instance;
}

mapping EObject::writeObject() : EObject {
    init{
        var cont := self.eContainer();
        result := self.deepClone().oclAsType(EObject);
    }
}

helper ObjectExp::eval(env : Environment) : OclAny {
var type := self.instantiateClass;
var instance := type.ePackage.eFactoryInstance.create(type);
var newEnv := new Environment(self.referredObject.name, instance, env);
self.body.eval(newEnv);
instance := newEnv.get(self.referredObject.name).value().oclAsType(EObject);
var extFrame := newEnv.get(self.extent.name);
if extFrame <> null then {
  var extent := extFrame.value().oclAsType(Model);
  if self.referredObjectoclIsKindOf(MappingParameter) then {
    var refObj := self.referredObjectoclAsType(MappingParameter);
    if refObj.name = 'result' then {
      env.put('result', createFrame(instance));
      instance := newEnv.get(self.extent.name).
      return instance;
    }
  }
} endif;
instance.map writeObject();
return instance;

helper TypeExp::eval(env : Environment) : OclAny {
  var type := self.referredType;
  var javatype := self.prototype().oclAsType();
  return type;
}

helper ConstructorBody::eval(env : Environment) : OclAny {
  self.content->eval(env);
  //self.operation;
  //self.variable;
  return null;
}

helper ASTNode::eval(env : Environment) : OclAny {
  return null;
property _tab_ : String = '    ';

// QvtPrettyPrinter

// printTabs

helper printTabs(tabs : Integer) : String {
  var i := 0;
  var code := '';
  while (i < tabs) {
    code := code + _tab_;
    i := i + 1;
  }
  return code;
}

// printArgs

helper printArgs(list : OrderedSet(ecore::EObject)) : String {
  if (list->isEmpty()) then
    return ''
  endif;
  var code : String := list->first().print(0);
  var rest := list->subOrderedSet(2, list->size());
  rest->forEach(expr) {
    code := code + ' , ' + expr.print(0);
  }
  return code;
}

// printArgs

helper printArgs(list : OrderedSet(ecore::EModelElement)) : String {
  if (list->isEmpty()) then
    return ''
  endif;
  var code : String := list->first().print(0);
  var rest := list->subOrderedSet(2, list->size());
  rest->forEach(expr) {
    code := code + ' , ' + expr.print(0);
  }
  return code;
}
helper printArgs(list : OrderedSet(ocl::expressions::CollectionLiteralPart)) : String {
    if (list->isEmpty()) then
        return "";
    endif;
    var code : String := list->first().print(0);
    var rest := list->subOrderedSet(2, list->size());
    rest->forEach(expr) {
        code := code + ', ' + expr.print(0);
    };
    return code;
}

helper printArgs(list : OrderedSet(ocl::utilities::ASTNode)) : String {
    if (list->isEmpty()) then
        return "";
    endif;
    var code : String := list->first().print(0);
    var rest := list->subOrderedSet(2, list->size());
    rest->forEach(expr) {
        code := code + ', ' + expr.print(0);
    };
    return code;
}

helper printExpressions(exprList : OrderedSet(ocl::ecore::OCLExpression), tabs : Integer) : String {
    var code : String;
    exprList->forEach(expr) {
        code := code + printTabs(tabs) + expr.print(tabs) + ';\n';
    };
    return code;
}
3756 helper.ecore::EObject::print(tabs : Integer) : String {
3757     var nameFeature := self.eClass().getEStructuralFeature('name');
3758     return self.eGet(nameFeature).oclAsType(String);
3759 }

3762 ///////////////////////////////////////////////////////////////////////////////////////////////
3763 //
3764 // EStructuralFeature::print
3765 //
3766 ///////////////////////////////////////////////////////////////////////////////////////////////

3768 helper.ecore::EStructuralFeature::print(tabs : Integer) : String {
3769     var code : String := self.name;
3770     return code;
3771 }

3774 ///////////////////////////////////////////////////////////////////////////////////////////////
3775 //
3776 // EModelElement::print
3777 //
3778 ///////////////////////////////////////////////////////////////////////////////////////////////

3780 helper.ecore::EModelElement::print(tabs : Integer) : String {
3781     assert(true);
3782     return "";
3783 }

3786 ///////////////////////////////////////////////////////////////////////////////////////////////
3787 //
3788 // ASTNode::print
3789 //
3790 ///////////////////////////////////////////////////////////////////////////////////////////////

3792 helper.ocl::utilities::ASTNode::print(tabs : Integer) : String {
3793     assert(true);
3794     return "";
3795 }

3798 ///////////////////////////////////////////////////////////////////////////////////////////////
3799 //
3800 // OperationalTransformation::print
3801 //
3802 ///////////////////////////////////////////////////////////////////////////////////////////////

3805 helper.OperationalTransformation::print(tabs : Integer) : String {
3806     var code : String := "";
3807     self.usedModelType->reject(mt|mt.name = '_INTERMEDIATE')->forEach(mt) {
3808         code := code + printTabs(tabs) + mt.print(0);
3809     };
3810     code := code + '\n' + printTabs(tabs) + 'transformation ' + self.name + '{';
3811     code := code + printArgs(self.modelParameter);
3812     code := code + '}\n';
3813     self.configProperty->forEach(p) {
3814         code := code + '\nconfiguration property ' + p.name + ' : ' + p.eType.name + ';';
3815     };
3816 }

3817    });
3818    code := code + '
';
3820    self.intermediateClass->forEach(klass) {
3821        code := code + '\nintermediate class ' + klass.name + ' \n';
3822        klass.eAttributes->forEach(attr) {
3823            code := code + printTabs(tabs + 1) + attr.name + ' : ' + attr.eType.name + '\n';
3824        }
3825    }
3826    code := code + '}';
3827    }
3828    code := code + '\n';
3830    self.intermediateProperty->forEach(p) {
3831        var cp := p.oclAsType(ContextualProperty);
3832        code := code + '\ninintermediate property ' +
3833            cp.context.name + ' : ' + cp.name + ' : ' + cp.eType.name + ';
3834        if cp.initExpression <> null then {
3835            code := code + '=' + cp.initExpression.print(tabs);
3836        } endif;
3837        code := code + '};
3838    }
3839    code := code + '\n';
3841    var set := self.eAllStructuralFeatures - self.configProperty;
3842    set->forEach(p|p.oclIsKindOf(EAttribute)) {
3843        var pa := p.oclAsType(EAttribute);
3844        code := code + '\nproperty ' + pa.name + ' : ' + pa.eType.name;
3845        code := code + '=' + pa.eAnnotations.contents->first().oclAsType(ocl::ecore::OCLExpression).print(0) + ';
3846    }
3847    code := code + '\n';
3848    code := code + '\n' + self.entry.print(tabs);
3849    self.eOperations->forEach(op | op.oclIsKindOf(MappingOperation)) {
3850        code := code + '\n' + op.oclAsType(MappingOperation).print(tabs);
3851    }
3852    self.eOperations->forEach(op | op.oclIsKindOf(Helper)) {
3853        code := code + '\n' + op.oclAsType(Helper).print(tabs);
3854    }
3855    return code;
3857 }
3858
3861 helper ModelType::print(tab : Integer) : String {
3862    var code : String := 'modeltype ' + self.name;
3863    if (self.conformanceKind != null) then {
3864        code := code + ' ' + self.conformanceKind + '"';
3865    }
3866    endif;
code := code + 'uses ';
var package := self._metamodel->first();
var meta : String := package.name;
package := package.eSuperPackage;
while (package != null) {
  meta := package.name + '::' + meta;
  package := package.eSuperPackage;
}
code := code + '{' + self._metamodel->first().nsURI.quotify('\') + '};' + '\n';
return code;

helper ModelParameter::print(tabs : Integer) : String {
  var code : String := self.kind.repr() + ' ' + self.name + '; ' + self.eType.name;
  return code;
}

helper VarParameter::print(tabs : Integer) : String {
  var code : String := self.name + ' : ' + self.eType.name;
  return code;
}

helper EntryOperation::print(tabs : Integer) : String {
  var code : String = printTabs(tabs) + 'main () {
';
  code := code + self.body.print(tabs + 1) + printTabs(tabs) + '} \n';
  return code;
}

helper Helper::print(tabs : Integer) : String {
  var code : String = printTabs(tabs) + 'helper ';
  if (self.context != null) then {
    code := code + self.context.eType.name + '::';
  }
}
3939  endif;
3940  code := code + self.name;
3941  code := code + '(' + printArgs(self.eParameters) + ')';
3942  code := code + ':' + printArgs(self._result) + '
';
3943  code := code + self.body.print(tabs + 1);
3944  code := code + printTabs(tabs) + '
';
3945  return code;
3946 }

3949 //unce/*******/
3950 //
3951 // Helper::signature
3952 //
3953 //unce/*******/

3955 helper Helper::signature() : String {
3956  var code : String;
3957  if (self.context != null) then {
3958    code := code + self.context.eType.name + ':';
3959  }
3960  endif;
3961  code := code + self.name;
3962  code := code + '(' + printArgs(self.eParameters) + ')';
3963  return code;
3964 }

3967 //unce/*******/
3968 //
3969 // MappingParameter::print
3970 //
3971 //unce/*******/

3973 helper MappingParameter::print(tabs : Integer) : String {
3974  var code : String := self.name + ':' + self.eType.name;
3975  return code;
3976 }

3979 //unce/*******/
3980 //
3981 // MappingOperation::print
3982 //
3983 //unce/*******/

3985 helper MappingOperation::print(tabs : Integer) : String {
3986  var code : String = printTabs(tabs) + 'mapping ' + self.context.eType.name + ':' + self.name;
3987  code := code + '(' + printArgs(self.eParameters) + ')';
3988  code := code + ':' + self._result->first().eType.name;
3989  if (self._when->notEmpty()) then {
3990    code := code + '
' + printTabs(tabs) + 'when (' + self._when->first().print(0);
3991    var rest := self._when->asSequence()->subSequence(2, self._when->size());
3992    rest->forEach(expr) {
3993      code := code + '; ' + expr.print(0);
3994    };
3995    code := code + ')
';
3996  }
3997  endif;
3998  code := code + '
' + printTabs(tabs) + '
';
3999  code := code + self.body.print(tabs + 1);

247
code := code + printTabs(tabs) + '}\n';
return code;
}

(helper OperationBody::print(tabs : Integer) : String {
var code : String := printExpressions(self.content, tabs);
return code;
})

(helper MappingBody::print(tabs : Integer) : String {
var code : String;
var needsPopulation := self.initSection->notEmpty();
if self.initSection->notEmpty() then {
  code := printTabs(tabs) + 'init \n';
  code := code + printExpressions(self.initSection, tabs + 1);
  code := code + printTabs(tabs) + '}\n';
} endif;
if needsPopulation then {
  code := code + printTabs(tabs) + 'population \n';
  code := code + printExpressions(self.content, tabs + 1);
  code := code + printTabs(tabs) + '}\n';
} else {
  code := code + printExpressions(self.content, tabs);
} endif;
if self.endSection->notEmpty() then {
  code := code + printTabs(tabs) + 'end \n';
  code := code + printExpressions(self.endSection, tabs + 1);
  code := code + printTabs(tabs) + '}\n';
} endif;
return code;
})

(helper MappingCallExp::print(tabs : Integer) : String {
var code : String := self.source.print(0) + '.' + 'map ';
return code + self.referredOperation.print(0) + '(' + printArgs(self.argument) + ')';
return code;

helper ImperativeIterateExp::print(tabs : Integer) : String {

    var code : String := self.source.print(0) + '->' +
    self.name +
    '(';

    var iter := self.iterator->first().repr();
    code := code + iter;
    /*
    var iter := self.iterator->first().oclAsType(ocl::ecore::Variable);
    code := code + iter.name + ':' + iter.eType.name;
    self.iterator->subOrderedSet(2,self.iterator->size())->forEach(it) {
        iter := it.oclAsTyp(ocl::ecore::Variable);
        code := code + ',' + iter.name + iter.eType.name;
    }
    */
    if (self.body != null) then
        code := code + '|' + self.body.print(0)
    endif;
    if (self.condition != null) then
        code := code + '|' + self.condition.print(0)
    endif;
    code := code + ')';
    return code;
}

helper ImperativeLoopExp::print(tabs : Integer) : String {

    var code : String := self.source.print(0) + '->' +
    self.name +
    '(';
    var iter := self.iterator->first().repr();
    code := code + iter;
    /*
    var iter := self.iterator->first().oclAsType(ocl::ecore::Variable);
    code := code + iter.name + ':' + iter.eType.name;
    self.iterator->subOrderedSet(2,self.iterator->size())->forEach(it) {
        iter := it.oclAsTyp(ocl::ecore::Variable);
        code := code + ',' + iter.name + iter.eType.name;
    }
    */
    if (self.condition != null) then
        code := code + '|' + self.condition.print(0)
    endif;
    endif;
}

//
4122  code := code + ');

4124  if (self.body != null) then {
4125    code := code + self.body.print(tabs);
4126  }
4127  endif;
4129  return code;
4130 }

4135  /////////////////////////////////////////////////////////////////////////////
4136  /////////////////////////////////////////////////////////////////////////////
4137  // ObjectExp::print
4138  /////////////////////////////////////////////////////////////////////////////
4139  /////////////////////////////////////////////////////////////////////////////

4141  helper ObjectExp::print(tabs : Integer) : String {
4142    var code : String := 'object {;' +
4143    if (self.referredObject <> null and self.referredObject.name <> '') then {
4144      code := code + self.referredObject.name + ':' +
4145    } endif;
4148    if (self.instantiatedClass <> null and self.instantiatedClass.name <> '') then {
4149      code := code + self.instantiatedClass.name +
4152    } then {
4153      code := code + self.instantiatedClass.name
4154    }
4156    else {
4157      if (self.eType <> null and self.eType.name <> '') then {
4158        code := code + ':' + self.eType.name;
4160      } endif;
4162    } endif;
4164    if self.extent <> null then {
4165      code := code + 'g' + self.extent.name;
4166    } endif;
4168    code := code + '
' +
4169    code := code + self.body.print(tabs + 1);
4170    code := code + printTabs(tabs) + '}
' +
4171    return code;
4172 }

4175  /////////////////////////////////////////////////////////////////////////////
4176  /////////////////////////////////////////////////////////////////////////////
4177  // SwitchExp::print
4178  /////////////////////////////////////////////////////////////////////////////
4179  /////////////////////////////////////////////////////////////////////////////

4181  helper SwitchExp::print(tabs : Integer) : String {
4182    var code : String := 'switch {n' +
4183    code := code + self.body

code := code + printExpressions(self.alternativePart, tabs + 1);
if (self.elsePart != null) then {
    code := code + '\n' + printTabs(tabs + 1) + 'else ' + self.elsePart.print(tabs + 1);
}
endif;
return code;

helper AltExp::print(tabs : Integer) : String {
    var code : String := 'case ' + '{' + self.condition.print(0) + '}' + '\n';
    code := code + printTabs(tabs) + self.body.print(tabs);
    return code;
}

helper AssertExp::print(tabs : Integer) : String {
    var code : String := 'assert(' + self.assertion.print(0) + ')';
    return code;
}

helper AssignExp::print(tabs : Integer) : String {
    var code : String := self.left.print(0);
    code := code + ' := ';
    self.value->forEach(expr) {
        code := code + expr.print(0);
    }
    return code;
}

helper BlockExp::print(tabs : Integer) : String {
    var code : String := '{\n';
    code := code + printExpressions(self.body, tabs + 1);
    code := code + printTabs(tabs) + '}';
}
```java
return code

helper BreakExp::print(tabs : Integer) : String {
  var code : String := 'break';
  return code;
}

helper CatchExp::print(tabs : Integer) : String {
  var code : String := 'catch';
  return code;
}

helper ComputeExp::print(tabs : Integer) : String {
  var code : String := 'compute(' + self.returnedElement.repr() + ')';
  code := code + self.body.print(tabs);
  return code;
}

helper ContinueExp::print(tabs : Integer) : String {
  var code : String := 'continue';
  return code;
}

helper InstantiationExp::print(tabs : Integer) : String {
  var code : String := 'InstantiationExpNotImpl';
```
4305  return code;
4306  }

4309  //........................................................................
4310  // DictLiteralPart::print
4311  //
4312  //........................................................................
4313  helper DictLiteralPart::print(tabs : Integer) : String {
4314    var code : String := self.key.print(tabs) + ' = ' + self.value.print(tabs);
4315    return code;
4316  }

4321  //........................................................................
4322  // DictLiteralExp::print
4323  //
4324  //........................................................................
4325  helper DictLiteralExp::print(tabs : Integer) : String {
4326    var code : String := 'Dict {
4327    code := code + printArgs(self.part->asOrderedSet());
4328    code := code + ' }
4329    return code;
4330  }

4335  //........................................................................
4336  // ListLiteralExp::print
4337  //
4338  //........................................................................
4339  helper ListLiteralExp::print(tabs : Integer) : String {
4340    var code : String := 'ListLiteralExpNotImpl'
4341    return code;
4342  }

4347  //........................................................................
4348  // LogExp::print
4349  //
4350  //........................................................................
4351  helper LogExp::print(tabs : Integer) : String {
4352    var code : String := 'log(' + printArgs(self.argument) + ')'
4353    return code;
4354  }

4360  //........................................................................
4361  // RaiseExp::print
4362  //
4363  //........................................................................
4366 helper RaiseExp::print(tabs : Integer) : String {
4367 var code : String := 'raise';
4368 return code;
4369 }

4372 //________________________________________________________________________________________
4373 //
4374 // ReturnExp::print
4375 //
4376 //________________________________________________________________________________________

4378 helper ReturnExp::print(tabs : Integer) : String {
4379 var code : String := 'return ' + self.value.print(tabs);
4380 return code;
4381 }

4385 //________________________________________________________________________________________
4386 //
4387 // TryExp::print
4388 //
4389 //________________________________________________________________________________________

4391 helper TryExp::print(tabs : Integer) : String {
4392 var code : String := 'try';
4393 return code;
4394 }

4397 //________________________________________________________________________________________
4398 //
4399 // Typedef::print
4400 //
4401 //________________________________________________________________________________________

4403 helper Typedef::print(tabs : Integer) : String {
4404 var code : String := 'typedef';
4405 return code;
4406 }

4411 //________________________________________________________________________________________
4412 //
4413 // UnlinkExp::print
4414 //
4415 //________________________________________________________________________________________

4417 helper UnlinkExp::print(tabs : Integer) : String {
4418 var code : String := 'unlink';
4419 return code;
4420 }

4423 //________________________________________________________________________________________
4424 //
4425 // UnpackExp::print
4426 //
helper UnpackExp::print(tabs : Integer) : String {
    var code : String := 'unpack';
    return code;
}

helper VariableInitExp::print(tabs : Integer) : String {
    var code := 'var ' + self.referredVariable.name;
    if (self.referredVariable.eType !== null and
        self.referredVariable.eType.name !== '') then {
        code := code + ': ' + self.referredVariable.eType.name;
    }
    endif;
    if (self.referredVariable.initExpression !== null) then {
        var iexp := self.referredVariable.initExpression;
        code := code + ':=' + iexp.print(tabs).replace('n', ' ').replace(' ', ' ');
    }
    endif;
    return code;
}

helper WhileExp::print(tabs : Integer) : String {
    var code : String := 'while (' + self.condition.print(0) + ') ';
    code := code + self.body.print(tabs);
    return code;
}

helper ocl::expressions::CollectionLiteralPart::print(tabs : Integer) : String {
    return self.repr();
}

helper ocl::expressions::CollectionItem::print

helper ocl::expressions::CollectionLiteralPart::print(tabs : Integer) : String {
    return self.repr();
}
helper CollectionItem::print(tabs : Integer) : String {
    return self.item.print(0);
}

(helper) CollectionLiteralExp::print(tabs : Integer) : String {
    var code : String := '';
    switch {
    case (self.eType.oclIsTypeOf(SetType) or self.kind = ocl::expressions::CollectionKind::Set)
    { code := code + 'Set{';
    }
    case (self.eType.oclIsTypeOf(SequenceType) or self.kind = ocl::expressions::CollectionKind::Sequence)
    { code := code + 'Sequence{';
    }
    case (self.eType.oclIsTypeOf(OrderedSetType) or self.kind = ocl::expressions::CollectionKind::OrderedSet)
    { code := code + 'OrderedSet{';
    }
    case (self.eType.oclIsTypeOf(BagType) or self.kind = ocl::expressions::CollectionKind::Bag)
    { code := code + 'Bag{';
    }
    case (self.eType.oclIsTypeOf(ListType))
    { code := code + 'List{'
    }
    }
    code := code + printArgs(self.part);
    code := code + '});
    return code;
}

ocl::ecore::TypeExp::print(tabs : Integer) : String {
    var code : String := self.referredType.print(0);
    return code;
}

ocl::ecore::TypeExp::print(tabs : Integer) : String {
    var code : String := self.referredType.print(0);
    return code;
}
helper ocl::ecore::StringLiteralExp::print(tabs : Integer) : String {
    var code : String := self.stringSymbol.replace("\n","\n").replace("\t","\t").quotify("\"");
    return code;
}

helper ocl::ecore::OperationCallExp::print(tabs : Integer) : String {
    var code : String := "";
    if (self.source <> null)
        code := code + self.source.print(0);
    if (opName = '+' or opName = '-' or opName = '*' or opName = '/' or opName = 'mod' or opName = '<' or opName = '>' or opName = 'eq' or opName = '<>' or opName = '=<' or opName = '=>') then
        code := code + ' ' + opName + ' ' + printArgs(self.argument);
    return code;
}

helper ocl::ecore::PropertyCallExp::print(tabs : Integer) : String {
    var code : String := self.source.print(0);
    if (opName = '+' or opName = '-' or opName = '*' or opName = '/' or opName = 'mod' or opName = '<' or opName = '>' or opName = 'eq' or opName = '<>' or opName = '=<' or opName = '=>') then
        code := code + ' ' + opName + ' ' + printArgs(self.argument);
    return code;
}

helper ocl::ecore::OperationCallExp::print(tabs : Integer) : String {
    var code : String := self.source.print(0);
    if (opName = '+' or opName = '-' or opName = '*' or opName = '/' or opName = 'mod' or opName = '<' or opName = '>' or opName = 'eq' or opName = '<>' or opName = '=<' or opName = '=>') then
        code := code + ' ' + opName + ' ' + printArgs(self.argument).
    return code;
}
code := code + '.' + self.referredProperty.print(0);

//Sequence(self.referredProperty)->switch(p) {
//  case (poclIsKindOf(EAttribute)) (code := code + '.' + self.referredProperty.oclAsType(EAttribute).name)
//  case (poclIsKindOf(EReference)) (code := code + '.' + self.referredProperty.oclAsType(EReference).name)
//  else {}
//};
return code;
}

helper ocl::ecore::IfExp::print(tabs : Integer) : String {
var code : String := 'if ' + '(' + self.condition.print(0) + ') ' + 'then ';
code := code + self.thenExpression.print(tabs) + '
';
if self.elseExpression != null then {
code := code + printTabs(tabs) + 'else ' + self.elseExpression.print(tabs) + '
';
endif;
code := code + printTabs(tabs) + 'endif';
return code;
}

helper ocl::ecore::OCLExpression::print(tabs : Integer) : String {
var code : String := self.repr();
return code;
}

helper ocl::expressions::OCLExpression::print(tabs : Integer) : String {
var code : String := self.repr();
return code;
}

intermediate class TransformationContext
{
fixedElements : OrderedSet<EObject>;
varElements : OrderedSet<EObject>;
}
varClasses : OrderedSet(EClass);
inModel : ModelParameter;
outModel : ModelParameter;
};

constructor TransformationContext::TransformationContext() {
}

helper createTransformationContext() : TransformationContext {
    return new TransformationContext();
}

helper OperationalTransformation::bta(bt : Dict(String, BTAKind), inout context : TransformationContext) : BTAKind {
    var inModel : ModelParameter := self.modelParameter->select(mp|mp.kind = DirectionKind::_in)->asSequence()->first();
    var outModel : ModelParameter := self.modelParameter->select(mp|mp.kind = DirectionKind::_out)->asSequence()->first();
    context.inModel := inModel;
    context.outModel := outModel;
    var inModelType : EPackage := inModel.eType.oclAsType(ModelType)._metamodel->first();
    var outModelType := outModel.eType.oclAsType(ModelType)._metamodel->first();
    var annotations := inModelType.eAnnotations;
    var fixedAnnot := annotations->select(ann|ann.source = 'FIXED')->asSequence()->first();
    var varAnnot := annotations->select(ann|ann.source = 'VAR')->asSequence()->first();
    context.fixedElements := fixedAnnot._references;
    varAnnot._references->forEach(element) {
        switch {
            case (element.oclIsTypeOf(EClass))
            {
                context.varClasses += element.oclAsType(EClass);
                break;
            }
            case (element.oclIsTypeOf(EReference))
            {
                var ref := element.oclAsType(EReference);
                context.varClasses += ref.eContainingClass;
            }
            case (element.oclIsTypeOf(EAttribute))
            {
                var attr := element.oclAsType(EAttribute);
                context.varClasses += attr.eContainingClass;
            }
        }
    };
}
var statements := self.entry.body.content;
var transBTA := BTAKind::STATIC;
statements->forEach(expr) {
    if (expr.bta(bt, context) = BTAKind::DYNAMIC) then {
        transBTA := BTAKind::DYNAMIC;
    }
    endif;
};
return transBTA;

helper ASTNode::bta(bt: Dict(String, BTAKind), inout context : TransformationContext) : BTAKind {
    return BTAKind::DYNAMIC;
}

helper PropertyCallExp::bta(bt: Dict(String, BTAKind), inout context : TransformationContext) : BTAKind {
    var srcBT := self.source.bta(bt, context);
    if srcBT = BTAKind::STATIC then {
        return BTAKind::STATIC;
    }
    endif;
    return BTAKind::MAYBE;
}

helper LiteralExp::bta(bt: Dict(String, BTAKind), inout context : TransformationContext) : BTAKind {
    return BTAKind::STATIC;
}

helper LogExp::bta(bt: Dict(String, BTAKind), inout context : TransformationContext) : BTAKind {
    return BTAKind::DYNAMIC;
}
helper ReturnExp::bta(bt : Dict(String, BTAKind), inout context : TransformationContext) : BTAKind {
    return self.value.bta(bt, context);
}

helper AssignExp::bta(bt : Dict(String, BTAKind), inout context : TransformationContext) : BTAKind {
    var valueBT := self.value->first().bta(bt, context);
    if valueBT <> BTAKind::STATIC then {
        return valueBT;
    } endif;
    return self.left.bta(bt, context);
}

helper VariableExp::bta(bt : Dict(String, BTAKind), inout context : TransformationContext) : BTAKind {
    if (bt->hasKey(self.name)) then {
        return bt->get(self.name);
    } endif;
    return BTAKind::MAYBE;
}

helper VariableInitExp::bta(bt : Dict(String, BTAKind), inout context : TransformationContext) : BTAKind {
    var varName := self.referredVariable.name;
    var staticReferredVars := self.referredVariable.initExpression.
        allSubobjectsOfType(VariableExp)->oclAsType(VariableExp)->select(v|bt->get(v.name) = BTAKind::STATIC)

    var dynamicReferredVars :=
        self.referredVariable.initExpression.allSubobjectsOfType(VariableExp)->
oclAsType(VariableExp)->select(v|bt->get(v.name) = BTAKind::DYNAMIC)

var initBTA := self.referredVariable.initExpression.bta(bt, context);
bt->put(varName, initBTA);
return initBTA;

if (dynamicReferredVars->size() > 0) then {
bt->put(varName, BTAKind::DYNAMIC);
return BTAKind::DYNAMIC;
} else {
bt->put(varName, BTAKind::STATIC);
return BTAKind::STATIC;
}

helper ImperativeIterateExp::bta(bt : Dict<String, BTAKind>, inout context : TransformationContext) : BTAKind {
return self.source.bta(bt, context);
}

helper OperationCallExp::bta(bt : Dict<String, BTAKind>, inout context : TransformationContext) : BTAKind {
if (self.source.oclIsTypeOf(VariableExp) and
self.source.oclAsType(VariableExp).name = context.inModel.name) then {
var modelRef := self.source.oclAsType(VariableExp);
var op := self.referredOperation.oclAsType(EOperation);
if (op.name = 'objectsOfType') then {
var type := self.argument->first().oclAsType(TypeExp).eType.name;
if (context.varClasses->exists(cl|cl.name = type)) then {
return BTAKind::DYNAMIC;
}
else {
// CHECK this later: if a type in the input metamodel is not in the var classes
// can it be declared as STATIC in all situations?
return BTAKind::STATIC;
}
}
else {
// CHECK this later: if a type in the input metamodel is not in the var classes
// can it be declared as STATIC in all situations?
return BTAKind::STATIC;
}
}
}
var srcBTA : BTAKind;
if (self.source.oclIsTypeOf(VariableExp) and
self.source.oclAsType(VariableExp).name = 'this') then {
srcBTA := BTAKind::STATIC;
```plaintext
4914 }
4915 else {
4916   srcBTA := self.source.bta(bt, context);
4917 }
4918 endif;
4919 var callArgs := self.argument;
4920 var callArgsBT := callArgs->bta(bt, context);
4921 var anyNonStaticArg := callArgsBT->exists(t|t <> BTAKind::STATIC);
4922 if (srcBTA <> BTAKind::STATIC or anyNonStaticArg) then {
4923   return BTAKind::MAYBE;
4924 }
4925 endif;
4926 if not self.referredOperation.oclIsKindOf(Helper) then {
4927   return BTAKind::STATIC;
4928 }
4929 endif;
4930 var helperOp := self.referredOperation.oclAsType(Helper);
4931 if bt->hasKey(helperOp.signature()) then {
4932   return BTAKind::STATIC;
4933 }
4934 endif;
4935 var params := helperOp.eParameters->asOrderedSet();
4936 var newBt : Dict(String, BTAKind) := Dict();
4937 newBt->put(helperOp.signature(), BTAKind::MAYBE);
4938 bt->keys()->forEach(k) {
4939   newBt->put(k, bt->get(k));
4940 }
4941 var i := 1;
4942 var n := params->size();
4943 while (i <= n) {
4944   newBt->put(params->at(i).name, callArgsBT->at(i));
4945   i := i + 1;
4946 }
4947 newBt->put('self', srcBTA);
4948 var bodyBT := self.referredOperation.oclAsType(Helper).body.content->bta(newBt, context);
4949 if bodyBT->exists(b|b <> BTAKind::STATIC) then {
4950   return BTAKind::MAYBE;
4951 }
4952 endif;
4953 return BTAKind::STATIC;
4954 }
4955
4956 // ConstructorBody::bta
4957 // ConstructorBody::bta
4958
4959 helper ConstructorBody::bta(bt : Dict(String, BTAKind), inout context : TransformationContext) : BTAKind {
4960 if (self.content->bta(bt, context)->exists(x|x <> BTAKind::STATIC)) then {
4961   return BTAKind::MAYBE;
4962 }
4963 endif;
4964 return BTAKind::STATIC;
```
helper ObjectExp::bta(bt : Dict(String, BTAKind), inout context : TransformationContext) : BTAKind {
    return self.body.bta(bt, context);
}

helper IfExp::bta(bt : Dict(String, BTAKind), inout context : TransformationContext) : BTAKind {
    var conditionBT := self.condition.bta(bt, context);
    var thenBT := self.thenExpression.bta(bt, context);
    var elseBT := if self.elseExpression <> null then self.elseExpression.bta(bt, context) else BTAKind::STATIC endif;
    if (thenBT = BTAKind::STATIC and elseBT = BTAKind::STATIC) then {
        return BTAKind::STATIC;
    }
    endif;
    if (conditionBT = BTAKind::STATIC) then {
        // if we could eval the condition
        return BTAKind::MAYBE;
    }
    endif;
    return BTAKind::DYNAMIC;
}

helper ForExp::bta(bt : Dict(String, BTAKind), inout context : TransformationContext) : BTAKind {
    var srcBT := self.source.bta(bt, context);
    var itername := self.iterator->first().oclAsType(Variable).name;
    if srcBT = BTAKind::STATIC then {
        var newBT : Dict(String, BTAKind) := Dict();
        bt->keys()->forEach(k) {
            newBT->put(k, bt->get(k));
        };
        newBT->put(itername, srcBT);
        var conditionBT :=
            if (self.condition <> null) then
                self.condition.bta(newBT, context)
            else
                BTAKind::STATIC
            endif
        ;
        if conditionBT = BTAKind::STATIC then {
            var bodyBT := self.body.bta(newBT, context);
        }
    };
}
return bodyBT;
}  
}  
endif;

return srcBT;
}  

helper WhileExp::bta(bt : Dict(String, BTAKind), inout context : TransformationContext) : BTAKind {
  var condBT := self.condition.bta(bt, context);
  if (condBT = BTAKind::STATIC) then {
    return self.body.bta(bt, context);
  }  
  endif;
  return condBT;
}  

helper BlockExp::bta(bt : Dict(String, BTAKind), inout context : TransformationContext) : BTAKind {
  var bodyBT := self.body->bta(bt, context);
  var blockBT : BTAKind;
  if bodyBT->exists(b|b = BTAKind::DYNAMIC) then {
    blockBT := BTAKind::DYNAMIC
  }  
  else {
    if bodyBT->exists(b|b = BTAKind::MAYBE) then {
      blockBT := BTAKind::MAYBE;
    }  
    else {
      blockBT := BTAKind::STATIC;
    }  
  }  
  endif;
  return blockBT;
}  

property bindingTimes : Dict(String, BTAKind) = Dict();
property environment : Environment = new Environment();
property context : TransformationContext = new TransformationContext();
property newOperations : OrderedSet<EOperation> = OrderedSet();
property memoizeTables : Dict(String, EAttribute) = Dict();
property tempCount : Integer = 0;
property mixCount : Dict(String, Integer) = Dict();

helper ImperativeOperation::newMix() : String {
    var res : String := self.name + '__mix';
    if mixCount->hasKey(self.name) then {
        var n := mixCount->get(self.name);
        n := n + 1;
        mixCount->put(self.name, n);
        res := res + n.print();
    } else {
        mixCount->put(self.name, 1);
        res := res + '1';
    } endif;
    return res;
}

helper MappingOperation::mappingMixName() : String {
    var res : String := self.name + '__mix';
    if not mixCount->hasKey(self.name) then {
        mixCount->put(self.name, 1);
    } endif;
    return res;
}

helper newTemp() : String {
    tempCount := tempCount + 1;
    return '__temp_' + tempCount.print();
}

helper OperationalTransformation::mix

helper OperationalTransformation::mix() : OperationalTransformation {
  //context := createTransformationContext();
  self.modelParameter->forEach(m) {
    switch {
      case (m.kind = DirectionKind::_in)
      {
        environment.put(m.name, createFrame(inputModel));
      }
      case (m.kind = DirectionKind::_out)
      {
        environment.put(m.name, createFrame(outputModel));
      }
    };
  }
  var ops := self.eOperations;
  ops := ops->reject(o|o.oclIsKindOf(EntryOperation));
  var res := object OperationalTransformation {
    name := self.name;
    eStructuralFeatures := self.eStructuralFeatures;
    modelParameter := self.modelParameter;
    moduleImport := self.moduleImport;
    usedModelType := self.usedModelType;
    eOperations := ops;
    intermediateClass := self.intermediateClass;
    intermediateProperty := self.intermediateProperty;
    configProperty := self.configProperty;
    var newEntry := self.entry.mix(environment, bindingTimes);
    entry := newEntry;
    eOperations += newEntry;
    eStructuralFeatures += memoizeTables->values();
    newOperations->forEach(o) {
      res.eOperations += o;
    };
  };
  return res;
}

helper EntryOperation::mix(env : OclAny, bt : Dict(String, BTKind)) : EntryOperation {
  tempCount := 0;
  var ret : EntryOperation := object EntryOperation {
    name := self.name;
    context := self.context;
    body := self.body.mix(env, bt);
  };
}
5219  return ret;
5220 }

5223////////////////////////////////////////////////////////////////////////
5224//
5225// Helper::mix
5226//
5227////////////////////////////////////////////////////////////////////////
5229helper Helper::mix(env : OclAny, bt : Dict(String, BTAKind)) : Helper {
5230  var oldTempCount := tempCount;
5231  tempCount := 0;
5232  var resOp := self.deepclone().oclAsType(Helper);
5234  resOp.body := resOp.body.mix(env, bt);
5235  tempCount := oldTempCount;
5236  return resOp;
5237 }

5240////////////////////////////////////////////////////////////////////////
5241//
5242// MappingBody::mix
5243//
5244////////////////////////////////////////////////////////////////////////
5246helper MappingBody::mix(env : OclAny, bt : Dict(String, BTAKind)) : OperationBody {
5247  var mixInit : OrderedSet(OCLExpression) := OrderedSet();
5248  var originalInit := self.initSection->deepclone().oclAsType(OCLExpression);
5249  originalInit->forEach(expr) {
5250    var bindingTime := expr.onlineBta(env, bt);
5251    if (bindingTime = BTAKind::DYNAMIC) then {
5252      mixInit += expr.mixReduce(env, bt);
5253    } else {
5254      mixInit += expr.reduce(env, bt);
5255    }
5256  } endif;
5257  var mixPopulation : OrderedSet(OCLExpression) := OrderedSet();
5258  var originalPopulation := self.content->deepclone()->oclAsType(OCLExpression);
5259  originalPopulation->forEach(expr) {
5260    var bindingTime := expr.onlineBta(env, bt);
5261    if (bindingTime = BTAKind::DYNAMIC) then {
5262      mixPopulation += expr.polyMixReduce(env, bt, self.operation);
5263    } else {
5264      mixPopulation += expr.reduce(env, bt);
5265    } endif;
5266  }
5267  var mixBody : MappingBody := new MappingBody();
5268  mixBody.content := mixPopulation;
5269  mixBody.initSection := mixInit;
5270  return mixBody;
5271 }

5279////////////////////////////////////////////////////////////////////////
// MappingOperation::mix
 confessed

helper MappingOperation::mix(env : OclAny, bt : Dict(String, BTKind)) : MappingOperation {
  var oldTempCount := tempCount;
  tempCount := 0;
  var ret := self.deepclone().oclAsType(MappingOperation);
  ret.name := ret.mappingMixName();
  ret.body := ret.body.mix(env, bt);
  ret.eParameters := self.eParameters->deepclone()->oclAsType(EParameter);
  tempCount := oldTempCount;
  return ret;
}

// OperationBody::mix

helper OperationBody::mix(env : OclAny, bt : Dict(String, BTKind)) : OperationBody {
  var expressions := OrderedSet(OCLExpression) := OrderedSet();
  var originalExpr := self.content->deepclone()->oclAsType(OCLExpression);
  originalExpr->forEach(expr) {
    var bindingTime := expr.onlineBta(env, bt);
    if (bindingTime = BTAKind::DYNAMIC) then {
      expressions += expr.mixReduce(env, bt);
    } else {
      expressions += expr.reduce(env, bt);
    }
  }
  if (expressions->last().oclIsTypeOf(ReturnExp)) then {
    break;
  } else {
    endif;
  }
  var mixBody : OperationBody := object OperationBody {
    content := expressions;
  }
  return mixBody;
}

// ASTNode::onlineBta

helper ASTNode::onlineBta(env : OclAny, bt : Dict(String, BTKind)) : BTKind {
  return BTKind::STATIC;
}

...
helper OCLExpression::onlineBta(env : OclAny, bt : Dict(String, BTAKind)) : BTAKind {
  return
  if self.bta(bt, context) = BTAKind::STATIC then
    BTAKind::STATIC
  else
    BTAKind::DYNAMIC
  endif
};
}

// VariableExp::onlineBta

helper VariableExp::onlineBta(env : OclAny, bt : Dict(String, BTAKind)) : BTAKind {
  var offBt := self.bta(bt, context);
  if offBt = BTAKind::MAYBE then {
    if getEnvironment(env).hasKey(self.name) then {
      return BTAKind::STATIC;
    } else {
      return BTAKind::DYNAMIC;
    }
  } endif;
};

// VariableInitExp::onlineBta

helper VariableInitExp::onlineBta(env : OclAny, bt : Dict(String, BTAKind)) : BTAKind {
  return self.referredVariable.initExpression.onlineBta(env, bt);
}

// IfExp::onlineBta

helper IfExp::onlineBta(env : OclAny, bt : Dict(String, BTAKind)) : BTAKind {
  var conditionBT := self.condition.onlineBta(env, bt);
  if conditionBT = BTAKind::STATIC then {
    var condition : Boolean := self.condition.eval(getEnvironment(env)).oclAsType(Boolean);
    if condition then {
      return self.thenExpression.onlineBta(env, bt);
    }
  }
}
else {
    return self.elseExpression.onlineBta(env, bt);
}
endif;

var thenBt := self.thenExpression.onlineBta(env, bt);
var elseBt := self.elseExpression.onlineBta(env, bt);
if thenBt = BTAKind::STATIC and elseBt = BTAKind::STATIC then {
    var thenVal := self.thenExpression.eval(getEnvironment(env));
    var elseVal := self.elseExpression.eval(getEnvironment(env));
    return if elseVal = thenVal then
        BTAKind::STATIC
    else
        BTAKind::DYNAMIC
    endif;
}
return BTAKind::DYNAMIC;

helper OperationCallExp::onlineBta(env : OclAny, bt : Dict(String, BTAKind)) : BTAKind {
    var offBta := self.bta(bt, context);
    if offBta = BTAKind::MAYBE then {
        if self.referredOperationoclIsKindOf(Helper) then {
            var newBt : Dict(String, BTAKind) := Dict();
            bt->keys()->forEach(k) {
                newBt->put(k, bt->get(k));
            }; 
            var params := self.referredOperation.oclAsType(Helper).eParameters->asOrderedSet();
            var callArgs := self.argument;
            var i := 1;
            var n := params->size();
            while (i < n) {
                newBt->put(params->at(i).name, callArgs->at(i).onlineBta(env, bt));
                i := i + 1;
            }
            newBt->put(‘self’, self.source.onlineBta(env, bt));
            var bodyBt := self.referredOperation.oclAsType(Helper).body.content->bta(newBt, context);
            if bodyBt->exists(b | b <> BTAKind::STATIC) then {
                return BTAKind::DYNAMIC;
            } 
            endif;
            return BTAKind::STATIC;
        }
    }
    else {
        return BTAKind::DYNAMIC;
    }
}
5463 )
5464     endif;
5465 
5466     endif;
5467     return offBta;
5468 }

5471 /////////////////////////////////////////////////////////////////////////////////////////////////// 5472 // ASTNode::eval 5473 // 5474 /////////////////////////////////////////////////////////////////////////////////////////////////// 5477 helper ASTNode::eval(env : OclAny, bt : Dict(String, BTAKind)) : OclAny {
5478     return null;
5479 }

5482 /////////////////////////////////////////////////////////////////////////////////////////////////// 5483 // 5484 // OCLExpression::eval 5485 // 5486 /////////////////////////////////////////////////////////////////////////////////////////////////// 5488 helper OCLExpression::eval(env : OclAny, bt : Dict(String, BTAKind)) : OclAny {
5489     return self.eval(getEnvironment(env));
5490 }

5494 /////////////////////////////////////////////////////////////////////////////////////////////////// 5495 // 5496 // ASTNode::sideEffectsEval 5497 // 5498 /////////////////////////////////////////////////////////////////////////////////////////////////// 5500 helper ASTNode::sideEffectsEval(env : OclAny, bt : Dict(String, BTAKind), inout effects : EObject) : OclAny {
5501     return self.eval(env, bt);
5502 }

5505 /////////////////////////////////////////////////////////////////////////////////////////////////// 5506 // 5507 // WhileExp::sideEffectsEval 5508 // 5509 /////////////////////////////////////////////////////////////////////////////////////////////////// 5511 helper WhileExp::sideEffectsEval(env : OclAny, bt : Dict(String, BTAKind), inout effects : EObject) : OclAny {
5512     var oldEnv := getEnvironment(env);
5513     var newEnv := oldEnv.copy();
5514     var body :=
5515     if (self.body.oclIsKindOf(BlockExp)) then
5516         self.body.oclAsType(BlockExp).body
5517     else
5518         OrderedSet(self.body)
5519     endif
5520 ;
5521     var vars := body[AssignExp]->
5522     xcollect(exp|exp.left)->
5523     xselect|o|o.oclIsKindOf(VariableExp)->
oclAsType(VariableExp)->
  xcollect(v|v.referredVariable.oclAsType(Variable).name)

self.eval(newEnv);

var assignments : OrderedSet(OclAny) := OrderedSet();

vars->forEach(v) {
  var newVal := newEnv.get(v).value();
  var oldValue := oldEnv.get(v).value();
  if (newVal <> oldValue) then {
    assignments += object AssignExp {
      left := object VariableExp {
        referredVariable := object Variable {
          name := v;
        };
        name := referredVariable.oclAsType(Variable).name;
      };
      value := newVal.makeExp();
    };
  }
}

setCollectionWrapper(effects, assignments);

return null;

// ForExp::sideEffectsEval
helper ForExp::sideEffectsEval(env : OclAny, bt : Dict(String, BTAKind), inout effects : EObject) : OclAny {
  var oldEnv := getEnvironment(env);
  var newEnv := oldEnv.copy();
  var body :=
    if (self.body.oclIsKindOf(BlockExp)) then
      self.body.oclAsType(BlockExp).body
    else
      OrderedSet(self.body)
    endif

  var vars := body[AssignExp]->
  xcollect(exp|exp.left)->
  xselect(o|o.oclIsKindOf(VariableExp))->
  oclAsType(VariableExp)->
  xcollect(v|v.referredVariable.oclAsType(Variable).name)
  ;
  var res := self.eval(newEnv);

  var assignments : OrderedSet(OclAny) := OrderedSet();
  vars->forEach(v) {
    var newVal := newEnv.get(v).value();
    var oldValue := oldEnv.get(v).value();
    if (newVal <> oldValue) then {
      assignments += object AssignExp {
        left := object VariableExp {
          referredVariable := object Variable {
name := v;

name := referredVariable.oclAsType(Variable).name;

value := newVal.makeExp();

setCollectionWrapper(effects, assignments);

return res;

helper IfExp::eval(env : OclAny, bt : Dict(String, BTAKind)) : OclAny {
if (self.condition.onlineBta(env, bt) = BTAKind::STATIC) then {
if self.condition.eval(getEnvironment(env)).oclAsType(Boolean) then {
  return self.thenExpression.eval(env, bt);
else {
  return self.thenExpression;
}
}
else {
  if self.elseExpression.onlineBta(env, bt) = BTAKind::STATIC then {
    return self.elseExpression.eval(env, bt);
  } else {
    return self.elseExpression;
  }
}
}
else {
  if (self.thenExpression.onlineBta(env, bt) = BTAKind::STATIC and
  self.elseExpression.onlineBta(env, bt) = BTAKind::STATIC) then {
    var thenVal := self.thenExpression.eval(env, bt);
    var elseVal := self.elseExpression.eval(env, bt);
    if (elseVal = thenVal) then {
      return thenVal;
    } else {
      return self;
    }
  } else {
    return self;
  }
}
}
helper ASTNode::reduce(env : OclAny, bt : Dict(String, BTKind)) : OrderedSet(OCLExpression) {
  return null;
}

helper OCLExpression::reduce(env : OclAny, bt : Dict(String, BTKind)) : OrderedSet(OCLExpression) {
  var rewrite : OrderedSet(OCLExpression) := OrderedSet(self);
  return rewrite;
}

helper WhileExp::reduce(env : OclAny, bt : Dict(String, BTKind)) : OrderedSet(OCLExpression) {
  var rewrite : OrderedSet(OCLExpression) := OrderedSet();
  var sideEffects := createCollectionWrapper(rewrite);
  self.sideEffectsEval(env, bt, sideEffects);
  sideEffects.collection()->forEach(e) {
    rewrite += e.oclAsType(OCLExpression);
  };
  return rewrite;
}

helper BlockExp::reduce(env : OclAny, bt : Dict(String, BTKind)) : OrderedSet(OCLExpression) {
  var rewrite : OrderedSet(OCLExpression) := OrderedSet();
  self.body->forEach(statement) {
    rewrite += statement.reduce(env, bt);
  };
  return rewrite;
}
helper BlockExp::mixReduce(env : OclAny, bt : Dict(String, BTKind)) : OrderedSet(OCLExpression) {
    var rewrite : OrderedSet(OCLExpression) := OrderedSet();
    self.body->forEach(statement) {
        rewrite += statement.mixReduce(env, bt);}
    return rewrite;
}

helper IfExp::reduce(env : OclAny, bt : Dict(String, BTKind)) : OrderedSet(OCLExpression) {
    var rewrite : OrderedSet(OCLExpression) := OrderedSet();
    var sideEffects := createCollectionWrapper(rewrite);
    if self.condition.eval(env, bt).oclAsType(Boolean) then {
        rewrite += self.thenExpression.reduce(env, bt);
    } else {
        if self.elseExpression <> null then {
            rewrite += self.elseExpression.reduce(env, bt);
        } endif;
    } endif;
    return rewrite;
}

helper IfExp::mixReduce(env : OclAny, bt : Dict(String, BTKind)) : OrderedSet(OCLExpression) {
    var rewrite : OrderedSet(OCLExpression) := OrderedSet();
    var sideEffects := createCollectionWrapper(rewrite);
    if self.condition.eval(env, bt).oclAsType(Boolean) then {
        rewrite += self.thenExpression.mixReduce(env, bt);
    } else {
        rewrite += self.elseExpression.mixReduce(env, bt);
    } endif;
    return rewrite;
}

helper VariableInitExp::reduce(env : OclAny, bt : Dict(String, BTKind)) : OrderedSet(OCLExpression) {
    var rewrite : OrderedSet(OCLExpression) := OrderedSet();
var sideEffects := createCollectionWrapper(rewrite);
var rvalue := self.referredVariable.initExpression.sideEffectsEval(env, bt, sideEffects);
sideEffects.collection() -> forEach(e) {
    rewrite += e.oclAsType(OCLExpression);
}
var newInitExp := self.deepclone().oclAsType(VariableInitExp);
switch {
    case (rvalue.oclIsKindOf(EObject)) {
        newInitExp.referredVariable.initExpression := rvalue.oclAsType(EObject).makeExp();
    }
    else {
        newInitExp.referredVariable.initExpression := rvalue.makeExp();
    }
};
rewrite += newInitExp;
getEnvironment(env).put(self.referredVariable.name, createFrame(rvalue));
return rewrite;

helper ReturnExp::mixReduce(env : OclAny, bt : Dict(String, BTAKind)) : OrderedSet(OCLExpression) {
    var rewrite : OrderedSet(OCLExpression) := OrderedSet();
    var valMix := self.value.mixReduce(env, bt);
    var newRet := self.deepclone().oclAsType(ReturnExp);
    newRet.value := valMix->last();
    if (valMix->size() > 1) then {
        rewrite += valMix->subOrderedSet(1, valMix->size() - 1);
    }
    endif
    rewrite += newRet;
    return rewrite;
}

helper ASTNode::mixReduce(env : OclAny, bt : Dict(String, BTAKind)) : OrderedSet(OCLExpression) {
    return null;
}

helper OCLExpression::mixReduce
helper OCLExpression::mixReduce(env : OclAny, bt : Dict(String, BTAKind)) : OrderedSet(OCLExpression) {
    return OrderedSet(self);
}

helper ASTNode::polyMixReduce

helper ASTNode::polyMixReduce(env : OclAny, bt : Dict(String, BTAKind), context : EOperation) : OrderedSet(OCLExpression) {
    return null;
}

helper OCLExpression::polyMixReduce(env : OclAny, bt : Dict(String, BTAKind), context : EOperation) : OrderedSet(OCLExpression) {
    return self.mixReduce(env, bt);
}

helper VariableInitExp::mixReduce(env : OclAny, bt : Dict(String, BTAKind)) : OrderedSet(OCLExpression) {
    var rewrite : OrderedSet(OCLExpression) := OrderedSet();
    var newVarInit := self.deepclone().oclAsType(VariableInitExp);
    var staticEvalRewrite := self.reduce(env, bt);
    var staticResult := self.eval(env, bt);
    var mixed := self.referredVariable.initExpression.mixReduce(env, bt);
    var mixInit := mixed->last();
    var simplified := mixInit.toArithmeticExp().simplify().toOperationCall();
    var mergeExp := simplified.merge(staticResult, env, bt);
    var newInitExp : OCLExpression := mergeExp->last();
    if mixed->size() > 1 then {
        rewrite += mixed->subOrderedSet(1, mixed->size() - 1);
    }
    endif;
    if mergeExp->size() > 1 then {
        rewrite += mergeExp->subOrderedSet(1, mergeExp->size() - 1);
    }
    endif;
    if newInitExp <> simplified then {
        var subVar := object VariableInitExp {
            referredVariable := object Variable {
                name := self.subName();
                initExpression := newInitExp.oclAsType(OperationCallExp).source;
            };
        };
    }
name := referredVariable.name;
});

bt->put(subVar.referredVariable.name, BTAKind::DYNAMIC);
subVar.eval(env, bt);
rewrite += subVar;

var initWithSubVar := newInitExp.oclAsType(OperationCallExp);

var referredVariable := subVar.referredVariable.deepclone().oclAsType(Variable);
name := referredVariable.name;
eType := referredVariable.initExpression.getType().oclAsType(EClassifier);

newVarInit.referredVariable.initExpression := initWithSubVar.toArithmaticExp().simplify().toOperationCall();
}
else {
newVarInit.referredVariable.initExpression := newInitExp;
}
endif;
rewrite += newVarInit;
bt->put(self.referredVariable.name, BTAKind::DYNAMIC);

return rewrite;
}
5948 name := 'oclAsType';
5949 eoclAsTypeEObject();
5950 eType := staticResultLookupExp.eType.deepclone().oclAsType(EClassifier);
5951 argument := OrderedSet();
5952 argument += object TypeExp {
5953 referredType := self.referredVariable.eType.getElementType();
5954 }
5955 }
5956 }
5957 endif;
5958 var mixed := self.referredVariable.initExpression.polyMixReduce(env, bt, context);
5959 var mixedInit := mixed->last();
5960 var simplified := mixedInit.toArithmeticExp().simplify().toOperationCall().oclAsType(ocl::ecore::OCLExpression);
5961 var mergeExp := simplified.merge(staticResultLookupExp, env, bt);
5962 var newInitExp : OCLExpression := mergeExp->last();
5963 if merged->size() > 1 then {
5964 rewrite += merged->subOrderedSet(1, merged->size() - 1);
5965 }
5966 endif;
5967 if mergeExp->size() > 1 then {
5968 rewrite += mergeExp->subOrderedSet(1, mergeExp->size() - 1);
5969 }
5970 } 5971 endif;
5972 if newInitExp <> simplified then {
5973 var leftVar := self.referredVariable.deepclone().oclAsType(Variable);
5974 var subVarName := leftVar.subName();
5975 var subVarInit := newInitExp.oclAsType(OperationCallExp).source.oclAsType(OCLExpression);
5976 var subVar : OCLExpression;
5977 if getEnvironment(env).hasKey(subVarName) then {
5978 var subLeftVar := object Variable {
5979 name := subVarName;
5980 initExpression := subVarInit;
5981 }; 5982 subVar := object AssignExp {
5983 left := object VariableExp {
5984 name := subVarName;
5985 referredVariable := subLeftVar;
5986 }; 5987 value := OrderedSet(subVarInit);
5988 };
5989 initWithSubVar := object VariableInitExp {
5990 referredVariable := subLeftVar;
5991 name := subVarName;
5992 eType := referredVariable.oclAsType(Variable).initExpression.getType().oclAsType(EClassifier);
5993 };
5994 } 5995 else {
5996 var subRefVar := object Variable {
5997 name := subVarName;
5998 initExpression := subVarInit;
5999 }; 6000 subVar := object VariableInitExp {
6001 referredVariable := subRefVar;
6002 name := subVarName;
6003 };
6004 initWithSubVar := object VariableExp {
6005 referredVariable := subRefVar;
6006 name := subVarName;
6007 };
6008 }
eType := referredVariable.oclAsType(Variable).initExpression.getType().oclAsType(EClassifier);
}
endif;

bt->put(subVar.name, BTAKind::DYNAMIC);
subVar.eval(env, bt);
rewrite += subVar;
newVarInit.referredVariable.initExpression := initWithSubVar.toArithmaticExp().simplify().toOperationCall();
}
else {
newVarInit.referredVariable.initExpression := newInitExp;
}
endif;
newVarInit.referredVariable.eType := newVarInit.referredVariable.initExpression.getType().oclAsType(EClassifier);
rewrite += newVarInit;
bt->put(self.referredVariable.name, BTAKind::DYNAMIC);
return rewrite;

helper ASTNode::fullName() : String {
return null;
}

helper OCLExpression::fullName() : String {
return self.name;
}

helper PropertyCallExp::fullName() : String {
var prop := self.referredProperty.oclAsType(EStructuralFeature).name;
return "\"" + self.source.getName() + "\"" + prop;
}

helper VariableExp::fullName() : String {
return self.referredVariable.fullName();
}
Variable::fullName() : String {
    return '__' + self.name;
}

helper ASTNode::subName() : String {
    return self.fullName() + 'Sub';
}

helper OCLExpression::subName() : String {
    return self.fullName() + 'Sub';
}

AssignExp::mixReduce(env : OclAny, bt : Dict(String, BTAKind)) : OrderedSet(OCLExpression) {
    var rewrite : OrderedSet(OCLExpression) := OrderedSet();
    var newAssign := self.deepclone().oclAsType(AssignExp);

    var staticResult := self.eval(env, bt);
    var mixed := OrderedSet(OCLExpression) := self.value->first().mixReduce(env, bt);
    var mixedValue := mixed->last();
    var simplified := mixedValue.toArithmeticExp().simplify().toOperationCall().oclAsType(ocl::ecore::OCLExpression);
    var mergeExp := simplified.merge(staticResult, env, bt);
    var newValue := mergeExp->last();
    if newValue <> simplified then {
        var leftVar := self.left.deepclone().oclAsType(OCLExpression).reduce(env, bt)->last();
        }
var subVarName := leftVar.subName();
var subVarValue := newValue.oclAsType(OperationCallExp).source.oclAsType(OCLExpression);
var subVar : OCLExpression;
var valueWithSubVar := newValue.oclAsType(OperationCallExp);
if getEnvironment(env).hasKey(subVarName) then {
var subLeftVar := object VariableExp {
  name := subVarName;
  referredVariable := object Variable {
    name := subVarName;
  };
};
subVar := object AssignExp {
  left := subLeftVar;
  value := OrderedSet{subVarValue};
};
valueWithSubVar.source := object VariableExp {
  referredVariable := subLeftVar.referredVariable;
  name := referredVariable.oclAsType(Variable).name;
  eType := new CollectionType();
};
}
else {
var subRefVar := object Variable {
  name := subVarName;
  initExpression := subVarValue;
};
subVar := object VariableInitExp {
  referredVariable := subRefVar;
  name := subVarName;
};
valueWithSubVar.source := object VariableExp {
  referredVariable := subRefVar;
  name := subVarName;
  eType := new CollectionType();
};
}
}
}
}
}
bt->put(subVar.name, BTAKind::DYNAMIC);
subVar.eval(env, bt);
rewrite += subVar;
newAssign.value := OrderedSet(valueWithSubVar.toArithmeticExp().simplify().toOperationCall().oclAsType(OCLExpression))
};
}
else {
newAssign.value := OrderedSet(simplified);
}
}
}
return rewrite;
return rewrite;
return rewrite;
return rewrite;
return rewrite;
return rewrite;
/////////////////////////////////////////////////////////////////////////////////////////////
// AssignExp::polyMixReduce
/////////////////////////////////////////////////////////////////////////////////////////////
helper AssignExp::polyMixReduce(env : OclAny, bt : Dict(String, BTKind), context : EOperation) : OrderedSet(OCLExpression) {
var rewrite : OrderedSet(OCLExpression) := OrderedSet();
return rewrite;
var newAssign := self.deepclone().oclAsType(AssignExp);

// var staticEvalRewrite := self.reduce(env, bt);
var staticResult := self.eval(env, bt);
var staticResultName := '\_\_' + context.name + self.left.fullName();
var staticResultCache := staticResultName + 'Cache';
var selfPath := getEnvironment(env).get('self').value().oclAsType(EOcl::ecore::OCLExpression).path();
memozieVariable(staticResultCache, selfPath, staticResult);
var staticResultLookupExp := makeLookupExp(staticResultCache);

var mixed : OrderedSet(OCLExpression) := self.value->first().polyMixReduce(env, bt, context);
var mixedValue := mixed->last();
var simplified := mixedValue.toArithmeticExp().simplify().toOperationCall().oclAsType(ocl::ecore::OCLExpression);
var mergeExp := simplified.merge(staticResultLookupExp, env, bt);
var newValue := mergeExp->last();

if mixed->size() > 1 then {
  rewrite += mixed->subOrderedSet(1, mixed->size() - 1);
}
endif;
if mergeExp->size() > 1 then {
  rewrite += mergeExp->subOrderedSet(1, mergeExp->size() - 1);
}
endif;
if newValue <> simplified then {
  var leftVar := self.left.deepclone().oclAsType(OCLExpression).reduce(env, bt)->last();
  var subVarName := leftVar.subName();
  var subVarValue := newValue.oclAsType(OperationCallExp).source.oclAsType(OCLExpression);
  var subVar := OCLExpression;
  var valueWithSubVar := newValue.oclAsType(OperationCallExp);
  if getEnvironment(env).hasKey(subVarName) then {
    var subLeftVar := object VariableExp {
      name := subVarName;
      referredVariable := object Variable {
        name := subVarName;
        };
    };
    subVar := object AssignExp {
      left := subLeftVar;
      value := OrderedSet(subVarValue);
    };
    valueWithSubVar.source := object VariableExp {
      referredVariable := subLeftVar.referredVariable;
      name := referredVariable.oclAsType(Variable).name;
      eType := new CollectionType();
    };
  } else {
    var subRefVar := object Variable {
      name := subVarName;
      initExpression := subVarValue;
      };
    subVar := object VariableInitExp {
      referredVariable := subRefVar;
      name := subVarName;
    };
    valueWithSubVar.source := object VariableExp {
      referredVariable := subRefVar;
      name := subVarName;
    };
  }
}
eType := new CollectionType();
}
}
endif;

bt->put(subVar.name, BTAKind::DYNAMIC);
subVar.eval(env, bt);
rewrite += subVar;
newAssign.value := OrderedSet(valueWithSubVar.toArithmaticExp().simplify().toOperationCall().oclAsType(OCLExpression));
}
else {
newAssign.value := OrderedSet(simplified);
}
endif;
rewrite += newAssign;
return rewrite;
}

// AssignExp::reduce

helper AssignExp::reduce(env : OclAny, bt : Dict(String, BTAKind)) : OrderedSet(OCLExpression) {
var rewrite : OrderedSet(OCLExpression) := OrderedSet();
var rightBt := self.value->first().onlineBta(env, bt);
var newAssignment := object AssignExp {
  left := self.left;
  value := self.value->first().eval(env, bt).makeExp();
};
rewrite += newAssignment;
return rewrite;
}

// ReturnExp::reduce

helper ReturnExp::reduce(env : OclAny, bt : Dict(String, BTAKind)) : OrderedSet(OCLExpression) {
var rewrite : OrderedSet(OCLExpression) := OrderedSet();
var newRet := object ReturnExp {
  value := self.value.eval(env, bt).makeExp();
};
rewrite += newRet;
return rewrite;
}

// ConstructorBody::mixReduce

helper ConstructorBody::mixReduce(env : OclAny, bt : Dict(String, BTAKind)) : OrderedSet(OCLExpression) {
var newBody : OrderedSet(OCLExpression) := OrderedSet();
```java
self.content->forEach(expr) {
  newBody += expr.mixReduce(env, bt);
}
return newBody;
```
```
6370 };
6371 rewrite += newObjectExp;
6372 return rewrite;
6373 }
6374
6375 //////////////////////////////////////////////////////////////////////////////////////////
6376 //
6377 // LogExp::mixReduce
6378 //
6379 //////////////////////////////////////////////////////////////////////////////////////////
6380 helper LogExp::mixReduce(env : OclAny, bt : Dict(String, BTAKind)) : OrderedSet(OCLExpression) {
6381 var rewrite : OrderedSet(OCLExpression) := OrderedSet();
6382 var args : OrderedSet(ocl::expressions::OCLExpression) := OrderedSet();
6383 self.argument->forEach(a) {
6384 var argBt := a.onlineBta(env, bt);
6385 if (argBt = BTAKind::STATIC) then {
6386 args += a.eval(env, bt).makeExp();
6387 } else {
6388 args += a;
6389 } endif;
6390 }
6391 var newLog := object LogExp {
6392 argument := args;
6393 }
6394 rewrite += newLog;
6395 return rewrite;
6396 }
6397
6398 //////////////////////////////////////////////////////////////////////////////////////////
6399 //
6400 // MappingCallExp::mixReduce
6401 //
6402 //////////////////////////////////////////////////////////////////////////////////////////
6403 helper MappingCallExp::mixReduce(env : OclAny, bt : Dict(String, BTAKind)) : OrderedSet(OCLExpression) {
6404 var rewrite : OrderedSet(OCLExpression) := OrderedSet();
6405 var srcBT : BTAKind := self.source.onlineBta(env, bt);
6406 var argsBT := self.argument->onlineBta(env, bt);
6407 if argsBT->forAll(t|t = BTAKind::DYNAMIC) and srcBT = BTAKind::DYNAMIC then {
6408 var srcRedex := self.source.mixReduce(env, bt);
6409 var newCall := self.deepclone().oclAsType(MappingCallExp);
6410 newCall.source := srcRedex->last().deepclone().oclAsType(OCLExpression);
6411 var staticResult := self.eval(env, bt);
6412 var mergeExp := newCall.merge(staticResult, env, bt);
6413 var s := srcRedex->excluding(srcRedex->last())->asOrderedSet();
6414 s += mergeExp;
6415 rewrite += s;
6416 var newBt : Dict(String, BTAKind) := Dict();
6417 bt->keys()->forEach(k) {
6418 newBt->put(k, bt->get(k));
6419 }
6420 var params := self.referredOperation.oclAsType(MappingOperation).eParameters->asOrderedSet();
6421
287
var callArgs := self.argument;

var funcEnv := createEnvironment();
funcEnv.parentEnv := getEnvironment(env);

newBt-&gt;put('self', srcBT);
var srcVal := self.source.eval(env, bt);
funcEnv.put('self', createFrame(srcVal));
if (srcBT = BTAKind::STATIC) then {
  funcEnv.put('self', createFrame(srcVal));
  newCall.source := srcVal.makeExp();
}

var newCallArgs : OrderedSet(ocl::expressions::OCLExpression) := OrderedSet();
var newParams : OrderedSet(EParameter) := OrderedSet();
i := 1;
var n := params-&gt;size();
while (i <= n) {
  var arg := callArgs-&gt;at(i);
  var param := params-&gt;at(i);
  var pbt := arg.onlineBta(env, bt);
  newBt-&gt;put(param.name, pbt);
  if (pbt = BTAKind::STATIC) then {
    var val := arg.eval(env, bt);
    funcEnv.put(param.name, createFrame(val));
    if (not self.referredOperation.oclIsTypeOf(MappingOperation)) then {
      newCallArgs += val.makeExp();
    }
  } else {
    newCallArgs += arg.deepclone().oclAsType(OCLExpression);
    newParams += param.deepclone().oclAsType(EParameter);
  }
  i := i + 1;
}
newCall.argument := newCallArgs;
var isFirstMix := not mixCount-&gt;hasKey(self.referredOperation.oclAsType(MappingOperation).name);
var mixedOp := self.referredOperation.oclAsType(MappingOperation).mix(funcEnv, newBt);
mixedOp.eParameters := newParams;
newCall.referredOperation := mixedOp.oclAsType(EObject);
if isFirstMix then {
  newOperations += mixedOp;
}
else {
  var refOp := newCall.referredOperation.oclAsType(MappingOperation);
  refOp.name := refOp.name + '__mix1';
  newCall.referredOperation := refOp.oclAsType(EObject);
}
rewrite += newCall;
}

endif;
```cpp
return rewrite;
```
if srcMix->size() > 1 then {
    rewrite = srcMix->subOrderedSet(1, srcMix->size() - 1);
}
endif;
newCall.source := srcMix->last();
endif;
}
endif;

var newCallArgs : OrderedSet(ocl::expressions::OCLExpression) := OrderedSet();
var newParams : OrderedSet(EParameter) := OrderedSet();

var i := 1;
var n := params->size();
while (i <= n) {
    var arg := callArgs->at(i);
    var param := params->at(i);
    var pbt := arg.onlineBta(env, bt);
    newBt->put(param.name, pbt);
    if (pbt = BTAKind::STATIC) then {
        var val := arg.eval(env, bt);
        funcEnv.put(param.name, createFrame(val));
        if (not self.referredOperation.oclIsTypeOf(Helper)) then {
            newCallArgs += val.makeExp();
        }
    } else {
        var argMix := arg.mixReduce(env, bt);
        if argMix->size() > 1 then {
            rewrite = argMix->subOrderedSet(1, argMix->size() - 1);
        }
    }
    endif;
    newCallArgs += argMix->last(); //arg.deepclone().oclAsType(OCLExpression);
    newParams += param.deepclone().oclAsType(EParameter);
}
endif;
i := i + 1;
}
endif;
newCall.argument := newCallArgs;

if (not self.referredOperation.oclIsTypeOf(Helper)) then {
    rewrite := newCall;
    return rewrite;
}
endif;
var mixedOp := self.referredOperation.oclAsType(Helper).mix(funcEnv, newBt);
newOperations := newParams;
newOperations += mixedOp;
newCall.referredOperation := mixedOp.oclAsTypeEObject;
rewrite := newCall;
return rewrite;
}
helper OperationCallExp::polyMixReduce(env : OclAny, bt : Dict(String, BTAKind), context : EOperation) : OrderedSet(OCLExpression) {
  var rewrite : OrderedSet(OCLExpression) := OrderedSet();
  var srcBT : BTAKind;
  var hasSource := self.source <> null or not self.source.oclIsInvalid();
  if hasSource then {
    srcBT := self.source.onlineBta(env, bt);
  } else {
    srcBT := BTAKind::STATIC;
  }
  var srcRedex := self.source.polyMixReduce(env, bt, context);
  var newCall := self.deepclone().oclAsType(OperationCallExp);
  newCall.source := srcRedex->last().deepclone().oclAsType(OCLExpression);
  if srcRedex->size() > 1 then {
    rewrite += srcRedex->subOrderedSet(1, srcRedex->size() - 1);
  }
  var newBt : Dict(String, BTAKind) := Dict();
  bt->keys()->forEach(k) {
    newBt->put(k, bt->get(k));
  }
  var params := self.referredOperationoclAsType(EOperation).eParameters->asOrderedSet();
  var callArgs := self.argument;
  var funcEnv := createEnvironment();
  funcEnv.parentEnv := getEnvironment(env);
  var newCall := self.deepclone().oclAsType(OperationCallExp);
  if (hasSource) then {
    newBt->put('self', srcBT);
    if (srcBT = BTAKind::STATIC) then {
      var srcVal := self.source.eval(env, bt);
      funcEnv.put('self', createFrame(srcVal));
      newCall.source := srcVal.makeExp();
    }
  }
  var newCallArgs : OrderedSet(ocl::expressions::OCLExpression) := OrderedSet();
  return rewrite;
}
var newParams : OrderedSet(EParameter) := OrderedSet();

var i := 1;
var n := params->size();
while (i <= n) {
    var arg := callArgs->at(i);
    var param := params->at(i);
    var pbt := arg.onlineBta(env, bt);
    newBt->put(param.name, pbt);
    if (pbt = BTAKind::STATIC) then {
        var val := arg.eval(env, bt);
        funcEnv.put(param.name, createFrame(val));
        if (not self.referredOperation.oclIsTypeOf(Helper)) then {
            newCallArgs += val.makeExp();
        }
    } else {
        newCallArgs += arg.deepclone().oclAsType(OCLExpression);
        newParams += param.deepclone().oclAsType(EParameter);
    }
    i := i + 1;
}
newCall.argument := newCallArgs;

if (not self.referredOperation.oclIsTypeOf(Helper)) then {
    rewrite += newCall;
    return rewrite;
} else {
    var srcMix := self.source.mixReduce(env, bt);
    if srcMix->size() > 1 then {
        rewrite += srcMix->subOrderedSet(1, srcMix->size() - 1);
    }
    newOperations += mixedOp;
    newCall.referredOperation := mixedOp.oclAsType(EObject);
    rewrite += newCall;
    return rewrite;
}

helper ImperativeIterateExp::mixReduce(env : OclAny, bt : Dict(String, BTAKind)) : OrderedSet(OCLExpression) {
    var rewrite : OrderedSet(OCLExpression) := OrderedSet();
    var srcVariable : VariableInitExp;
    if self.source.oclAsType(ETypedElement).eType.oclIsKindOf(CollectionType) then {
        var srcMix := self.source.mixReduce(env, bt);
        if srcMix->size() > 1 then {
            rewrite += srcMix->subOrderedSet(1, srcMix->size() - 1);
        }
        end;
        srcVariable := object VariableInitExp {
            referredVariable := object Variable {
name := newTemp();

initExpression := srcMix->last().deepClone().oclAsType(OCLExpression);
}

name := referredVariable.name;
}

}

else {
srcVariable := self.makeSourceVariable(newTemp());
}
endif;

var environ := getEnvironment(env);
bt->put(srcVariable.name, BTAKind::DYNAMIC);
var srcVal := srcVariable.eval(env, bt);
rewrite += srcVariable;

var last := getCollectionWrapper(environ.get(srcVariable.referredVariable.name).value()).collection()->size();
var lastVarName := srcVariable.referredVariable.name + 'Last';
var lastVar := makeVar(lastVarName, last);
var lastVal := lastVar.eval(env, bt);
rewrite += lastVar;

var sizeVar := makeSizeVar(srcVariable.referredVariable);
bt->put(sizeVar.name, BTAKind::DYNAMIC);
var sizeVal := sizeVar.eval(environ, bt);
rewrite += sizeVar;

var subSetVar := makeSubSetVar(srcVariable.referredVariable);
bt->put(subSetVar.name, BTAKind::DYNAMIC);
var subSetVal := subSetVar.eval(env, bt);
rewrite += subSetVar;

var subSetVarExp := object VariableExp {
  referredVariable := subSetVar.referredVariable.deepClone().oclAsType(Variable);
  name := referredVariable.oclAsType(Variable).name;
};

var imperativeExp := self.deepClone().oclAsType(ImperativeIterateExp);
imperativeExp.source := subSetVarExp;
if self.name = 'xcollect' then {
  var newbt : Dict(String, BTAKind) = Dict();
  bt->keys()]->forEach(k) {
    newbt->put(k, bt->get(k));
  };
  newbt->put(self.iterator->first().getName(), BTAKind::DYNAMIC);
  var newEnv := createEnvironment(getEnvironment(env));
  var newBody : OCLExpression;
  getCollectionWrapper(self.source.eval(env, newbt)).collection()->forEach(element) {
    newEnv.put(self.iterator->first().getName(), createFrame(element));
    var bodyMix := self.body.mixReduce(newEnv, newbt);
    if bodyMix->size() > 1 then {
      rewrite += bodyMix->subOrderedSet(1, bodyMix->size() - 1);
    }
    endif;
    newBody := bodyMix->last();
  };
  // imperativeExp.body := newBody;
  var union := object OperationCallExp {
    source := imperativeExp;
referredOperation := object EOperation {
  name := 'union';
}.oclAsType(EOperation);
argument := OrderedSet();
argument += object ImperativeIterateExp {
  name := self.name;
  source := object OperationCallExp {
    referredOperation := object EOperation {
      name := 'subOrderedSet';
    }.oclAsType(EOperation);
    source := self.source.deepclone().oclAsType(OCLExpression);
    argument := OrderedSet();
    argument += 1.makeExp();
    argument += object VariableExp {
      referredVariable := lastVar.referredVariable;
      name := lastVar.referredVariable.name;
    };
  };
  iterator := self.iterator;
  body := newBody;
}
rewrite += union;
}
return rewrite;
}

helper ImperativeIterateExp::polyMixReduce(env : OclAny, bt : Dict(String, BTAKind), context : EOperation) : OrderedSet(OCLExpression) {
  var rewrite : OrderedSet(OCLExpression) := OrderedSet();
  var selfPath := getEnvironment(env).get('self').value().oclAsType(EObject).path();
  var srcVariable : VariableInitExp;
  var srcVarName := '__' + context.name + newTemp();
  if self.source.oclAsType(ETypedElement).eType.oclIsKindOf(CollectionType) then {
    var srcMix := self.source.polyMixReduce(env, bt, context);
    if srcMix->size() > 1 then {
      rewrite += srcMix->subOrderedSet(1, srcMix->size() - 1);
    }
    endif;
  srcVariable := object VariableInitExp {
    referredVariable := object Variable {
      name := srcVarName;
      initExpression := srcMix->last().deepclone().oclAsType(OCLExpression);
    };
    name := referredVariable.name;
  };
  else {
    srcVariable := self.makeSourceVariable(srcVarName);
  }
  endif;
  var environ := getEnvironment(env);
bt->put(srcVariable.name, BTAKind::DYNAMIC);
var srcVal := srcVariable.eval(env, bt);
rewrite += srcVariable;

var last := getCollectionWrapper(env.get(srcVariable.referredVariable.name).value()).collection()->size();
var lastVarName := srcVarName + 'Last';
bt->put(lastVarName, BTAKind::DYNAMIC);
memoizeVariable(lastVarCache, selfPath, last);
var lookupExp := makeLookupExp(lastVarCache);
var lastVar := makeVar(lastVarName, lookupExp);
var lastVal := lastVar.eval(env, bt);
getEnvironment(env).put(lastVarName, createFrame(last));
rewrite += lastVar;

var sizeVar := makeSizeVar(srcVariable.referredVariable);
bt->put(sizeVar.name, BTAKind::DYNAMIC);
var sizeVal := sizeVar.eval(env, bt);
rewrite += sizeVar;

var subSetVar := makeSubSetVar(srcVariable.referredVariable);
bt->put(subSetVar.name, BTAKind::DYNAMIC);
var subSetVal := subSetVar.eval(env, bt);
rewrite += subSetVar;

var imperativeExp := self.deepclone().oclAsType(ImperativeIterateExp);
imperativeExp.source := subSetVarExp;
if self.name = 'xcollect' then {
  var newBt : Dict(String, BTAKind) := Dict();
  bt->keys()->forEach(k) {
    newBt->put(k, bt->get(k));
  }
  newBt->put(self.iterator->first().getName(), BTAKind::DYNAMIC);
  var newEnv := createEnvironment(getEnvironment(env));
  var newBody : OCLExpression;
  getCollectionWrapper(self.source.eval(env, newBt)).collection()->forEach(element) {
    var bodyMix := self.body.mixReduce(newEnv, newBt);
    if bodyMix->size() > 1 then {
      rewrite += bodyMix->subOrderedSet(1, bodyMix->size() - 1);
    }
    end;
  newBody := bodyMix->last();
  }
  imperativeExp.body := newBody;
  }
  rewrite += imperativeExp;
  return rewrite;
}
helper ASTNode::merge(val : OclAny, env : OclAny, bt : Dict(String, BTAKind)) : OrderedSet(OCLExpression) {
    return null;
}

helper OCLExpression::merge(val : OclAny, env : OclAny, bt : Dict(String, BTAKind)) : OrderedSet(OCLExpression) {
    return OrderedSet(self);
}

helper ASTNode::getSub(env : OclAny, bt : Dict(String, BTAKind)) : OCLExpression {
    return null;
}

helper OCLExpression::getSub(env : OclAny, bt : Dict(String, BTAKind)) : OCLExpression {
    return self;
}

helper VariableExp::getSub(env : OclAny, bt : Dict(String, BTAKind)) : OCLExpression {
    var subName := self.subName();
    if getEnvironment(env).hasKey(subName) then {
        var newVar := object VariableExp {
            name := subName;
            referredVariable := object Variable {
                name := subName;
                eType := self.eType.deepclone().oclAsType(EClassifier);
            };
            eType := referredVariable.oclAsType(Variable).eType;
        };
        return newVar;
    } else {
        return self;
    }
}
helper OperationCallExp::merge(val : OclAny, env : OclAny, bt : Dict(String, BTAKind)) : OrderedSet(OCLExpression) {
  var rewrite : OrderedSet(OCLExpression) := OrderedSet();
  if self.referredOperation.oclIsKindOf(Helper) then {
    return OrderedSet(self);
  }
  var opName := self.referredOperation.oclAsType(EOperation).name;
  var subSrc := self.source.getSub(env, bt);
  var newCall : OperationCallExp;
  if subSrc <> self.source then {
    newCall := self.deepClone().oclAsType(OperationCallExp);
    newCall.source := subSrc;
  } else {
    newCall := self; // var y := 2 * xd should not be replaced by var y := 2 * xs (the static value of xd), hence returning here.
    if self.source.onlineBta(env, bt) = BTAKind::STATIC then {
      rewrite += newCall;
      return rewrite;
    }
  }
  switch {
    case (opName = 'sum')
      rewrite += mergeSum(newCall, val);
    case (opName = '+' or opName = 'x')
      { 
        var argBt := newCall.argument->first().onlineBta(env, bt);
        if (argBt = BTAKind::STATIC) then {
          newCall.argument := OrderedSet{val.makeExp()};
        }
      
      rewrite += newCall;
    case (opName = '-')
      
      newCall.argument := OrderedSet{val.makeExp()};
      
      rewrite += newCall;
    case (opName = 'indexOf')
      
      rewrite += val.makeExp();
    else
      
      rewrite += newCall;
  }
}

297
```cpp
#include <OclAny.h>

// Imperative::IterateExp::merge
helper ImperativeIterateExp::merge(val : OclAny, env : OclAny, bt : Dict(String, BTKind)) : OrderedSet(OCLExpression) {
    var rewrite : OrderedSet(OCLExpression) := OrderedSet();
    switch {
        case (self.name = 'xcollect' or self.name = 'xselect') {
            var union := object OperationCallExp {
                referredOperation := object EOperation {
                    name := 'union';
                    oclAsType(EObject);
                }.
            }.
            source := self.deepclone().oclAsType(OCLExpression);
            argument := OrderedSet();
            argument += object OperationCallExp {
                referredOperation := object EOperation {
                    name := 'as' +
                    if self.eType.oclAsType(CollectionType).kind = ocl::expressions::CollectionKind::Set then 'Set' else 'Bag'
                    endif;
                }.
            }.
            source := val.makeExp();
            rewrite += union;
        } else {
            rewrite += self;
        }
    }
    return rewrite;
}

// mergeHom
helper OperationCallExp::mergeHom(val : OclAny) : OrderedSet(OCLExpression) {
    var res : OrderedSet(OCLExpression) := OrderedSet();
    var mergeExp := object OperationCallExp {
        source := self;
        argument := val.makeExp();
        referredOperation := self.referredOperation.deepclone().oclAsType(EObject);
    };
    res += mergeExp;
    return res;
}
```
helper mergeSum(sumOp : OperationCallExp, val : OclAny) : OrderedSet(OCLExpression) {
    var res : OrderedSet(OCLExpression) := OrderedSet();
    var mergeExp := object OperationCallExp {
        source := sumOp;
        argument := val.makeExp();
        referredOperation := object EOperation {
            name := '+';
            }.oclAsType(EObject);
    };
    res += mergeExp;
    return res;
}

helper makeVar(varName : String, val : OclAny) : VariableInitExp {
    return object VariableInitExp {
        referredVariable := object Variable {
            name := varName;
            initExpression := val.makeExp();
        };
        name := referredVariable.name;
    };
}

helper makeSizeVar(variable : Variable) : VariableInitExp {
    return object VariableInitExp {
        referredVariable := object Variable {
            name := variable.name + 'Size';
            initExpression := object OperationCallExp {
                referredOperation := object EOperation {
                    name := 'size';
                }.oclAsType(EObject);
            };
            source := object VariableExp {
                referredVariable := variable;
                name := referredVariable.oclAsType(Variable).name;
                eType := new CollectionType();
            };
        };
        name := referredVariable.name;
    };
}
```cpp
helper makeSubSetVar(variable : Variable) : VariableInitExp {
    return object VariableInitExp {
        referredVariable := object Variable {
            name := variable.name + 'Sub';
        }
        initExpression := object IfExp {
            condition := object OperationCallExp {
                referredOperation := object EOperation {
                    name := '>';
                }.
oclAsType(EObject);
                source := object VariableExp {
                    referredVariable := object Variable {
                        name := variable.name + 'Size';
                    };
                    name := referredVariable.
oclAsType(Variable).name;
                }
                argument := OrderedSet{
                    object VariableExp {
                        referredVariable := object Variable {
                            name := variable.name + 'Last';
                        };
                        name := referredVariable.
oclAsType(Variable).name;
                    }
                };
                thenExpression := object OperationCallExp {
                    source := object VariableExp {
                        referredVariable := variable;
                        name := referredVariable.
oclAsType(Variable).name;
                        eType := new CollectionType();
                    };
                    referredOperation := object EOperation {
                        name := 'subOrderedSet';
                    }.
oclAsType(EObject);
                    argument += object OperationCallExp {
                        referredOperation := object EOperation {
                            name := '+';
                        }.
oclAsType(EObject);
                        source := object VariableExp {
                            referredVariable := object Variable {
                                name := variable.name + 'Last';
                            };
                            name := referredVariable.
oclAsType(Variable).name;
                        };
                        argument += object IntegerLiteralExp {
                            integerSymbol := 1;
                        };
                    };
                    argument += object VariableExp {
                        referredVariable := object Variable {
                            name := variable.name + 'Size';
                        };
                        name := referredVariable.
oclAsType(Variable).name;
                    };
                };
                elseExpression := object CollectionLiteralExp {
```
kind := ocl::expressions::CollectionKind::OrderedSet;
eType := object OrderedSetType {
  elementType := variable.eType.oclAsType(EObject);
};
name := referredVariable.name;
referredOperation := object EOperation {
    name := 'get';
    argument := OrderedSet(
        object OperationCallExp {
            source := object OperationCallExp {
                source := object VariableExp {
                    name := 'self';
                    referredVariable := object Variable {
                        name := 'self';
                    };
                    eType := object EClass {
                        name := 'EObject';
                    };
                    referredOperation := object EOperation {
                        name := 'oclAsType';
                        argument := OrderedSet(
                            object TypeExp {
                                referredType := object EClass {
                                    name := 'EObject';
                                };
                                eType := object EClass {
                                    name := 'EObject';
                                };
                                referredOperation := object EOperation {
                                    name := 'path';
                                    eType := memoizeTables->get(cacheName).eType.getElementType().oclAsType(EClassifier);
                                };
                                return lookupExp;
                            }
                        }
                    };
                }
            }
        }
    }.
    eType := memoizeTables->get(cacheName).eType.getElementType().oclAsType(EClassifier);
    return lookupExp;
}
Appendix E

Change Factorization Algorithm Implementation in C++

C++ Code

```cpp
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <iterator>
#include <map>
#include <cstdlib>
#include <ctime>

enum ChangeType {
    NOP,
    UPD,
    DEL,
    INS
};

int cost(ChangeType c) {
    switch (c) {
    case NOP:
        return 0;
    case UPD:
    case DEL:
    case INS:
        return 1;
    }
}

std::string print(ChangeType c) {
```

303
switch (c) {
    case NOP:
        return "NOP";
    case UPD:
        return "UPD";
    case DEL:
        return "DEL";
    case INS:
        return "INS";
}

struct Node {
    Node()
    {}
    Node(const std::string& v) : label(v)
    {}
    std::string label;
    std::vector<Node*> children;
};

bool operator==(const Node& n1, const Node& n2) {
    if (n1.label != n2.label) {
        return false;
    }
    return n1.children == n2.children;
}

std::string print(const Node& node) {
    std::string s = node.label;
    std::vector<Node*>::const_iterator it = node.children.begin(), itend = node.children.end();
    if (it == itend)
        return s;
    s += "{" + print(**it);
    ++it;
    for (; it != itend; ++it) {
        s += "," + print(**it);
    }
    s += "}";
    return s;
}

Node pruneLast(const Node& node) {
    Node n = node;
    n.children.pop_back();
    return n;
}

Node last(const Node& node) {
    if (node.children.empty())
        return Node();
    }
else
    return *(node.children.back());

typedef std::map<std::pair<std::string, std::string>, int> DistanceTable;
typedef std::map<std::pair<std::string, std::string>, ChangeType> ChangeTable;

DistanceTable dt;
ChangeTable ct;
int hits = 0;

int dist(const Node& s, const Node& t) {
    std::pair<std::string, std::string> key(print(s),print(t));
    DistanceTable::iterator it = dt.find(key);
    if (it != dt.end()) {
        ++hits;
        return it->second;
    }
    if (t.children.empty() && s.children.empty()) {
        if (t.label.empty()) {
            dt[key] = 0;
            ct[key] = NOP;
            return 0;
        }
        if (t.label == s.label) {
            dt[key] = 0;
            ct[key] = NOP;
            return 0;
        } else {
            ct[key] = UPD;
            dt[key] = cost(UPD);
            return cost(UPD);
        }
    } else {  // s.children.empty() 
        ct[key] = INS;
        int mc = cost(INS) + dist(s, pruneLast(t)) + dist(Node(), last(t));
        dt[key] = mc;
        return mc;
    }
    if (t.children.empty()) {
        int mc = cost(Del);
        ct[key] = DEL;
        dt[key] = mc;
        return mc;
    }
    int updCost = dist(pruneLast(s), pruneLast(t)) + dist(last(s), last(t));
    int insCost = dist(s, pruneLast(t)) + dist(Node(), last(t)) + cost(INS);
    int delCost = dist(pruneLast(s), t) + cost(DEL);
    int minCost = updCost;
    ChangeType change = UPD;
    if (insCost < updCost) {
        if (insCost < delCost) {
            // Insert case
            ...
minCost = insCost;
change = INS;
}  
else {
minCost = delCost;
change = DEL;
}
else if (delCost < updCost) {
minCost = delCost;
change = DEL;
}
ct[key] = change;
dt[key] = minCost;
return minCost;
}

int rand(int l, int h)
{
return l + (h - l + 1) * (rand() / (RAND_MAX + 1.0));
}

#define N 30
Node* randomTree()
{
    int r = rand(1, N);
    int n = r;
    Node* root = new Node("root");
    Node* p = root;
    for (int i = 0; i < n; ++i) {
        r = rand(1, N);
        for (int j = 0; j < r; ++j) {
            std::ostringstream os;
            os << "c_" << i << '_' << j;
            Node* c = new Node(os.str());
            p->children.push_back(c);
            int s = rand(1, N);
            for (int k = 0; k < s; ++k) {
                std::ostringstream os2;
                os2 << os.str() << '_' << k;
                Node* v = new Node(os2.str());
                c->children.push_back(v);
            }
        }
    }
    return root;
}

int depth(Node* node)
{
    if (node->children.empty()) {
        return 1;
    }
    else {
        std::vector<int> cdepth;
        std::transform(
            node->children.begin(),
            node->children.end(),
            cdepth,
        }
216     std::back_inserter(cdepth),
217         depth
218     );
219     return 1 + *(std::max_element(cdepth.begin(), cdepth.end()));
220 }
221 }

223 int size(Node* node)
224 {
225     int s = 1;
226     for (std::vector<Node*>::const_iterator
227         it = node->children.begin(), itend = node->children.end();
228         it != itend;
229         ++it) {
230         s += size(*it);
231     }
232     return s;
233 }

235 int size(Node* node, int level)
236 {
237     if (level == 1)
238         return 1;
239     if (level == 2)
240         return node->children.size();
241     int s = 0;
242     for (std::vector<Node*>::iterator
243         it = node->children.begin(),
244         itend = node->children.end();
245         it != itend;
246         ++it) {
247             s += size(*it, level - 1);
248         }
249     return s;
250 }

253 void destroy(Node* n)
254 {
255     if (!n->children.empty()) {
256         std::for_each(n->children.begin(), n->children.end(), destroy);
257     }
258     delete n;
259 }

261 long bigO(Node* s, Node* t)
262 {
263     int hmax = std::max(depth(s), depth(t));
264     int o = 1;
265     for (int i = 2; i <= hmax; ++i) {
266         o += (1 + size(s, i)) * (1 + size(t, i));
267     }
268     return o;
269 }

272 int main()
273 {
274     srand((unsigned)time(0));
275     rand();

std::cout << "|s|,|t|,h(s),h(t),O(d),|d|,hits,#op,time" << std::endl;

for (int i = 0; i < 100; ++i) {
    Node* s = randomTree();
    Node* t = randomTree();

    std::cout << size(s) << ',';
    std::cout << size(t) << ',';
    std::cout << depth(s) << ',';
    std::cout << depth(t) << ',';
    std::cout.flush();
    std::cout << bigO(s, t) * 3 << ',';
    std::cout.flush();
    clock_t t1 = clock();
    int delta = dist(*s, *t);
    clock_t t2 = clock();
    std::cout << dt.size() << ',' << hits << ',';
    std::cout << delta << ',' << double(t2 - t1) / CLOCKS_PER_SEC;
    std::cout << std::endl;
    destroy(s);
    destroy(t);
    dt.clear();
    ct.clear();
    hits = 0;
}
}
Bibliography


312


