Incremental Change Propagation from Software to Performance Models

by

Taghreed Arafat Mohammad Altamimi

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

© 2019

Taghreed Arafat Mohammad Altamimi
Abstract

Model-Driven Engineering (MDE) enables the automatic generation of performance models from software design models by model transformations. The performance models are used for performance analysis of the software under development, to guide the design choices from an early stage and to ensure that the system will meet its performance requirements. The software model evolves during development, so inconsistencies may appear between the software and performance models. This research aims at keeping the software and performance models synchronized.

An important concept in model transformations is the mapping between the source and target (meta)models, which can be specified in a reusable manner with the help of mapping patterns. In this research we propose a subclass of such patterns, called containment-based mapping patterns, used to map a group of containment-related source model elements into a group of target model elements. We focus on these patterns because the containment relationship is frequently found in metamodel specifications. The containment mapping patterns are applied in the context of a non-trivial transformation from UML software models extended with MARTE performance annotations into Layered Queueing Network (LQN) performance models. We show how the mapping patterns can be applied for designing the transformation rules for a batch transformation implemented in a specific language. (The complete implementation of the batch transformation was done in separate work). In this research, we extend the batch transformation to generate, beside the target model, a traceability model between the mapped source and target elements. After solving the generated LQN model with an existing solver, the performance results are fed back to the software model by following the cross-model trace links.
The next objective of the research is to design (based on the mapping patterns), implement and evaluate an incremental change propagation (ICP) approach to re-synchronize the software and performance models. During the development process, when the software model evolves, we detect the changes with the Eclipse EMF Compare tool, then incrementally propagate them to the LQN model. The proposed ICP is implemented with the Epsilon Object Language (EOL) and is evaluated by applying it to a set of case studies.
Acknowledgements

First and foremost, I would like to thank my thesis supervisor Professor Dorina C. Petriu, for the patient guidance, the encouragement and the advice she has provided throughout the five years as her student. I have been extremely fortunate to have a supervisor who cared so much about my work, and who responded to my questions and queries so promptly.

I was always dreaming of this moment, but it never came to my mind that it will come without my brother, Dr. Loay, who always believed in me. I will never forget how happy he was the moment I told him that I got my PhD admission. He was always encouraging, helping and advising me. Even during his illness, he never forgot to follow up and ask me about my research. Loay is always my role model in life. I wish he was here to see that his sister fulfilled her dream.

This journey would not have been possible without the support of my parents, who always supported me to follow my dreams. Their prayers have been lighting my path, for me to achieve my goals.

I owe thanks to a very special person, my husband Wissam for his continued and unfailing love, support and understanding during my pursuit of the Ph.D. degree, that made the completion of this thesis possible. You were always around at times I thought that it’s impossible to continue, you helped me to keep things in perspective. I greatly value your contribution and deeply appreciate your belief in me.

Special thanks to my kids Tayem and Merriam, who always were patient and gave me happiness during this long journey.

It is not a fair task to acknowledge all the people who made this Ph.D. thesis possible with
a few words. However, many thanks to my cousins, friends and everyone who supported me and surrounded me with love and care.
This work is dedicated to

My beloved brother Dr. Loay Arafat Altamimi

My Parents Arafat Altamimi, Faridah Altamimi

My dear husband Wissam Altamimi

and

My lovely kids Tayem & Merriam

My brothers, Raed & Mutaz
Table of Contents

Abstract.................................................................................................................................................. ii

Acknowledgements ................................................................................................................................. iv

Table of Contents ................................................................................................................................... vii

List of Tables .......................................................................................................................................... xi

List of Figures .......................................................................................................................................... xii

List of Code Fragments ........................................................................................................................... xvi

List of Acronyms ..................................................................................................................................... xvii

1 Chapter: Introduction .......................................................................................................................... 1
  1.1 Motivation ........................................................................................................................................ 1
  1.2 Research Objectives ......................................................................................................................... 4
  1.3 Thesis Contributions ......................................................................................................................... 5
  1.4 Thesis Contents ................................................................................................................................ 7

2 Chapter: Background and State of the Art ......................................................................................... 9
  2.1 Model Transformations .................................................................................................................... 9
    2.1.1 Transformation Languages ....................................................................................................... 9
    2.1.2 Batch versus Incremental Transformation ............................................................................... 12
  2.2 Model Driven Analysis of Non-Functional Properties (NFP) ........................................................... 20
    2.2.1 Generate Analysis Models from Software Models ................................................................. 20
      Generating Performance Models .................................................................................................... 21
      Generating Dependability Models ................................................................................................ 23
    2.3 Co-evolution of Software and Analysis Models ............................................................................ 24
    2.4 Model Comparison ......................................................................................................................... 25
    2.5 Traceability .................................................................................................................................... 25
2.6 Higher-order transformation ................................................................. 26
2.7 Patterns for Model Transformation ......................................................... 28

3 Chapter: Overview of the Proposed Integration Approach ......................... 33
3.1 Integration of NFP analysis into the MDE process .................................. 33
3.2 Examples of integration ............................................................................ 35
   3.2.1 Performance analysis roundtrip ......................................................... 35
   3.2.2 Incremental change propagation (ICP) ................................................ 36
3.3 Source Model ............................................................................................ 38
3.4 Target Model ............................................................................................ 42

4 Chapter: Containment-based Mapping Patterns ......................................... 45
4.1 Definition of mapping patterns ................................................................. 45
4.2 Specifying Containment-based Mapping Patterns in UML ....................... 46
4.3 Classifying and documenting of Containment-based mapping Patterns ......... 47
   4.3.1 Classification of Containment-based mapping ...................................... 47
   4.3.2 Documenting the mapping patterns ..................................................... 47
4.4 Mapping patterns applied between source and target metamodels. ............. 48
4.5 Mapping Patterns applied to One Metamodel ........................................... 51
4.6 Application of mapping patterns in the context of UML+MARTE and LQN ...... 57
4.7 Using Containment-based mapping patterns for Transformation Rule Design .... 63
   4.7.1 Rule based on the S-T Mapping pattern ............................................. 64
   4.7.2 Rule based on the Containment Mapping pattern .................................. 66
   4.7.3 Rule based on Virtual Container and Mutually Exclusive Containers pattern .... 69

5 Chapter: Performance analysis round trip ................................................... 72
5.1 Batch Transformation ................................................................................. 72
   5.1.1 Extending the batch transformation to generate the Traceability model .... 72
5.1.2 Automate the Transformation Process .......................................................... 74
5.2 Feedback of Performance Results to the software Model ............................ 76
5.3 Performance Analysis ....................................................................................... 77

6 Chapter: Incremental Change Propagation (ICP) ........................................ 81
6.1 Change Detection Phase .................................................................................. 81
6.2 Forms of mapping .......................................................................................... 83
6.3 Incremental Change Propagation (ICP) ......................................................... 86
   6.3.1 Assumptions .............................................................................................. 86
   6.3.2 Notation .................................................................................................... 87
   6.3.3 ICP main steps ........................................................................................ 89
   6.3.4 Potential Extension .................................................................................... 91

7 Chapter: ICP Application Cases ..................................................................... 92
7.1 Case study 1: Applying Façade Pattern ......................................................... 92
7.2 Case study 2: Changing from sequence of activities to graph .................. 105
7.3 Case study 3: Changing container (Move) .................................................... 113
7.4 Case study 4: Updating stereotype attribute .............................................. 116

8 Chapter: ICP Evaluation ................................................................................... 119
8.1 Evaluating the Correctness of ICP ............................................................... 119
8.2 Case studies .................................................................................................. 121
   8.2.1 Secure communication pattern .............................................................. 121
   8.2.2 Virtual Proxy pattern ............................................................................ 126
8.3 Evaluation of ICP Run time efficiency .......................................................... 130
   8.3.1 ICP execution time versus type of change ............................................ 130
   8.3.2 ICP execution time versus number of changes ..................................... 131
   8.3.3 ICP execution time versus the size of the model .................................. 134
9 Chapter: Conclusion ........................................................................................................... 136
  9.1 Completed work ......................................................................................................... 136
  9.2 Limitations ................................................................................................................ 138
  9.3 Directions for future research ................................................................................... 138

Appendices .......................................................................................................................... 140
  Appendix A: Summarized ICP Algorithm ....................................................................... 140
  Appendix A.1: Summarized ICP Case 1 Algorithm ......................................................... 140
  Appendix A.2: Summarized ICP Case 2 Algorithm ......................................................... 142
  Appendix A.3: Summarized ICP Case 3 Algorithm ......................................................... 143
  Appendix A.4: Summarized Algorithm for Move Difference .......................................... 143
  Appendix A.5: Summarized Algorithm for Add New Element Difference ..................... 144
  Appendix A.6: Summarized Algorithm for Delete Element Difference ......................... 144
  Appendix A.7: Summarized Algorithm for Change Difference ...................................... 144
  Appendix A.8: Summarized Algorithm for Update Stereotype attribute Difference .......... 144
  Appendix A.9: Summarized Algorithm for Update attribute difference ......................... 145

References ............................................................................................................................. 146
List of Tables

Table 2-1 Comparison Criteria .................................................................................................................. 16
Table 2-2 Comparison of incremental transformation approaches......................................................... 19
Table 4-1 High-level view of the transformation rules to transform UML+MARTE to LQN elements ................................................................................................................................. 65
Table 8-1 Execution time for ICP based on number of changes ............................................................... 133
**List of Figures**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>Performance Analysis Roundtrip</td>
<td>36</td>
</tr>
<tr>
<td>3-2</td>
<td>Incremental Change propagation (ICP)</td>
<td>38</td>
</tr>
<tr>
<td>3-3</td>
<td>Deployment diagram of Ecommerce System</td>
<td>41</td>
</tr>
<tr>
<td>3-4</td>
<td>Activity diagram of Place Order Scenario</td>
<td>41</td>
</tr>
<tr>
<td>3-5</td>
<td>Activity diagram for Browse Catalogue Scenario</td>
<td>42</td>
</tr>
<tr>
<td>3-6</td>
<td>Activity diagram for Browse Cart Scenario</td>
<td>42</td>
</tr>
<tr>
<td>3-7</td>
<td>LQN Emfatic Ecore metamodel</td>
<td>43</td>
</tr>
<tr>
<td>3-8</td>
<td>LQN Model generated from e-commerce software model</td>
<td>44</td>
</tr>
<tr>
<td>4-1</td>
<td>Collaboration elements</td>
<td>46</td>
</tr>
<tr>
<td>4-2</td>
<td>Collaboration use elements</td>
<td>47</td>
</tr>
<tr>
<td>4-3</td>
<td>Containment-based mapping patterns relationships</td>
<td>48</td>
</tr>
<tr>
<td>4-4</td>
<td>S-T Mapping Pattern: definition</td>
<td>49</td>
</tr>
<tr>
<td>4-5</td>
<td>S-T Mapping Pattern: application</td>
<td>49</td>
</tr>
<tr>
<td>4-6</td>
<td>Containment Mapping Pattern: definition</td>
<td>50</td>
</tr>
<tr>
<td>4-7</td>
<td>Containment Mapping Pattern: application</td>
<td>51</td>
</tr>
<tr>
<td>4-8</td>
<td>Simple Container Pattern: definition</td>
<td>51</td>
</tr>
<tr>
<td>4-9</td>
<td>Simple Container Pattern: application</td>
<td>52</td>
</tr>
<tr>
<td>4-10</td>
<td>Intermediate Container Pattern: definition</td>
<td>52</td>
</tr>
<tr>
<td>4-11</td>
<td>Intermediate Container Pattern: application</td>
<td>53</td>
</tr>
<tr>
<td>4-12</td>
<td>Virtual Containers Pattern: definition</td>
<td>54</td>
</tr>
<tr>
<td>4-13</td>
<td>Virtual Containers Pattern: application</td>
<td>55</td>
</tr>
<tr>
<td>4-14</td>
<td>Mutually Exclusive Containers: definition</td>
<td>55</td>
</tr>
</tbody>
</table>
Figure 7-6 Example1: LQN after creating new task F and propagating diff 4& 6 ........ 100
Figure 7-7 Example 1: LQN after propagating diff 13 .............................................. 102
Figure 7-8 Example1: LQN after adding new Synchcall and propagating difference ... 104
Figure 7-9 Example1: Updated Trace Model .......................................................... 105
Figure 7-10 Example 2: Activity partitions without graph pattern ......................... 106
Figure 7-11 Changed Example 2: Activity partition A2 with graph pattern .......... 106
Figure 7-12 Original LQN for Example 2 in Emfatic Ecore format ....................... 107
Figure 7-13 Intermediate step for adding Task-activities ..................................... 107
Figure 7-14 Example 2: LQN after Propagating Differences 1 through 5 .......... 109
Figure 7-15 Example 2: LQN after adding new Precedence ................................. 110
Figure 7-16 Example2: LQN after adding reply entry and replyactivity ............. 111
Figure 7-17 Graphical representation of LQN model before and after propagating all changes ......................................................................................................................... 114
Figure 7-18 Example 2: Updated Trace model ...................................................... 114
Figure 7-19 Deployment Diagram before moving Artifact .................................. 115
Figure 7-20 Deployment Diagram after moving Artifact F ................................. 116
Figure 7-21 Moving task F from processor D2 to processor D1 ......................... 116
Figure 7-22 Example4: Activity diagram after updating Host demand of CalloperationActionPh1 ................................................................. 117
Figure 7-23 LQN after updating Hostdemandmean of CallOperationActionph1 Activity ................................................................. 118
Figure 8-1 Activity Diagram of Browse Catalogue scenario before Secure communication pattern .................................................................................................................. 122
Figure 8-2 Activity Diagram of Browse Catalogue scenario Secure communication pattern ............................................................................................................................................. 123
Figure 8-3 LQN model after change propagating ................................................................................................................. 123
Figure 8-4 Graphical representation of LQN before applying Secure Communication pattern ................................................................. 124
Figure 8-5 Graphical representation of LQN after applying Secure Communication pattern ................................................................. 125
Figure 8-6 Activity diagram for Browse Cart scenario before Proxy pattern ........................................ 126
Figure 8-7 Activity diagram for Browse Cart scenario after Proxy pattern ................................................... 127
Figure 8-8 LQN model before applying Proxy pattern ................................................................................................................. 128
Figure 8-9 LQN model after Incremental Change Propagation ....................................................................................... 129
Figure 8-10 Comparing ICP execution time for different types of change ............................................... 133
Figure 8-11 Execution time for different size of models when apply update attribute change ......................................................................................................................... 135
Figure 8-12 Execution time for 1000 elements model size when apply create new element change ............................................................................................................................................. 135
List of Code Fragments

Code Fragment 4-1 Transformation Rule Model2Qnmodel .................................. 68
Code Fragment 4-2 Operation getDevices.................................................................. 68
Code Fragment 4-3 Operation hasStereotype.................................................................. 68
Code Fragment 4-4 Transformation Rule Device2Processor ........................................... 68
Code Fragment 5-1 Ant build file.................................................................................. 75
List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>AttributeChange</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modeling Framework</td>
</tr>
<tr>
<td>EOL</td>
<td>Epsilon Object language</td>
</tr>
<tr>
<td>Epsilon</td>
<td>Extensible Platform of Integrated Languages for Model Management</td>
</tr>
<tr>
<td>ETL</td>
<td>Epsilon Transformation Language</td>
</tr>
<tr>
<td>HOT</td>
<td>Higher-order Transformation</td>
</tr>
<tr>
<td>ICP</td>
<td>Incremental Change Propagation</td>
</tr>
<tr>
<td>LQN</td>
<td>Layered Queuing Networks</td>
</tr>
<tr>
<td>MARTE</td>
<td>Modeling and Analysis of Real-Time and Embedded Systems</td>
</tr>
<tr>
<td>MDE</td>
<td>Model Driven Engineering</td>
</tr>
<tr>
<td>NFP</td>
<td>Non-Functional Properties</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>RAC</td>
<td>Resource Attachment Change</td>
</tr>
<tr>
<td>RC</td>
<td>Reference Change</td>
</tr>
<tr>
<td>SAC</td>
<td>StereotypeAttributeChange</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
</tbody>
</table>
1 Chapter: Introduction

1.1 Motivation

Model-Driven Engineering (MDE) shifts the focus of software development from code to models. The emergence of MDE has enabled the generation of formal analysis models (such as Markov chains, Queueing Networks, Layered Queueing Networks, Stochastic Petri Nets, Stochastic Process Algebra, fault trees, etc.) for the verification of Non-Functional Properties (NFP) (such as performance, reliability, availability, fault-tolerance, scalability, security, maintainability, etc.). The analysis models can be automatically derived by model transformations from the software models built for development.

The main motivation of this research is the need to seamlessly integrate formal analysis models for NFPs into the MDE process, in order to guide the design choices from an early stage and to ensure that the system under construction will meet its non-functional requirements. More specifically, this research focuses on the integration of performance analysis based on Layered Queueing Network [1][2][3] models in the model-driven engineering process using UML [4] [5]. The source software models are extended with MARTE [6] performance annotations, as described in more details in the thesis. The transformation from UML+MARTE to LQN is non-trivial because there is a large semantic difference between the source and the target models. Moreover, some aspects of the source model (such as nesting of software components or concrete data values) are not captured in the target model. This prevents us from using bi-directional transformations, which have the advantage of providing model synchronization for free [7].

An important concept in model transformations is the mapping between the source and target (meta)models, addressed by mapping patterns. In this research we propose a subclass
of such patterns called containment-based mapping patterns, which take into account the containment relationship when mapping the source and target (meta)models. Mapping is defined as the correspondence between elements of the source and target (meta)models. A design pattern in software design is defined as a reusable solution to a commonly occurring design problem within a given context [8]. Combining the definitions of mapping and design patterns, we define a mapping pattern as a reusable solution to a commonly occurring mapping problem between source and target metamodels in a specific context. A UML-based notation for specifying the definition and application of the mapping patterns, independent of the language used for implementing the transformation, is proposed in the thesis. Such a mapping specification is reusable and can be used to design different kinds of transformation (such as batch or incremental) implemented in different model transformation languages (such as ATL [9], QVT [10], Epsilon [11]).

The motivation for the mapping patterns is to answer a challenging question that is “what needs to be transformed into what”, thus shifting the focus from the implementation of the mapping by transformation rules to the mapping itself. We show how the containment-based mapping patterns are applied to the design of transformations from UML+MARTE to LQN, both for batch and incremental transformations. (Please note that the complete implementation of the batch transformation in Epsilon ETL was done in separate work). In this research, we extended the batch transformation to generate, beside the target model, a traceability model containing trace links between mapped source and target elements. After solving the generated performance model with an existing solver, the performance results are fed back to the software model by following the cross-model trace links. The software
developers can see the performance results as MARTE stereotype attributes, using a standard UML editor.

Another motivation of the research is to keep the software and analysis models consistent after the software model evolves during the software development process to meet functional or non-functional requirements. The proposed incremental change propagation aims to keep UML software model and performance model consistent with each other. More specifically, the thesis presents the design (based on the mapping patterns), implementation (in Epsilon EOL language) and evaluation of an incremental change propagation (ICP) approach to re-synchronize the software and performance models after any kind of change. During the development process, when the software model evolves, we detect the changes with the Eclipse EMF Compare tool, then incrementally propagate them to the LQN model. The proposed ICP is implemented with the Epsilon Object Language (EOL) and it is evaluated by applying it to a set of case studies.

Other aspects that motivate the work in this thesis are as follows:

- Change propagation is still an open problem [12] in the literature. The proposed incremental change propagation approach fulfils the need addressed by the authors in [13] [14] for supporting evolution and co-evolution between different artifacts such as analysis and design models.

- Improving the quality of nonfunctional requirements with the help of analysis models. According to [15] solutions to model evolution should be integrated with model quality solutions and model consistency challenges, since the goal of model evolution is to improve the quality of the system.
• Providing automated support for performance evaluation as early as possible in the life cycle, in order to help the developers to understand and interpret the performance results from the point of view of the software rather than the performance model.

• Gaining experience with a new family of languages Epsilon (standing for Extensible Platform of Integrated Languages for model management) intended for model transformation and model management [11]. Epsilon is interesting because, according to [103] it has been adopted in the industry as well as in the academia.

1.2 Research Objectives

The proposed research is in alignment with the objectives of an industrial initiative called PolarSys [16], created by major industry players and tool providers to develop and support open source MDE tools over Eclipse. In this research, we use the open-source tool Papyrus [17] developed under the PolarSys umbrella. There are many benefits to the software industry from the integration of the NFP analysis in the MDE process, for example improving the quality of software products by early detection and solution of NFP problems, in addition to avoiding project cancellation due to NFP failure.

The objectives of the proposed research are summarized in this section.

a. The objectives of the containment-based mapping patterns are as follows:
   • Provide high level specification of mapping between source and target (meta)models.
   • Answer a challenging question that is “what needs to be transformed into what”, thus shifting the focus from the implementation of the mapping by transformation rules to the mapping itself.

b. The performance analysis roundtrip has three objectives:
• Automating the generation of cross-model traces.

• Developing the performance analysis roundtrip for performance results feedback to the UML+MARTE model based on cross-model traces.

• The ability to store the software model and its performance results in the same file offers opportunities for further automating the analysis. For example, it could be used to complete the automation process in the performance improvement approach based on software antipatterns proposed in [18]. So far, the method for detecting antipatterns takes as input an XML file built by hand, which combines information about the software model and the performance results. This step could be automated now by applying our approach. This would relieve the analysts of repetitive error-prone manual actions.

c. The incremental change propagation approach has its own objectives:

• Support (semi-)automatic co-evolution of software model and analysis model.

• Support incremental propagation of (small) changes between the software model and analysis model.

1.3 Thesis Contributions

The contributions of this research are summarized as follows:

1. Propose a set of containment-based mapping patterns which are used to map a group of containment-related source model elements into a group of target model elements. A UML-based notation for specifying the definition and application of the mapping patterns, independent of the language used for implementing the transformation, is used in the thesis. Such a mapping specification is reusable and can be applied to the design of different kinds of transformation (such as batch or incremental) implemented in
different model transformation languages. We apply the patterns in the context of a non-trivial transformation from UML software models annotated with MARTE performance information into Layered Queueing Network (LQN) performance models (see Chapter 4).

2. Help designers to improve their design by feeding back the performance results to the software model via performance analysis roundtrip. First we extend the batch transformation to generate, beside the target model, a traceability model containing trace links between the mapped source and target elements. After generating the performance model and solving it with an existing solver, the performance results are fed back to the software model by following the cross-model trace links (see Chapter 5).

3. Develop a fully automated incremental change propagation (ICP) approach from the UML+MARTE software model (the source) to the LQN performance model (the target). The ICP is designed by reusing the pattern-based mapping specification between the source and target models (introduced in Chapter 4) and is implemented in the EOL language (see Chapter 6). ICP applications are given in Chapter 7 and the ICP evaluation is presented in Chapter 8.

The results of this research are published in the following three papers and another two are in preparation:


1.4 **Thesis Contents**

This section presents the overall organization of the thesis proposal and the content of each chapter.

Chapter 2 describes the background and state of the art in model transformation, model driven analysis of nonfunctional properties (NFP), coevolution of software and analysis models, comparison and traceability of models.

Chapter 3 describes a high-level view of the proposed approach.

Chapter 4 presents the containment-based mapping patterns and applies them in the context of the transformation from UML+MARTE as source to LQN as target. It continues by describing the application of the mapping patterns to the design of the transformation rules for a batch transformation implemented in ETL.
Chapter 5 extends the batch transformation with a trace model and shows how to realize a performance analysis roundtrip by using the trace model for feeding back the performance results to the software model.

Chapter 6 presents the algorithm for the whole ICP from UML+MARTE to LQN, including the change detection phase and the incremental change propagation steps.

Chapter 7 presents the propagation of different kinds of changes with more details.

Chapter 8 describes the evaluation of the ICP by applying it to a few case studies.

Chapter 9 concludes the thesis and discusses future research directions.
2 Chapter: Background and State of the Art

2.1 Model Transformations

Model transformations are considered crucial to Model Driven Engineering (MDE). In [19] a model transformation is defined as an automatic generation of target models from source models according to a given set of transformation rules. The source and target models are specified with modeling languages, which in turn are defined by metamodels describing the abstract syntax and static semantics of the language. According to OMG’s guide [20], a metamodel is a model that not only defines a modeling language, but is also expressed using a modeling language. The authors in [21] classify the transformations according to the source and target metamodels to two types: exogenous and endogenous. In exogenous transformations, the source and target metamodels are different. For instance, the transformation from UML software model to Layered Queueing Network (LQN) performance model proposed in [22] and presented in chapter 5 is exogenous. Exogenous transformations are also called in-out place transformation. Second, the endogenous transformations (known also as in-place transformation) are those in which the source and target model has the same metamodel. For example, refactoring UML class diagrams is an endogenous transformation.

2.1.1 Transformation Languages

Model transformation languages can be classified in three styles: declarative, imperative and hybrid. Each style has benefits and drawbacks. Declarative languages are more concerned with “what” rather than “how”, focusing more on identifying what are the mapping between the elements in the source model and the elements in the target model. Declarative language provide short, compact and brief transformations [23]. They can be
very useful in the cases where the mapping is simple because the source and target metamodels are very similar. However, they cannot handle the more complicated cases when the source and target metamodels are different in terms of their structures [11].

On the other hand, imperative languages focus on ‘how’ rather than ‘what’. They specify how the transformation is supposed to be executed and present it as a sequence of actions. Imperative transformation languages are similar to general purpose languages such as C++ or Java [23]. They can handle more complex transformations and give a higher level of control to the user. However, imperative languages have some drawbacks, for instance the user needs to tackle some issues like tracing, resolving target elements, and orchestrating the transformation execution. This means that the language operates at a low level of abstraction. The last style, hybrid languages, which is a combination of the previous two, helps in overcoming the shortcoming of the imperative and declarative languages.

QVT, ATL and ETL are examples of hybrid languages as they are capable to handle complex transformation as they have the imperative features and providing a declarative rule-based execution scheme [11].

**QVT (Query/View/Transformation)** defined by the Object Management Group (OMG), consists of set of model transformation languages. QVT has two levels of declarative architecture: QVTr-Relational language that support generating traces, creating object templates and matching object patterns and QVTc-Core language that focuses on matching patterns where a set of variables are evaluated against a set of models. QVT relational and core levels are the execution semantics framework for the imperative part that utilizes them to invoke the imperative representations of transformations [10].
**ATL (Atlas Transformation Language)** was developed by the ATLAS group and built on top of the Eclipse platform. ATL expresses the transformation as a set of rules that represent the mapping between the target elements and the source elements [9].

**ETL (Epsilon transformation language)** is a declarative/imperative language from the Epsilon family, specialized for model transformations and offers powerful and concise language constructs. Also, the Epsilon engine takes over a number of tasks (such as what rule to apply next) that must be handled explicitly by a Java transformation. Epsilon is a family that provides different languages for model management tasks, such as Epsilon Validation Language (EVL), Epsilon Transformation Language (ETL), Epsilon Comparison Language (ECL), Epsilon Merging Language (EML), Epsilon Wizard Language (EWL), Epsilon Generation Language (EGL), Epsilon Object Language (EOL) and Epsilon Transformation Language (ETL) [11].

The batch transformation from UML+MARTE software model to LQN performance model is implemented in Epsilon in [22]. We built our incremental change propagation technique using the EOL language, a stand-alone general language that can be used to automate some general tasks. In this thesis, EOL facilitates reading from different files with different formats, like UML models and XMI files, and provides a mechanism to write and change LQN files and Trace files. Epsilon also provides an orchestration workflow solution extended from ANT [24] to a number of model management tasks developed in task-specific languages. In chapter 5, we discuss how we took advantage of ANT to automate the multi-step transformation that was developed in [22]. The transformation model represents the mapping between source and target models is generated automatically by Epsilon Haetae tool that is an extensible static analysis framework for the Epsilon
platform [25]. It takes as input the textual ETL transformation definition (transformation rules) and translates them to an ETL transformation model.

The Transformation Model conforms to the ETL and EOL metamodels [25]. An EOL program can have more than one module, whose body is a block consisting of a number of statements and operations that perform extra functions on the object’s types. *EolElement* is a super type for *Expression* and *Statement*. The root metaclass in the ETL metamodel is *EtlModule* which consists of a number of transformation rules. *TransformationRule* has a name and refers to a source and many target parameters. A guard is optional in an ETL rule and can be an EOL expression or block of EOL statements. The body of the rule is an EOL statement [26][11]. For more clarification let us look at the example of *Device2Processor* (see Code Fragment 4-4) rule representation in the transformation model. The *Device2Processor* class is an instance of *TransformationRule* metaclass bearing the name “*Device2Processor*“. Each rule has source and targets that are instances of an *Expression* class that is an *EolElement* whose type is *ModelElementType*. The *elementName* is *Device* in case of source and *Processor* in case of target.

### 2.1.2 Batch versus Incremental Transformation

Incremental transformation is becoming the alternative solution to the traditional solution (i.e., batch transformation) when the source model evolves during the software development. According to [21] [27], incremental transformation is defined as the ability to transform only the elements in the source model that have been changed and ignore the others.

In the literature, there are two main approaches for incremental transformation as noted in [12] [28]. The first approach is the batch transformation that depends mainly on rerunning
the whole transformation from scratch, even though not all parts of the source model have been evolved, and merge the existing target model with the new one [29]. However, rerunning the whole transformation does not maintain the transformation context which specifies the model transformation system execution state such as variable values and partial matches [28] and merging depends heavily on the trace information generated by the transformation language [12][30].

The second more practical solution is Incremental transformation. It focuses only on examining the elements of the target model affected by changes in the source model and propagating those changes from the source to the target model, without consuming time to re-execute the whole transformation. Incremental approaches are more economical in terms of execution time compared to the batch transformation [30], more practical and efficient [28][31], especially in large-scale systems [32], by avoiding unnecessary overhead caused by rerunning the whole transformation [33].

A good example of incremental approach can be found in [32]. The authors proposed a framework for incremental transformation. They developed two algorithms ShouldExist and DoesExist to identify the action that needs to be taken to change the target model according to the change in the source model. Our approach is similar to their approach in terms of checking the existence of a changed element in the source model, and then checking the existence of its corresponding element in the target model. However, in [32] is not taken into consideration that the changed element in the source model may need to satisfy some conditions in order create an effect in the target model. Our approach satisfies such conditions (called guards) which are verified before propagating the effect to elements in the target model.
Another technique for incremental transformation called Logic-based SLD Resolution presented in [30] is built in the context of *Tefkat* transformation language and can only support declarative transformations. On the other hand, it supports only atomic changes (element insertion and deletion) [33]. Our approach is built in the context of the ETL hybrid transformation language and supports both atomic changes (element insertion, deletion) and composite changes (element updating and moving).

The approach proposed in [34] is similar to our approach, as it supports incremental model synchronization in a unidirectional transformation. It is applied to class diagrams and relational database diagrams as source and target models, respectively. It depends on the old traces to update or delete elements in the target model. For creating a new element, it uses a knowledge base that contains information about the pattern of the transformation. However, the new element is created only at the end of the synchronization, which means that no modification can be done on the new element until the end of the synchronization. Our approach is different, as it supports change propagation between source and target models with very different metamodels. It was applied to a source model conforming to the UML metamodel and produces a target model conforming to the LQN metamodel. Each metamodel has different references and different elements with different properties. As a result, that difference brings more challenge and complexity in the interpretation of the change in UML and propagating it to LQN. Another difference that distinguishes our approach is that it needs to check some guards in order to decide whether a source element change will lead or not to the creation of a target element. In addition, our approach checks if the propagated change affects internally other elements in the target and updates them even if their source elements did not necessarily change.
The impact on the system performance model when applying a design pattern to a Service Oriented Architecture (SOA) design model is investigated in [35]. This work has been done in our performance research group at Carleton. The Role Based Modeling Language (RBML) is used to define the SOA design patterns applied to the source model. The changes produced by the pattern application are propagated to the performance model. The similarity between [35] and our work is that both use the transformation between software models in UML+MARTE and LQN performance models. One difference is that in [35] only modifications due to SOA design pattern applications are propagated to the LQN model via refactoring rules specific to every pattern, while in our case any modification to the software model detected with EMF Compare can be propagated to the LQN model. Another difference is that in our work we make use of mapping patterns to specify the mapping between source and target (meta)models in a reusable and implementation language independent way.

Forward and backward change propagation between source model and target model generated by an ATL transformation is proposed in [36]. The forward propagation depends on re-executing the whole batch transformation. The backward method does not support insertion in the target model. In the same context of ATL the authors of [31] proposed an incremental approach that supports only the imperative parts of the language, but does not support composite changes (e.g., move).

In the context of graph transformations, approaches such as [37][38][28] support incrementality. They were developed in the context of the VIATRA framework where the execution of the transformation is driven by change and based on graph pattern matching. However, only [38] supports composite change (update and move). In fact, the execution
Semantics of graph transformations is different from hybrid model transformation languages, such as ETL. Hybrid languages provide the designer with more practical solutions based on the problem at hand [27]. In graph transformations, models are represented as graphs and the transformation is based on graph theory [39]. In the same graph context, in [40][41] are proposed bidirectional incremental approaches based on Triple Graph Grammars, which are classified as declarative languages.

Different examples of bidirectional transformation languages, such as [7][42] and others, can be found in the survey paper [43], which discusses how bidirectionality supports incrementality. However, bidirectional transformation is not suitable for the problem we consider, because in our transformation not all properties of the source model are captured in the target model. For instance, the structural nesting of components cannot be represented in LQN, where the tasks that are counterparts of components cannot be nested.

Another example why the transformation from UML to LQN is not bidirectional is that LQN does not retain which entry corresponds to which activity partition, so we miss this information when transforming back from LQN to UML.

Table 2-2 compares the previously discussed incremental approaches with our proposed approach based on different criteria discussed in Table 2-1.

<table>
<thead>
<tr>
<th>Comparison criteria</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technique Used</td>
<td>The name of the technique used to apply the incremental transformation</td>
</tr>
<tr>
<td>Language Style</td>
<td>The classification of the transformation language used in the incremental transformation</td>
</tr>
<tr>
<td>Language Framework</td>
<td>The specific name of the transformation language</td>
</tr>
<tr>
<td>Change Type</td>
<td>What is the type of change?</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td></td>
<td>1) Atomic (single change) such as delete or insert.</td>
</tr>
<tr>
<td></td>
<td>2) Composite (set of changes) such as update or move</td>
</tr>
<tr>
<td>Source</td>
<td>What is the source model that represents the input of the transformation</td>
</tr>
<tr>
<td>Target</td>
<td>What is the target model that represents the output of the transformation</td>
</tr>
<tr>
<td>Conditions</td>
<td>If the incremental approach checks guard conditions that the source model element should satisfy before creating a new element in the target model</td>
</tr>
<tr>
<td>Profile</td>
<td>If the incremental transformation supports propagating a change in the value of a stereotype attribute when the source model element is extended with that stereotype.</td>
</tr>
<tr>
<td>Mapping</td>
<td>Specification of mapping between source model elements and target model elements</td>
</tr>
<tr>
<td>Automated Change detection</td>
<td>If the changes are detected manually or by a tool</td>
</tr>
<tr>
<td>Arbitrary or restricted changes</td>
<td>If the designer can apply any change or only specific changes</td>
</tr>
<tr>
<td>Reference</td>
<td>Technique Used</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------</td>
</tr>
<tr>
<td>[40]</td>
<td>Triple Graph Grammar (TGG)</td>
</tr>
<tr>
<td>[41]</td>
<td>Triple Graph Grammar (TGG)</td>
</tr>
<tr>
<td>[37]</td>
<td>Graph pattern matching</td>
</tr>
<tr>
<td>[38]</td>
<td>Graph pattern matching</td>
</tr>
<tr>
<td>[35] Coupled transformation</td>
<td>Hybrid</td>
</tr>
<tr>
<td>----------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>[34] Incremental model Synchronization</td>
<td>Hybrid</td>
</tr>
<tr>
<td>[31] Live transformation support for ATL</td>
<td>Hybrid</td>
</tr>
<tr>
<td>Proposed approach</td>
<td>Incremental change propagation (ICP)</td>
</tr>
</tbody>
</table>

Table 2-2 Comparison of incremental transformation approaches
In summary, the type of software artifact impacts the change mechanism that is required [44][45]. Only one of the examined approaches [35] supports incrementality between analysis models and design models, similar to our approach. However, it only supports some specific kinds of changes due the application of SOA design patterns, while our approach supports arbitrary changes detected by the EMF Compare tool. Additionally, the mapping between UML+MARTE to LQN was hard coded in the transformation from [35], while in our work we make use of mapping patterns to specify the mapping between source and target (meta)models in a reusable and implementation language-independent way. Our approach is the only one that updates the old traces after propagating the changes. Few approaches in literature support composite changes [33]; ours is one of them. According to the last comparison criterion, few approaches support the propagation of changes in the attribute values of stereotypes applied to the source model elements. The interest in handling stereotypes comes from the fact that the transformations from software to performance models takes as input UML software model with MARTE profile annotations.

2.2 Model Driven Analysis of Non-Functional Properties (NFP)

2.2.1 Generate Analysis Models from Software Models

Model-Driven Engineering (MDE) shifts the focus of software development from code to models. The emergence of MDE has enabled the generation of formal analysis models for Non-Functional Properties (NFP) verification, such as performance, reliability, fault-tolerance, scalability, security, availability, maintainability, cost, etc. The analysis models can be automatically derived by model transformations from the software models built for development. Many modeling formalisms and tools have been developed (such as queueing networks, stochastic Petri nets, stochastic process algebras, fault trees, formal
logic, probabilistic time automata, etc.) for the analysis of various non-functional properties (NFP). In this thesis, the Unified Modeling Language is used to represent the software model. There are different UML profiles defined to bridge the gap between analysis and UML software models. For example, MARTE (The UML Performance Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE)) [6] extends UML with timing concepts, resources and workload required for quantitative performance and schedulability analysis. The UML Profile for Schedulability, Performance and Time (SPT) [46] is a precursor of MARTE that was defined for UML 1.X. Both SPT and MARTE are used for annotating the performance properties of UML models. MARTE has been extended with a dependability profile called DAM that can be used for annotating dependability properties (such as availability, reliability, safety) of UML models [47].

**Generating Performance Models**

In the software performance engineering field, there have been significant efforts to integrate performance analysis into the software development process by using different performance modeling formalisms: queueing networks, Petri nets, stochastic process algebras, Layered Queueing Networks (LQN), and simulation. Performance model represents the quantifiable attributes of the system such as response time (how quickly the system responds to an event) or throughput (how much work the system can achieve in a specific time). A good survey of the techniques for deriving performance models from UML models is given in [4], and later in the book [5]. A few early examples of derivation of different kinds of performance models from UML are as follows. The technique presented in [48] follows the Software Performance Engineering (SPE) [49] methodology very closely, generating the same kind of models as in [49], but it cannot take as input
UML files produced with standard editors. In [50] UML models are transformed into Petri Nets, but the contention for hardware resources is not considered. In [51] it is presented a transformation from UML to Stochastic Process Algebra.

The performance research group from Carleton University has implemented UML-to-LQN transformations in different languages (such as graph-rewriting language Progres, text transformation language XSLT and general-purpose language Java) and was the first to use the standard UML metamodel libraries that were current at the time and the standard performance profiles SPT [46] and MARTE [6].

The most comprehensive model transformation of the Carleton group, which takes as input a number of different software models (including UML+SPT and UML+MARTE) and generates a number of target performance models (such as LQN, QN and Petri nets) is the PUMA transformation [52][53] and its extensions for Service-oriented Architecture, PUMA4SOA [54]. PUMA uses an intermediate model called Core Scenario Model (CSM) [53]. This way, PUMA succeeds in minimizing the large semantic gap between UML models and performance models and reduces the complexity of the transformation at the cost of having two separate transformations: one from UML+MARTE to CSM [55] and another from CSM to LQN [56].

Comparing the light-weight Epsilon ETL transformation developed in [22] with PUMA, the ETL transformation goes directly from UML+MARTE to LQN, eliminating the intermediate model used in PUMA. Thus, the transformation is faster (as there is no need to generate and store an intermediate model) and supports easily inter-model traceability between the source and target models. Other differences stem from the languages used to implement the transformations. Epsilon ETL is a declarative/imperative language
specialized for model transformations, which offers more powerful and concise language constructs for transformation than general-purpose languages. Also, the Epsilon engine takes over a number of tasks (such as what rule to apply next) that must be handled explicitly by a Java transformation. On the other hand, PUMA was developed in Java, a general-purpose language that does not provide built in operations to help navigating the source model, which makes the transformation code more detailed and complicated. In this thesis we built our incremental change propagation (ICP) approach based on the mapping patterns and the ETL transformation [22], which takes as input a UML software model with MARTE performance annotations, and generates a corresponding Layered Queueing Network (LQN) performance model in an XML format understood by the existing LQN solvers.

**Generating Dependability Models**

Dependability represents the ability of the system to perform its tasks in a specific time (reliability, availability) and to function correctly under stressful conditions (robustness). There are several approaches in the literatures proposing to generate dependability analysis models from software models. For instance, in [57] it is proposed an algorithm to convert fault tree models (FT) to equivalent Generalized Stochastic Petri Nets (GSPN) and Stochastic Reward Nets (SRN). The dependability of the system can be improved by using fault tolerance, which is defined as avoiding failure in the presence of fault mechanisms. FT analysis is an example of dependability analysis approach, where the system is exposed to predetermined failures and the FT analysis identifies which part of the system was the reason of that failure.
Another example of dependability analysis can be found in [58], where the authors propose an algorithm to automatically synthesize dynamic fault trees (DFT) - an extension of FT - from UML software models. Several approaches, such as [59][60][61][62][63], propose the derivation of state based analysis models SPN (Stochastic Petri Net), GSPN, and SRN from UML models.

2.3 Co-evolution of Software and Analysis Models

In the context of Model-Driven Engineering (MDE), where the models are at the heart of the life cycle of system development, model evolution cannot be avoided. The different types of models used are very exposed to change during the software development cycle due to different reasons, such as responding to new functional requirements, improving their quality, or reducing the complexity. In the literature, there are different classifications of model evolution. For instance, in [64] model evolution is classified in four types: 1) regular evolution, where the model changes 2) metamodel evolution, where the metamodel changes and then accordingly the models need to be updated to be conformant to their new metamodels; 3) platform evolution that leads to change in the application framework and code generators; 4) abstraction evolution due to the use of a new modeling language. In [65] another classification is suggested: 1) local/syntactic model evolution refers to changes in models due to changes in metamodels; 2) local/content-related model evolution refers to changes to model elements due to addition, deletion or update; 3) systemic/syntactic model evolution, concerned with changes in the modeling language; 4) systemic/content-related, concerned with merging different parts of the system. According to [66] the main reasons for model evolution are requirement evolution, language evolution and change in style (refactoring).
In this research, we focus on local/content-related model or regular model evolution. We consider evolving the software model with the goal of improving the nonfunctional requirements represented in the analysis models without changing the functionality. Such evolution can be called refactoring. It minimizes the complexity and improves the quality of the model. This is considered in [67] and [68] a necessary tool for handling model evolution. We use refactoring to evolve a UML+MARTE software model that represents the source model, and then automatically propagate all changes incrementally to the performance model (LQN), whose results are used for performance analysis.

2.4 Model Comparison

There are two ways to detect model changes according to the literature [69] [70] [71]: a) an operator-based approach that detects the changes as a set of operations [72], or b) direct comparison (e.g., by using EMF Compare [73]) to detect the differences between two versions of the same model. Direct comparison has advantages, especially in the case when comparing models that contain elements with unique identifiers [74] (UML is such a model). Therefore, we use the open-source EMF Compare tool to determine the differences between two versions of the same UML source model. EMF Compare depends on IdentifierEObjectMatcher.DefaultIDFunction that matches elements with the same identifiers [73]. According to [75], using Static Identity-Based Matching approach (which states that every model element has a unique identifier ) can be faster and more user-independent, in the sense that no configuration is needed from the user 's side.

2.5 Traceability

Traceability plays an important role in MDE for building relationships between source model elements and target model elements, by establishing trace links between those
elements to track, analyze and propagate the impact of change which results from evolving software models. There are different classifications of traceability approaches in the literatures. For example, [76] categorize the traceability approaches in three types: requirements, modeling and transformation. The requirements approach focuses on tracing the requirements in different development phases in the development life cycle to give the stakeholders a better understanding of the requirements semantics. Modeling approaches focus on tracing the relationship between different elements in different models. Transformation approaches focus on generating automatically trace links during the transformation. Trace links represent the relations between source and target elements. Different classifications based on storing and managing the trace links were proposed in [77]. *Embedded traceability* links are those where the trace links are stored internally with the models. Using this approach has the disadvantage that it pollutes the model by including all traces inside it. Another approach is *external traceability* links, which solves the pollution problem by storing the trace links as a separate model. In this thesis, we use external traceability links that fall in the transformation category, as they are generated automatically by running the transformation. The traceability links are used in this research for two purposes: to feed back the performance results to the UML software model and during the change propagation process. The traces themselves evolve when the UML model evolves. Therefore, they need to be updated during the change propagation. Further details about using traceability links are provided in chapter 5 and 6.

### 2.6 Higher-order transformation

A higher-order transformation (HOT) is defined in [78] [79] as a transformation that takes a transformation model as input and/or produces such a model as output. The textual ETL
transformation definition (transformation rules) is translated to an ETL transformation model. The ETL transformation model conforms to the ETL metamodel and represents the mapping between source and target models. Two advantages of the transformation model are highlighted in [80], which compares model transformation with transformation model. First, using a transformation model allows for focusing more on the properties of the transformation than on its execution. Secondly, transformation models enable the construction of higher-order transformation (HOT).

According to [78] [81], change propagation can be realized as a higher-order transformation which takes a transformation model as one of its inputs. In [78], HOT is classified into four types: a) transformation synthesis creates a new transformation from data modeled in other forms; b) transformation analysis processes other transformations to extract meaningful data; c) transformation (de)composition merges or splits other transformations, according to a (de)composition criterion; and d) transformation modification manipulates the logic of an input transformation.

In literature, a typical example of HOT is presented in [78] and developed in the AmmA framework. An ATL textual transformation is translated to an ATL transformation model by TCS injector [82]. An ATL transformation model is the input to HOT that produces another ATL transformation model, which is translated to a textual transformation program by TCS extraction. The input and output models conform to the ATL metamodel.

The proposed higher order transformation (HOT) for realizing Incremental Change propagation (ICP) takes ETL transformation model conforming to the ETL metamodel as an input for ICP, while the output is an updated performance model. Another example of HOT can be found in [83], which verifies the correctness of the ETL language. The first
step transforms an ETL transformation to DSLTrans, which is a graph transformation language; the second step verifies the transformation with the SyVOLT tool, which provides a symbolic execution of DSLTrans transformations.

2.7 Patterns for Model Transformation

Design patterns for model transformations provide solutions for a number of model transformation specification and design problems, improving the quality of model transformation. An important concept in model transformations is the mapping between the source and target (meta)models, which is addressed by mapping patterns. Success of MDE approach depends on mapping quality [84]. In this thesis we discuss a subclass of such patterns, called containment-based mapping patterns, which are used to map a group of containment-related source model elements into a group of target model elements (see Chapter 4). In this section we focus on two main categories of related works: mapping patterns and model transformation patterns.

Model Transformation Patterns

Patterns are a popular technique in the field of software engineering to solve recurring design problems. Model transformation patterns have been proposed and adopted in [85] [86] [87] [88] [89] [90] to solve model transformation problems. In [91] model transformation patterns are defined as “descriptions of transformation rules and transformations that are customized to solve a general model transformation design problem in a particular context”. Model transformation patterns are identified and classified in [92] [91]. Both provide a catalogue of design patterns in the following categories:

- **Rule modularization patterns** are concerned with improving the structure of the transformation rules by organizing them and their relationships to enhance the
transformation flexibility, maintainability and quality. An example of such pattern is the Phased Construction pattern [93], used to decompose the transformation into phases or stages, where each phase is considered a sub-transformation with one rule for creating objects of one target type.

- **Optimization patterns** focus on improving the efficiency of transformation execution. As an example Remove Duplicated Expression Evaluations pattern to cache the expressions values in the rules in order to avoid the recurrence in their evaluations [91].

- **Model to text patterns** to generate code from models. An example is the Visitor pattern to traverse the model [27].

- **Expressiveness patterns** provide the transformation language with extra abilities when it has a limitation. For example, the Simulating Explicit Rule Scheduling pattern helps in putting a constraint in the order of rules application [91].

- **Architectural patterns** are concerned with organizing the transformation systems or the relationships between the transformations in order to enhance their modularity and efficiency. An example of such pattern is the Transformation Chain to split up the transformations into sequences of sub-transformations.

- **Bidirectional transformation (Bx) patterns** are concerned with maintaining consistency between the source and target model, by defining not only a forward transformation from source to target, but also a backward transformation from target to source. For instance, Auxiliary Correspondence Model pattern is used to map the source and target metamodels.
• Classical patterns are related to popular design patterns [8] such as the Template pattern.

The model transformation patterns listed above are targeting the transformation rules to achieve the goals for each category and are concerned with the level of detail provided by the model transformation languages. As opposed to this, we focus on mapping patterns at a higher-level of abstraction, independent of the transformation language, concentrating on the conceptual relations between the source and target metamodels.

**Mapping Patterns**

In [90] are introduced four mapping patterns in model transformations between concrete and abstract domain specific language (DSL). These mapping patterns are: a) *element mapping* to map elements from the source model to elements from the target model; b) *attribute mapping* to map a property of a source element to a property of a target element; c) *element mapping variability* to map one element from the source model to many elements from the target model and d) *link mapping* to map a relationship between source elements to a relationship between target elements. The first three patterns can be considered as special cases of the S-T mapping pattern presented in detail in chapter 4. The other mapping patterns discussed in this thesis introduce more complex mapping problems, such as mutually exclusive containers and virtual container (presented in chapter 4) that map a group of source elements (i.e., a special container and contained elements) to a corresponding group of target elements.

Another kind of mapping approach between models is a *weaving model* (AMW) proposed in [94] [95], which defines a set of links between the source and target metamodels. A weaving metamodel based on EMF is presented in [96] in the context of the ATL language.
AMW was extended in [97] where the MeTAGeM framework was proposed for generating model transformation code for hybrid transformation languages such ATL and ETL by High Order Transformation (HOT). Weaving models and matching transformations to semi-automate the development of transformations are proposed in [98]. A weaving model captures the relations between source and target models and the corresponding matching transformation generate the weaving models. The approach can be applied when the source and target metamodels have a similar structure. A set of pre-defined mapping operation (MOps) are proposed in [99] to generate executable ATL code out of HOT and MOps. MOps are used to specify the mapping by extending AMW.

In general, a weaving model does not support complex mapping such as the containment-based mapping patterns presented in this thesis, which are mapping a group of source elements involved in a containment relation to a group of related target elements. A disadvantage of AMW are its platform limitations (a compatible versions of Eclipse is needed [100]). As opposed to it, our proposed mapping patterns are independent of the transformation language/platform selected for implementation. In [101] is presented a generic mapping algorithm that focuses on simple mapping models by analyzing the target metamodel to solve the ambiguity caused by the semantic gap between the mapping models and model transformation. Since this algorithm is based on “as good as possible” principle, it might require user interaction to solve the ambiguity correctly.

In [102] it is stated that traceability is a core topic in model transformations and representing the transformation by mapping allows one to query the transformation by algebraic operations. The paper focuses on chain transformations and suggested mathematical operations for transformation chaining. However, the paper does not discuss
how to handle guard conditions. The paper does not discuss the inconsistency of traces when the source model evolves. In our case we are interested in keeping the trace model consistent after evolving the source model therefore the traces are updated after every propagation of changes.
3 Chapter: Overview of the Proposed Integration Approach

3.1 Integration of NFP analysis into the MDE process

The main objective of the thesis is to integrate the NFP analysis based on quantitative models into the MDE process. More specifically, we are interested in the integration of performance analysis within the software development process, by using quantitative performance models (e.g., LQN [3] [1]) that are automatically derived from the software models (e.g., UML) extended with performance annotations (e.g., MARTE [6]). Performance analysis conducted by solving the performance models helps detecting the performance trouble spots and suggests how to change the software model in order to meet the performance requirements [53]. The integration of performance analysis in the MDE process presented in the thesis is implemented in the context of Epsilon, a new family of language consisting of different languages such as Epsilon Object Language (EOL) and Epsilon Transformation Language (ETL) [11]. According to [103], Epsilon has been adopted in industry (such as IBM Haifa, Telefonica, WesternGeco, Siemens, and NASA’s Jet Propulsion Laboratory) as well as in academia (such as the Universities of Texas, Oslo, Kassel and Ottawa). Also, Epsilon has a very active forum that provides an immediate technical support.

The proposed integration approach consists of the following:

a) We developed two integration use cases: performance analysis roundtrip (Section 3.2.1) and incremental change propagation (Section 3.2.2). In both cases, we make use of the containment-based mapping patterns presented in Chapter 4 to specify the mapping between UML source models annotated with MARTE profile (presented in section 3.3) to LQN target performance models (presented in section 3.4).
b) The batch transformation developed separately in [22] corresponds to the implementation in Epsilon ETL of the mapping patterns application. It automatically generates LQN performance models $P$ from UML software models $S$ annotated with the standard MARTE profile. In this thesis, we extended the transformation from [22] to generate a trace model along with the performance model, which contains cross-model trace links between elements from $S$ and $P$ mapped to each other by the transformation.

c) The goal of the performance analysis roundtrip is feeding back to $S$ the performance results. After solving the performance model $P$ with an existing solver, the performance results are fed back to the software model $S$ by following the cross-model trace links in reverse. The proposed roundtrip gives the software developers the ability to see the performance results using a standard UML editor and the ability to assess as early as possible the performance effects of different architecture, design, implementation and deployment alternatives, in order to satisfy the performance requirements.

d) Analysis and change of UML model: the UML software model is modified by the designer/analyst, who adds, deletes, updates and/or moves some model elements. The reasons for change are diverse: in the case of the performance roundtrip, the changes are intended to fix performance problems highlighted by the performance results, while in the case of ICP, the changes can be due to adding new functional features to the system or improving functional and non-functional requirements. This step produces a changed version $S'$ of the original UML model, $S$. Please note that the designer/analyst is responsible for making changes to the UML software model and to verify whether the model is correct after the changes. The performance analyst can analyze the
performance results and evaluate if the performance results meet the performance requirements after propagating the changes to the performance model by ICP.

e) Automatic change detection: All changes between the $S$ and $S'$ need to be detected. For this purpose, in this research we use an existing tool, *EMF Compare* which automatically detects all the differences between $S$ and $S'$.

f) Automatic incremental propagation of changes (ICP): The differences found in the previous step, the mapping patterns applied to the source and target metamodels, the trace model, the ETL transformation model and the old LQN model $P$ become inputs to the ICP module used to propagate the changes to the target model $P'$, which is denoted as $P'$ after being changed.

Section 3.2.1 describes briefly the performance analysis roundtrip process, and Section 3.2.2 the incremental change propagation (ICP) process. The last two sections of the chapter describe a running example of source and target model.

### 3.2 Examples of integration

#### 3.2.1 Performance analysis roundtrip

The performance analysis roundtrip aims to help developers understand and interpret the performance results in the context of the software model rather than the performance model. As shown in Figure 3-1, a UML software model with MARTE performance annotations (as described in Section 3.3) is taken as input in Step1 by a ETL batch transformation, which was developed in previous work [22] and extended in this thesis with the facility to generate a trace model along with the LQN target model. Cross model traceability means having direct trace links between $S$ and $P$, which helps in different ways:

a) propagate small changes from $S$ to $P$, b) support the co-evolution of the software and
performance model, and c) import the performance results obtained by solving $P$ to the software domain. Point (c) is discussed in more detail in chapter 5, while (a) and (b) are discussed in chapter 6. $P'$ represents the LQN model with performance results obtained after solving $P$ with an existing solver in Step 2, and $S'$ is the software model with performance results stored as values of MARTE stereotype attributes. In order to feed back the performance results from $P'$ to $S'$ in Step 3, the trace links are traversed in reverse, starting from elements in $P'$ to the corresponding elements in $S'$. Steps 4 and 5 represent the analysis and new changes made until the performance requirements are satisfied.

![Figure 3-1 Performance Analysis Roundtrip](image)

### 3.2.2 Incremental change propagation (ICP)

Another objective of the thesis is to design, implement and evaluate an Incremental Change Propagation (ICP) technique from a UML+MARTE software model to the corresponding LQN model. During the development process, the UML model evolves in order to meet the functional and non-functional requirements. This continuous evolution creates inconsistencies between the software and analysis models. There is
an urgent need to support the evolution of the performance model (or other analysis models) in step with the software model. Therefore, we develop an Incremental Change propagation technique that detects the changes with the EMF Compare tool, then incrementally propagates them to the LQN model to keep it synchronized.

As shown in Figure 3-2, the entire process is “primed” by transforming the original source UML+MARTE model for the first time with the batch transformation developed in previous work (Step1). The outcome is the performance model $P$ and the Trace model. Please note that the batch transformation is able to generate an entire target model at once from an entire source model, but it does not support incremental transformation. During the software development process, the following chain of actions is repeated many times:

- A decision is taken in Step2 to change the software model $S$ for a number of reasons (such as adding new functionality, improving the system architecture, improving non-functional properties, etc.)
- Different changes are applied to $S$ manually or automatically in Step3, producing a changed model $S'$, which is now out-of-synch with the performance model $P$.
- The differences between $S$ and $S'$ are detected with the help of the tool EMF Compare in Step 4.
- The ICP module (Step 5) takes as input the following: performance model $P$, trace model, differences between $S$ and $S'$ and mapping patterns between the source and target (meta)models, which provide a high-level specification focusing more on "what needs to be transformed into what" rather than "how". ICP takes into account
the set of changes between $S$ and $S'$ and propagates them to the target model $P'$ (Step6), synchronizing it with $S'$.

- After deriving $P'$, it is solved with an existing LQN solver in Step 7, obtaining performance results for a range of workload and configurations, which allows us to evaluate whether the performance requirements are satisfied after propagating the changes (step7).

The whole ICP approach is implemented in Epsilon Object Language (EOL) an imperative programming language for creating, querying and modifying EMF (Eclipse Modeling Framework) models [104].

Figure 3-2 Incremental Change propagation (ICP)

3.3 Source Model

The source model taken as input by the transformation to performance model is a UML 2.5 [105] software model annotated with MARTE [6] performance information. The source model contains two types of UML diagrams: a deployment diagram representing the
structure of the system and one or more activity diagrams representing the behaviour.

The deployment diagram contains a set of UML nodes stereotyped «device» that represent physical computational resources with processing capability, and a set of artifacts representing software components, each deployed on a device. Each activity diagram represents a scenario that is the realization of a use case and models the interaction between software components. The behavior of each participating component is modeled inside an *ActivityPartition* with one dimension (also known as a swimlane) which belongs to an *ActivityGroup*. A swimlane contains different types of action nodes and control nodes linked together by edges. Examples of types of action nodes are: a) *AcceptEventAction* - executed when an event has been triggered; b) *SendSignalAction* - responsible for creating and transmitting signal instances to the target object; c) *CallOperationAction* - transmits a message representing an operation call request to the target object and waits until a reply is received; and d) *Opaque Action* - a type of UML abstract class considered as an executable node included within the behaviour. The control nodes are responsible for the flow of tokens between other nodes. Examples of control nodes are the initial node, which indicates the starting point of the execution of the scenario and the final node, which indicates the termination point of the execution. *ForkNode, JoinNode, MergeNode,* and *DecisionNode* are other examples of control nodes. Other type of model element is *ControlFlow*, an activity edge responsible for passing tokens from its source node to its destination node. The activity edges interconnect activity nodes into a graph that represents the behaviour of an activity as a sequence of subordinate units.

In this chapter we use the example of e-commerce system model introduced by the authors of a book on software performance analysis [5] as the source model for our transformation.
The system contains three performance-critical use cases selected for performance analysis: *Browse Catalogue*, *Browse Cart*, and *Place Order*. Figure 3-3 represents the annotated deployment diagram of the system, showing the run-time architecture and the allocation of software components to hardware processing nodes. The system has three classes of customers with a population of $N1$, $N2$ and $N3$ users, respectively. (Note that $N1$, $N2$ and $N3$ are variables in the MARTE annotations). Each of the users is deployed on its own UserDevice host. In order to insure this, the multiplicity of *UserDevice1* is $N1$, and so on.

Each class of users is executing repeatedly the use case corresponding to its class. The scenarios that represent the realization of the three use cases are modeled by three activity diagrams: *PlaceOrder* scenario is given in Figure 3-4, *BrowseCatalogue* scenario is given in Figure 3-5 and *BrowseCart* scenario is given in Figure 3-6. For running the batch transformation successfully and get the expected results, the source model needs to satisfy the following assumptions: The namespace for each device element needs to be initialized to the UML element containing it; also, the namespace for each artifact needs to be initialized to the device containing it. Each artifact has at least one defined corresponding activity partition and both should have same name. *ControlFlow* has a property called *inPartition*, which must be set only if the control flow is defined inside an *ActivityPartition*; for those *ControlFlow* representing call requests that cross the border between *ActivityPartitions*, the *inPartition* property does not have to be set.
Figure 3-3 Deployment diagram of Ecommerce System

Figure 3-4 Activity diagram of Place Order Scenario
3.4 Target Model

The target model for this transformation is the Layered Queuing Network (LQN) model [1][2][3]. LQN is a performance model that is extended from queuing networks and can represent nested services (i.e., a server may also be also a client to other servers). An LQN model is a graph whose nodes are either software tasks (parallelograms) or hardware devices (circles) and the arcs denote service requests. Figure 3-8 shows the LQN model generated from the e-commerce example [22]. The tasks with outgoing but no incoming
arcs play the role of clients (also called reference tasks), the intermediate nodes with both incoming and outgoing arcs are usually software servers and the leaf nodes are hardware servers. A software or hardware server node can be either a single-server or a multi-server. Software tasks have entries corresponding to different services (represented as smaller parallelograms inside the tasks). The LQN metamodel is shown in Figure 3-7, and is based on the XML schema defined in the LQN user manual [3]. The Epsilon transformation engine, however, requires that the target metamodel be represented in EMF Ecore (the metamodeling language of the underlying platform Eclipse EMF [104]). The Eclipse framework offers a language called Emfatic, designed to represent EMF Ecore models in textual form. Therefore, we used the Emfatic language to express the metamodel from Figure 3-7 in a textual form, which in turn was converted into EMF Ecore.

Like the XML-based metamodel from [3], the root model element of the LQN metamodel is \textit{lqnmodel}, which is composed of one or more \textit{processor} model elements by using
composition associations. In other words, *lqnmodel* and *processor* have a whole-part relationship, following the hierarchy of the XML-based metamodel. *Processor* is composed of *tasks*, which in turn is composed of *entries* or *task-activities*. Entry is the parent of *entry-phase-activities* model element. *task-activities* and *entry-phase-activities* are mutually exclusive containers of *activity* model element. *Activity* is the parent of children of type *synch-call* and *asynch-call*. *Task-activities* element is composed of elements of three types: *Activity*, *Precedence* and *Reply-entry*. *Reply-entry* is the parent of *Reply-activity*. In addition, the elements named *Pre*, *Pre-or*, *Pre-and*, *Post*, *Post-or* and *Post-and* are all children of *Precedence* model elements.

![Figure 3-8 LQN Model generated from e-commerce software model](image-url)
4 Chapter: Containment-based Mapping Patterns

4.1 Definition of mapping patterns

In model transformations, the understanding of the source and target metamodels is an essential requirement for mapping the source model elements to target model elements. Once the mapping is identified at a high-level of abstraction, it can be implemented as a transformation with any available model transformation language. The main advantage of the separation of concerns between mapping and the implementation of the transformation is the independence of the mapping from the language used for implementing the transformation. Thus, the designer has the opportunity to focus more on the conceptual aspects of the relations between source and target (meta)models [106] and has flexibility in choosing the language for implementation. Our vision is compatible with other research projects who have studied the specification of mapping between metamodels as an independent concept [107] [108] [109] [96].

In this chapter we present a set of patterns called containment-based mapping patterns which take into account the containment relationship when mapping the source and target (meta)models. Mapping is defined as the description of the correspondence between the source and target (meta)models. A design pattern in software design is defined as a reusable solution to a commonly occurring problem within a given context [8]. Combining the definitions of mapping and design patterns, we define a mapping pattern as a reusable solution to a commonly occurring mapping problem between source and target metamodels in a specific context. The goal of the mapping patterns is to answer a challenging question that is “what needs to be transformed into what”, thus shifting the focus from the implementation of the mapping by transformation rules to the mapping itself.
4.2 Specifying Containment-based Mapping Patterns in UML

In UML patterns can be represented by *Collaborations*. According to UML2.5 [105] *Collaboration* is a way to represent how a set of instances interacting with each other to accomplish a specific goal. Every instance in the *Collaboration* plays a *collaborationRole* which defines the use of that instance. The relationships between the instances in the *Collaboration* are represented as *Connectors* between the *collaborationRoles* that are played by those instances. The *Connectors* identify the communications paths between the instances that participate in the *Collaboration*. The application of the pattern in a specific context is called a *CollaborationUse*. In a *CollaborationUse*, instances from that context are binding to the *collaborationRoles* of the *Collaboration* by *roleBindings*. Each *roleBinding* shows every instance with its *collaborationRole*.

The *Collaboration* is shown with its internal structure (*collaborationRoles, Connectors*) as an ellipse shape stereotyped «Collaboration» (see Figure 4-1). The *collaborationRoles* are considered the *Properties* of the *Collaboration*. The *CollaborationUse* is shown as ellipse within the internal structure compartment of the context *Classifier*. Each *CollaborationUse* has a name and a *Collaboration* type. The *roleBindings* in the *CollaborationUse* are implemented by *Dependencies* (see Figure 4-2).
4.3 Classifying and documenting of Containment-based mapping Patterns

4.3.1 Classification of Containment-based mapping

In this section we classify the mapping patterns in two categories: Figure 4-3 shows the relationships between these patterns.

1. Mapping patterns applied between source and target metamodels:
   i) S-T mapping pattern.
   ii) Containment mapping pattern.

2. Mapping patterns applied to one metamodel (source or target) to define the group of model elements involved in a containment relationship:
   i) Simple Container pattern
   ii) Intermediate Container pattern.
   iii) Virtual Container pattern.
   iv) Mutually Exclusive Containers pattern.

4.3.2 Documenting the mapping patterns

In this section we use a specific template for documenting the mapping patterns. The template, inspired by [8], includes pattern name (usually a few words describing the problem), problem description (describing the context of the problem and when to apply the pattern), solution (describing the elements that the solution consist of) and example
(showing the application of the pattern in a specific context). All applications in this chapter are given in the context of UML+MARTE model as a source model and LQN as a target model. There is a large semantic difference between the source and target models (see Sections 3.3 and 3.4). The mapping between source and target models are explained in the following subsections. As mentioned earlier, we specify the mapping patterns in UML, so the pattern notation is independent of the transformation language. Some examples of collaborationRoles used in the pattern definition are:

- sContainer: a container instance from the source model.
- sElement: contained instance from the source model.
- tContainer: container instance from the target model.
- tElement: contained instance from the target model.

![Containment-based mapping patterns relationships](image)

**Figure 4-3** Containment-based mapping patterns relationships

### 4.4 Mapping patterns applied between source and target metamodels.

#### i) S-T mapping pattern

*Problem description:* an element or property in the source model needs to be mapped to an element(s) or property(ies) in the target model. “S-T” in the pattern name stands for
Source-to-Target).

**Solution:** S-T mapping pattern has two collaborationRoles: source refers to an element, property or stereotype property from the source model and target refers to an element or property from the target model. An element or property from the source model could be mapped to one or more elements or properties in the target model (see Figure 4-4).

![Figure 4-4 S-T Mapping Pattern: definition](image)

**Example:** A device \( d \) from the UML source model plays the role of source and is mapped to Processor \( p \) from the LQN target model that plays the role of target (both shown in orange in Figure 4-5). Other examples of mapping properties and stereotype properties presented in section 4.4. Please note that we did not show the stereotype «GaExecHost» constraints for Device for keeping the figures more readable.

![Figure 4-5 S-T Mapping Pattern: application](image)

**ii) Containment mapping pattern.**

**Problem description:** A source model has a containment relationship in which an object contains other objects and their existence depends on the existence of the containing object. The containing object is called container and the contained object(s) is (are) content(s). The container-content relationship is also denoted as parent-child. The source containment submodel is mapped to a target containment submodel. When a new object needs to be
created in the target model, its container needs to be identified with respect to the containment relationship in the source model.

**Solution:** This pattern has four collaborationRoles: `sContainer`, `tContainer`, `sElement` and `tElement` (Figure 4-6). The container `sContainer` from the source model is mapped to the container `tContainer` in the target model and the content (called `sElement`) of `sContainer` is mapped to the content (called `tElement`) of `tContainer`. In other words, the containment relationship in the source model is mapped to the containment relationship in the target model. Thus, if `tElement` needs to be created in the target model its `tContainer` is identified by matching a group `{sContainer, sElement}` from the source model with a group `{tContainer, tElement}` from the target model. Please note that this pattern definition uses the application of the S-T pattern (via a CollaborationUse, as defined in the UML standard).

![Diagram of Containment Mapping Pattern](image)

**Figure 4-6 Containment Mapping Pattern: definition**

**Example:** in Figure 4-7, `m` is an instance of `Model` type and plays the role of `sContainer`. `Model` is a container of `Device` in the source model. An instance `d` of type `Device` from the source model plays the role of `sElement`. An instance `lqmmodel` of type `Lqmmodel` from the target model plays the role of `tContainer` and is mapped to an instance `m` (both shown in green). `Lqmmodel` is a container of `Processor`. The instance `p` of `Processor` type plays the role of `tElement` and is mapped to `sElement` `d` of `Device` type (both shown in orange).
4.5 Mapping Patterns applied to One Metamodel

i) Simple Container pattern

**Problem:** Two elements of a metamodel are involved in a simple containment relationship when the container and contained element are directly connected by a containment association.

**Solution:** the pattern is applied when the container and contained elements are connected directly. Figure 4-8 defines a Simple Container pattern from the target model. It has two collaborationRoles: `tContainer` played by the containing object and `containedElement` played the contained object(s).

**Example:** processor is an instance of type Processor and plays the role of `tContainer`, while `t` is an instance of type Task and plays the role of `containedElement` (see Figure 4-9). Other applications examples are shown in red color in Figure 4-16.
ii) **Intermediate Container pattern.**

**Problem:** There may be a need to generate a nested container for an element in between an existing container and that element.

**Solution:** This pattern has three collaborationRoles: `existingContainer`, `intermediateContainer` and `tElement`. The `intermediateContainer` represents the content of the `existingContainer` and plays the role of container for `tElement` (See Figure 4-10).
Example: Figure 4-11 shows that t: Task is an existing container in the target model, task-activities plays the role of intermediateContainer, and Precedence plays the role of tElement. Another example of applying the Intermediate Container pattern is shown in Figure 4-16 (blue color) where Reply-entry plays the role of tElement.

iii) Virtual Container pattern

Problem description: UML diagrams are classified into structural diagrams and behavioral diagrams. When a UML+MARTE software model is transformed into an LQN performance model, there is no separation of concerns between the structural and behavioral views in LQN, because a performance model captures the performance properties of the system in a single view.

Solution: This pattern helps to remove the separation between an element (sElement-Structural) from the structural view and its behavior represented by a collection of (sElement-Behavioural) from the behavior view. In the source model there may be no
direct association between (sElement-Structural) and (sElement-Behavioural), even if they are modeling different aspects of the same instance. In order to make clear which behavioural element corresponds to a given structural element, the pattern has a constraint that sElement-Structural and all its (sElement-Behavioural) should have the same name (see Figure 4-12). Using name matching, the pattern establishes a “virtual container” relationship between (sElement-Structural) and its corresponding (sElement-Behavioural).

**Example:** In the source models, an Artifact is a structural construct that models a deployed component. Artifact is mapped to an LQN Task (both shown in green in Figure 4-13). At the same time, we are interested in the runtime behavior of the component represented by the Artifact, which is described in the source model by different activity partitions included in the activity diagrams modeling scenarios. Although there is no direct association between an Artifact and its corresponding behaviours described by ActivityPartitions, we apply the Virtual Container pattern with a name-matching constraint to bring together the structural and behavioural views of each component. The CollaborationUse applying the Virtual Container pattern in Figure 4-13, shows that instance a of type Artifact, playing the role of sElement-Structural, is treated as a “virtual container” of the actual contents of actPart of type ActivityPartition that describe the component behavior. More details about the contents of ActivityPartition are given in in Figure 4-18.
i) Mutually exclusive Containers pattern

**Problem description:** Sometime content(s) in the model has(ve) two mutually exclusive containers. Only one of them should exist at a given time.

**Solution:** As shown in Figure 4-14 the Mutually Exclusive Containers pattern has three collaborationRoles: \( t\text{Container1} \) and \( t\text{Container2} \) are mutually exclusive containers, and \( t\text{Element} \) is the contained element. The content has a constraint helping to decide which container to use.

**Example:** According to the LQN metamodel, \( Activity \) has two mutually exclusive containers \( entry\text{-phase-activities} \) and \( task\text{-activities} \) (shown with green color in Figure 4-16). They play the role of \( t\text{Container1} \) and \( t\text{Container2} \), respectively. The LQN activity \( ac \) plays the role of \( t\text{Element} \) and has a constraint based on the entry type (see Figure 4-15).
If the entry is type “Graph” then use $t_{\text{Container1}}$, otherwise use $t_{\text{Container2}}$.

Figure 4-15 Mutually Exclusive Containers Pattern: application

Figure 4-16 LQN metamodel with some patterns applied

Please note that the applied mapping patterns in the target model are overlapped and the same model element can play different $\text{collaborationRoles}$ in different mapping patterns.
For example, in the Intermediate Container pattern that is overlapping with the Mutually Exclusive pattern, \textit{TaskActivities} (shown in Figure 4-16 half in blue and half in green) plays \textit{intermediateContainer} collaborationRole. At the same time, it plays \textit{tContainer2} collaborationRole in the Mutually Exclusive pattern. The \textit{Task} element is shown half in blue and half in red, as it plays \textit{containedElement} collaborationRole in the Simple Container pattern and \textit{existingContainer} collaborationRole in the Intermediate Container pattern. The last example is \textit{EntryPhaseActivities} that is shown in half in green and half in red, as it plays two collaborationRoles: \textit{tContainer1} in the Virtual Container pattern and \textit{containedElement} in the Simple Container pattern.

4.6 Application of mapping patterns in the context of UML+MARTE and LQN

As mentioned earlier the proposed containment-based mapping patterns are applied to map UML+MARTE source model elements to LQN target model elements and used in both the batch transformation presented in chapter 5, as well as in the incremental change propagation presented in chapter 6. In this section a combination of different patterns is applied together to show more complex mapping. As mentioned before, the mapping is not only between a source element and target element(s), but also between a group of related source elements to a group of related target elements. See for example, Figure 4-7 where (\textit{Model} with its content \textit{Device}) is mapped to (\textit{Lqnmodel} and its content \textit{Processor}) by applying the Containment pattern. There is a containment relationship in the UML model between \textit{Device} and \textit{Artifact}, and in LQN between \textit{Processor} and \textit{Task}. Since \textit{Processor} is mapped to \textit{Device} (shown in green in Figure 4-17) and \textit{Artifact} to \textit{Task} (shown in orange in Figure 4-17), the Containment mapping pattern can be applied to map a group of \{\textit{Device, Artifact}\} to a group of \{\textit{Processor, Task}\} (see Figure 4-17). In addition, a Simple
Container pattern is applied in the target models when there is a direct simple containment relationship.

In LQN each Task has one or more Entry constituted of an LQN activity graph generated from the partition of the UML activity diagram that models the entry behavior. (In general, a task may have multiple entries corresponding to different services offered by the task). In the activity diagram, an ActivityPartition starts either with an Initial Node or an AcceptEventAction, which are both are mapped to Entry type in LQN model (shown in orange in Figure 4-18). A combination of applying two patterns (Containment Mapping and Virtual Container) is shown in Figure 4-18. It maps the following UML group of elements:

\[ \{ s\text{Container} (Artifact), s\text{Element} (AcceptEventAction) \} \]

to an LQN group of elements:

\[ \{ t\text{Container} (Task), t\text{Element} (Entry) \} \].

Please note that Artifact is a virtual container that matches its name to ActivityPartition name. ActivityPartition contains an activity diagram composed of nodes and edges that represents the behaviour of an entry. There are two types of entries depending on the corresponding UML activity diagram structure: a) phase-based entry (called PH1PH2 type) composed of a sequence of one to three activities; and b) graph-based entry (GRAPH type) composed of an activity graph with branch/merge, split/join, etc.
In Figure 4-23, the top two tasks have a phase-based entry each, while the bottom task has a graph-type entry. If *Entry* is *PH1PH2* type, then *Entry* is the parent of *entry-phase-activities*. If *Entry* is *GRAPH* type, then *Task* is the parent of *task-activities*. Both containers *entry-phase-activities* and *task-activities* are mutually exclusive containers for a collection of activities that are generated from *OpaqueAction*, *CallOperationAction* and *SendSignalAction* which are mapped to LQN *activity* element. However, *SendSignalAction* should satisfy some conditions such it is located with its incoming control flow in activity graph type *GRAPH* which are mapped to LQN *activity* element and triggers creating Reply-entry and Reply-activity LQN elements. Figure 4-19 shows the application of three patterns: Containment Mapping, Virtual Container and Mutually Exclusive Containers to
map the source model graph composed of *CallOperationAction* (shown in pink in Figure 4-19) and its virtual container *Artifact* to a target model graph composed of *Activity* (shown in pink in Figure 4-19) and its mutually exclusive containers (*entry-phase-activities* and *task-activities*) (shown in orange in Figure 4-19).

![Diagram of mapping](image)

**Figure 4-19 Mapping {Artifact, CallOperationAction} to {(EntryPhaseActivities Xor TaskActivities), Activity}**

The rest of the elements in the activity diagram, such as *DecisionNode*, *MergeNode*, *JoinNode* and *ForkNode* are mapped to *Precedence* and its children *Pre*, *Pre-or*, *Pre-and*, *Post*, *Post-or* and *Post-and*. Such a case is considered one-to-many mapping. According to the LQN metamodel *task-activities* is an intermediate container to a *Precedence* element and its children. Figure 4-20 shows the application of intermediate container pattern where *Task* plays an *existingContainer* collaborationRole and a *task-activities* plays the role of *intermediateContainer* to *precedence* (shown in green) that plays *tElement* collaborationRole. Figure 4-20 shows only the mapping of *DecisionNode* (shown in green) as other control nodes have exactly the same mapping. In case of activity graph *ControlFlow* also is mapped to a *Precedence* only if it satisfies special conditions. For example, if the *ControlFlow* does not cross its partition to another one and it is not an edge for any control node such as *MergeNode* or *JoinNode*, then it can be mapped to an LQN.
Precedence element. Otherwise, if the ControlFlow is cross-border edge, outgoing from an activity node and targeting a node of type AcceptEventAction, then it has to be mapped to a Synchronous or an Asynchronous call in the LQN model. Such calls are sent from CallOperationAction node that is mapped to Activity the container of Synchcall element according to LQN metamodel (shown in green in Figure 4-21). Figure 4-21 shows the application of containment mapping pattern to map (CallOperationAction and its outgoing ControlFlow) to (Activity and its contained element Synchcall).

Figure 4-20 Mapping {Artifact, DecisionNode) to {TaskActivities, Precedence}

Figure 4-21 Mapping {CallOperationAction, ControlFlow} to {Activity, Synchcall}
The elements in the UML are annotated with stereotypes from the MARTE profile (especially the performance analysis PAM subprofile) to bridge the gap between the UML and LQN performance models. For more clarification, the deployment diagram contains two main types of elements, Device and Artifact. Stereotype «GaExecHost» annotates a Device to show execution resources and «Scheduler» stereotype annotated both Device and Artifact to represent a certain scheduling policy [6]. Figure 4-22 shows the mapping between the UML object d of type Device (shown in orange), its attribute (e.g., name shown in blue) and stereotype properties to its LQN counterpart p of type Processor (shown in orange) and its attributes.

![Figure 4-22 Mapping Device properties to Processor properties](image)

In the activity diagram AcceptActionElement, Initial Node, OpaqueAction, CallOperationAction and SendSignalAction are annotated with «PaStep» stereotype that is a type of «GaStep» and can inherits some of its properties; «PaStep» can be applied to UML action or message to indicate behavior steps and performance information, such as execution time demand (given by the PaStep attribute host-demand). It is important to emphasize that the UML extension mechanisms (stereotypes and their attributes) are an integral part of the mapping and have to be handled by the mapping patterns.
4.7 Using Containment-based mapping patterns for Transformation Rule Design

In model transformations, the understanding of the source and target metamodels is an essential requirement for mapping the source model elements to target model elements. Once the mapping is specified at a high-level of abstraction, it can be implemented with any available model transformation language. The main advantage of the separation of concerns between mapping and the implementation of the transformation is the independence of the mapping from the language used for implementation. Thus, the designer has the opportunity to focus more on the conceptual aspects of the relations between source and target (meta)models [106] and has flexibility in choosing the language for implementation.

In this section, we show some examples of the mapping patterns application in the context of the ETL batch transformation that was implemented in [12] without considering the containment-based mapping patterns. This batch transformation includes an ETL module that has 17 matched rules (non-abstract and non-lazy) and 26 EOL operations. The transformation rules represent the implementation of the mapping patterns which provide high-level specification independently of the transformation implementation. We provide some examples of the transformation rules used to transform each element from the UML+MARTE source model $S$ composed of a set of model elements $e \in S$, to its corresponding element(s) in the LQN target model $P$, composed of a set of model elements $t \in P$.

In ETL the matched rules are executed automatically by the transformation engine whenever the rule is applicable, but the lazy rules should be called explicitly by ETL built in equivalent() operation[11]. An operation can be called from any rule; it is a function
used to verify some conditions or return a value. The concrete syntax for the ETL rule has the \textit{rule} keyword followed by the rule name and by \textit{(transform, to)} keywords for defining the source and target parameter.

\subsection{Rule based on the S-T Mapping pattern}

Let us consider the implementation of the application of S-T mapping pattern. Figure 4-23 illustrates a high-level mapping between some of source and target model elements by the application of S-T mapping pattern to mirror the transformation rules showed in Table 4-1.

An ETL rule can have an optional \textit{guard} $g(e)$ which contains an EOL expression or block of EOL statements that must be satisfied in order to run the rule. If there is a guard $g(e)$ and it evaluates to \textit{true} for the respective $e$ instance taken as parameter, the rule will generate one or more target elements instances \{\textit{t}_1,\ldots,\textit{t}_n\} and initialize their properties:

\begin{align*}
\text{if } \left( ((g(e).\text{isDefined and } g(e)=\text{true}) \text{ or } g(e).\text{isUndefined} ) \right) \\
\mathcal{R}(e) & \rightarrow \{t_1,\ldots,t_n\}
\end{align*}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Fig4-23.png}
\caption{High-level view of the mapping between the source and target models}
\end{figure}
If a mapping pattern has constraints, they can be implemented as guard \( g(e) \). For example, Figure 4-24 shows the application of S-T mapping pattern with its implementation as a transformation rule to transform \( d < \text{type}> \) Device to \( p < \text{type}> \) Processor with a constraint (guard) stating that Device should have “GaExecHost” stereotype. The guard is implemented as EOL operation (see Code Fragment 4-3).

Table 4-1 High-level view of the transformation rules to transform UML+MARTE to LQN elements

<table>
<thead>
<tr>
<th>Transform UML Model Element ( E )</th>
<th>To LQN Model Element ( T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Model</td>
<td>lqnmodel</td>
</tr>
<tr>
<td>2. Device</td>
<td>Processor</td>
</tr>
<tr>
<td>3. Artifact</td>
<td>Task</td>
</tr>
<tr>
<td>4. Initial node</td>
<td>Entry</td>
</tr>
<tr>
<td>5. AcceptEventAction</td>
<td>Entry</td>
</tr>
<tr>
<td>6. OpaqueAction</td>
<td>Activity</td>
</tr>
<tr>
<td>7. CallOperationAction</td>
<td>Activity</td>
</tr>
<tr>
<td>8. ControlFlow</td>
<td>Precedence/pre/post</td>
</tr>
<tr>
<td>9. ControlFlow</td>
<td>synchcall</td>
</tr>
<tr>
<td>10. ControlFlow</td>
<td>asynchcall</td>
</tr>
<tr>
<td>11. DecisionNode</td>
<td>Precedence/Post/Or</td>
</tr>
<tr>
<td>12. MergeNode</td>
<td>Precedence/Post/Preor</td>
</tr>
<tr>
<td>13. JoinNode</td>
<td>Precedence/Post/Preand</td>
</tr>
<tr>
<td>14. ForkNode</td>
<td>Precedence/Pre/Postand</td>
</tr>
<tr>
<td>15. SendSignalAction</td>
<td>Activity/Replyentry/ReplyActivity</td>
</tr>
</tbody>
</table>

Figure 4-24 Example of Implementation of S-T mapping pattern
In order to create a new Device, the S-T mapping pattern and its implementation shown in Figure 4-24 is not complete because Processor’s container needs to be identified with respect to the containment relationship in the source model.

4.7.2 Rule based on the Containment Mapping pattern

As discussed in Chapter 4, containment-based mapping patterns intend to map a group of containment-related source model elements into a group of containment-related target model elements. Figure 4-7 shows the application of containment mapping pattern to map Model which is the root of the source model and has a simple containment relationship with Device element to Lqnmodel which is the root of the target model and has a simple containment relationship with LQN processor element. Please remember that the definition of containment mapping pattern (see Figure 4-6) uses the S-T mapping pattern to map Model to Lqnmodel and Device to Processor and Simple Pattern is applied to target model whenever there is a direct containment relationship (see Figure 4-16). It is important to note that containment mapping pattern has a horizontal view (where Model is mapped to Lqnmodel and Device is mapped to Processor according to containment mapping pattern definition) and a vertical view (where the submodel \( \{\text{Model, Lqnmodel}\} \) is mapped to \( \{\text{Device, Processor}\} \)). According to the syntax of ETL (which allows for a single source model element after the keyword transform) the implementation of this pattern needs two separate rules (see Figure 4-25) representing the horizontal view of the containment pattern. The first rule is Model2Lqnmodel (shown in Code Fragment 4-1), which generates the target element lqnmodel from the source element Model. Once the LQN Processor is generated it will be added as a child to the lqnmodel. The second rule is Device2Processor, which transforms a Device element from the source model to a Processor in the target
model and then adds the newly created Processor element to its container lqnmodel.

In order to implement the vertical view of the containment pattern that maps the source containment submodel \( \{ \text{Model, Device} \} \) to a target containment submodel \( \{ \text{Lqnmodel, Processor} \} \), ETL provides a built-in operation called “equivalent()” that resolves a source element to its target elements that has been transformed by other rules [11]. In other words, a processor element is generated from a Device source element and is added to the collection lqn.processor. As shown in Code Fragment 4-1 the transform part has two variables: “m” represents the source element Model of type UML and “lqn” the corresponding target element of type lqnmodel. Code Fragment 4-2 shows the operation getDevices() called from the rule Model2Lqnmodel in the context of the Model source element and returns a collection of all Device instances after verifying that each element in the collection (represented by \( d \) in the code) has the correct stereotype. This is done by passing it as a parameter to the operation called hasStereotype () (see Code Fragment 4-3) that returns “True” if the applied stereotype is equal to the passed parameter. Operation getAppliedStereotypes() is an ETL built-in method called to get all stereotypes that have been applied to a UML class since this operation was called within the context of the UML class.

![Figure 4-25 Implementation of mapping \{Model, Device\} to \{Lqnmodel, Processor\}](image-url)
The second rule, `Device2Processor` (Code Fragment 4-4) transforms each `Device` element of the UML source model to a `Processor` element in the LQN target model. This rule has a guard that has to be satisfied in order to be executed: check if the device instance has “GaExecHost” stereotype. The next part of the rule is to initialize the `Processor`’s attributes name and scheduling. This part maps the properties of `Device` to `Processor`’s properties.
The Processor name is initialized with the name of the corresponding device in the deployment diagram. The scheduling attribute is initialized by calling the operation GetStereotypeOfDevice() with the device as parameter to check if it has a Scheduler stereotype, then gets the value of the attribute otherSchedPolicy. The returned value is assigned to a variable schedpolicy which is then passed as a parameter to checkPSchedulerType() to set the value of scheduling attribute of the processor.

4.7.3 Rule based on Virtual Container and Mutually Exclusive Containers pattern

Figure 4-26 shows another example of implementing the containment mapping pattern where in the vertical view it maps the group \{Artifact, CallOperationAction\} to group \{(TaskActivities XOR EntryPhaseActivities), Activity\} and thus is implemented by “equivalent()”, as discussed before. As seen on the source model side, the Virtual Container pattern is applied, where Artifact is a virtual container to CallOperationAction element. In the target side mutually exclusive containers pattern is applied where only one of the containers TaskActivities, EntryPhaseActivities should be the container of Activity element based on the evaluation of the constraint. This pattern is implemented as two rules, Artifact2Task and CallOperationAction2Activity, that represent the horizontal view of the pattern. As discussed before, type of Entry decides the container of Activity. Therefore, during the execution of the rule Artifact2Task, the constraints is evaluated to decide the container. Activity element is generated from a CallOperationAction source element by calling CallOperationAction2Activity rule using equivalent(). If Entry type is Graph, then Activity is added to TaskActivities, and if it is PH1PH2, then Activity is added to EntryPhaseActivities.

It is important to emphasize the fact that containment-based mapping patterns mirror the
hierarchycal structure of the LQN metamodel that uses nested containment relationships to hold its elements together. The rules follow the mapping patterns to connect each generated target element with elements transformed by other rules, by using the \textit{equivalent ()} built in operation provided by ETL to resolve the source elements to their corresponding target elements \cite{11}.

For example, the \textit{Lqmmodel} root has a containment relationship with \textit{LQN Processors} elements, based on the fact that all generated \textit{Processor} elements have to be added as children to the \textit{lqn} root target element by applying the Simple Container pattern (shown in red color in Figure 4-16) and the Containment Mapping pattern that maps \{\textit{Model, Device}\} to \{\textit{Lqmmodel, Processor}\}.

Similarly, the Containment mapping pattern applied to map a group of \{\textit{Device, Artifact}\} to a group of \{\textit{Processor, Task}\} (see Figure 4-17) is implemented as two separate rules, \textit{Device2Processor} and \textit{Artifact2Task}, representing the horizontal view of the pattern. The vertical view is implemented by operation \textit{equivalent ()}, which invokes the \textit{Artifact2Task} rule to calculate the counterparts of each \textit{artifact} (i.e., the generated \textit{tasks}) and then adds them as children of the generated \textit{Processor} element in the target model (see Figure 4-17). \textit{Artifact2Task} invokes the rule to generate LQN entry type from a UML \textit{AcceptEventAction}. Both \textit{Artifact2Task} and \textit{AcceptEventAction2Entry} represent the implementation of the containment mapping pattern in Figure 4-18 that maps the virtual containment relationship between \{\textit{Artifact, AcceptEventAction}\} in the source model to the simple containment relationship \{\textit{Task, Entry}\} in the target model. Figure 4-19 shows the application of containment mapping pattern to map the group composed of \textit{CallOperationAction} and its virtual container \textit{Artifact} to a target model group composed
of Activity and its mutually exclusive containers (Entry-phase-activities and Task-activities). The corresponding rule for this pattern CallOperationAction2Activity (See Figure 4-26) is invoked by Artifact2Task rule to generate LQN Activity type. The rule which generates CallOperationAction invokes one of the two rules that transform a UML control flow to LQN Synchcall or LQN Asynchcall. Figure 4-21 shows the containment mapping pattern to map (CallOperationAction and its outgoing ControlFlow) to (Activity and its contained element Synchcall).

Figure 4-26 Implementation of Mapping {Artifact, CallOperationAction} to {(EntryPhaseActivities Xor TaskActivities), Activity}
5 Chapter: Performance analysis round trip

This chapter proposes an approach for performance analysis roundtrip in the context of model-driven engineering (MDE) of real-time distributed and embedded systems. More specifically, the proposed performance analysis roundtrip is between the UML source model composed of deployment and activity diagrams and the LQN target model. Please note that there are other forms of roundtrip that are not addressed in this thesis, such as between models and code, or between more 'front-end' models such as class and sequence diagrams and deployment models.

The starting point in the performance analysis roundtrip is a UML software model with MARTE performance annotations, such as performance requirements and resource demands. A batch transformation [22] generates the Layered Queueing Network (LQN) performance model from the source model. We extended the batch transformation to generate, beside the performance model, a traceability model containing trace links between mapped elements of the software and performance model. After solving the performance model with an existing solver, the performance results are fed back to the software model by following the cross-model trace links in reverse. The software developers can see the performance results as MARTE stereotype attributes, using a standard UML editor. The approach is illustrated by applying it to an e-commerce application inspired from [5].

5.1 Batch Transformation

5.1.1 Extending the batch transformation to generate the Traceability model

In MDE, traceability plays an important role for building relationships between source model elements and target model elements, by establishing trace links between those
elements to track, analyze and propagate the changes that results from evolving software models. In this section we present the steps for generating cross-model trace links. As every model used in MDE, the trace model has to conform to its metamodel. The authors of [110] argue that case-specific traceability metamodels can be more specialized according to each traceability scenario, therefore avoiding the establishing of illegitimate traceability links. This is unlike general-purpose traceability metamodels that provide to the tools the ability to create any number of trace links between elements from models regardless of their type, which give a good chance for creating illegitimate traceability links. Our Trace metamodel similarly to the Trace metamodel in [110], has a trace class that has a containment relationship with Tracelink class. However, our Trace metamodel is different from [110] in that the Tracelink class has two attributes sourceType and targetType representing the source element type and the target element type. Another thing that is different, instead of having sources and targets in Tracelink class as reference to source and target element instances, it has two attributes sourceName and targetName, representing the name of the source and target element instance, respectively (see Figure 5-1). The differences were made in order to be able to query the Trace metamodel to serve our purpose discussed in section 5.2.

Epsilon facilitates generating Trace model automatically when executing transformation module specifically the Post block. An ETL module can have Pre/Post blocks, where the post block is executed in the order that it has been specified after executing the transformation [11]. We create a new trace link for each rule executed by using transTrace that is a global variable set up automatically by ETL. Also getSource() and getTargets()
are methods of the public class *Transformation* in *org.eclipse.epsilon.etl.trace Package* and are used to retrieve the source and target elements instances.

![Figure 5-1 Trace Metamodel](image)

**5.1.2 Automate the Transformation Process**

As already mentioned, the transformation from UML+MARTE software model to LQN performance model is implemented in Epsilon, which is a family that provides different languages, such as Epsilon Object Language (EOL) and Epsilon Transformation Language (ETL) for building task-specific languages for model management tasks such as model transformation [11].

In order to generate LQN models from UML+MARTE software model, the multi-steps process presented below was followed in [22]:

a. *Building the source model.* The first step is building the UML software model with performance annotations as a source model for the transformation. An open source UML editor Papyrus [17] was used to build our source model. Papyrus development is supported by PolarSys, an Eclipse Industry Working Group created by large industry players and by
tools providers to collaborate on the creation and support of Open Source tools for the
development of embedded systems.

b. *Pre-transformation*: This is an optional step for checking and refining the source model
in order to discover and eliminate bugs or fix missing data before feeding it to the next
step, the main model transformation. An example is checking the *inPartition* attribute of
*ControlFlow* elements. Setting *inPartition* attribute was automated by using Epsilon
Object language (EOL) [11].

c. *Main Transformation*: developed in ETL language, it generates an initial LQN model in
XML format that needs some minor extra processing to be in a format acceptable for the
existing LQN solver tool.

```xml
<?xml version="1.0"?>
<project default="main">
  <target name="loadModels">
    <!--This part is to load the models(source,target) and their metamodels-->
  </target>
  <target name="Execute" depends="loadModels">
    <!--This part is to load ETL file and EOL file -->
  </target>
  <target name="main" depends="Execute">
    <java classname="tagPackage.TagModifier" classpath="bin">
      <!--This part is to load a Java class with a method that sends the initial LQN model file as input argument and
      returns LQN model file after changing the tag names as output argument-->
      <arg value="initial LQN model file"/>
      <arg value="modified LQN model file"/>
    </java>
  </target>
</project>
```

**Code Fragment 5-1 Ant build file**

d. *Post-Transformation: in this step* the XML file for the initial LQN model has to be
modified to be exactly conform to the XML schema [3]. The modification needs two steps.
First, changing tag names by inserting dashes ‘-’, which are not accepted by Emfatic, but
are used in the XML schema [3]. Second, adding a solver-param element that cannot be
derived from the source model. The modification for the tag names is done automatically by executing Java code but adding solver-param was done manually.

In this thesis, we automate the multi-steps transformation process using an orchestration workflow solution provided by Epsilon, which extended it from ANT,[24]. Code Fragment 5-1 represents the steps we followed to build the ANT file. Each workflow in the ANT file represents a project and each project has a number of targets, and each target has a number of tasks. The target can also depend on another target that has to be executed first. The default target is executed when the whole project is executed.

5.2 Feedback of Performance Results to the software Model

Our goal here is to feed back the performance results to UML software model during the round trip performance analysis. The developer can analyze the results, decide to make changes in the software model and repeat the process until satisfied with the performance results. In order to accomplish this goal, our approach in this round trip performance analysis depends basically on reading two files: first the Trace Model file generated automatically as a result of running the transformation; and secondly the XML file obtained from the LQN solver after reading, parsing and solving the LQN model file. Having the Trace model helps in matching each target element with its source element by following its trace link in a reverse way and then feedback its performance results to its source element. The source model is annotated with MARTE stereotypes that have performance attributes such as utilization, throughput and response time. We implemented the proposed approach in EOL language that allows us to read two files, Trace model and XML file with the performance results and write that results to the third file, the UML software model annotated with MARTE profile. EOL does not only provide a mechanism to read and write
from/to files, but also facilitate to create, query and modify XML documents. The procedure starts by querying the XML file with the performance results, and then querying the Trace model and getting the sourceName, sourceType, targetName and targetType attributes for each trace link. Next the targetName attributes are used to get the corresponding source attribute name. It is worth stressing that matching is not done only by name, but by type as well. Once the source element has been identified, then we can retrieve its applied stereotype in the UML software model and set all its performance attributes to values corresponding to the LQN results. The reason is to help the designers to better understand the performance results in terms of software model concepts rather than performance model concepts. The software developers may not be familiar with the performance model syntax but are certainly familiar with the software model. We applied our approach to the e-commerce application and fed back the performance results to the UML software model elements.

5.3 Performance Analysis

This section presents a brief performance analysis of the e-commerce system model introduced as an example in chapter 4, based on following the round-trip approach several times for different configurations corresponding to different resource multiplicities (both hardware and software). The software model is given in Figure 3-3, Figure 3-4, Figure 3-5 and Figure 3-6 while the automatically generated LQN model in Figure 3-7. As mentioned in chapter 4, there are three classes of users, each running a different scenario: BrowseCatalog, BrowseCart and PlaceOrder. The think time for each user is 1 s. The largest number of users is in class 1; in our experiments, we kept a constant ratio between the numbers of users: for 48 users in class 1, there is one in class 2 and one in class 3. For
each configuration, we solved the model under a variable load (from 100 to 800 users in total). Figure 5-2 shows the average response time for a class 1 user for four configurations, while Figure 5-3 shows the average throughput.

A. This is the base case, where the multiplicities of all tasks and processors are 1, with the exception of the User tasks (each with its own processor) and the CustomerInterface task, which has a thread for every User. The results show that the response time and throughput curves have each a knee starting at around 200 users; the performance deteriorates very quickly after the knee, so the operating point should be before the knee. The analysis of task and processor utilizations shows that this is a typical case of software bottleneck, with CustomerProcess the task that saturates first, limiting the concurrency level and the utilization of resources below the bottleneck. An appropriate solution is to raise the number of threads for CustomerProcess task – which is purely a software solution.

B. We raised the number of threads from 1 to 50. The results show that the response time and throughput improve considerably, and the knee of the curves moves to the right, being able to accommodate more than double the number of simultaneous users before the knee than in Case A. The next bottleneck is CatalogServer processor, followed immediately by the CatalogServer process. The solution is to raise the processor multiplicity to 5 and the software threads of Catalog Server (which is the next bottleneck) to 50.

C. The performance improvement from case B to C is less important than from A to B, but still, the knee moves further to the right, accommodating at least 100 more simultaneous users. The next bottleneck is the processor of CustomerProcess.
D. After increasing the multiplicity of the *CustomerProcess* processor to 5, the response time improves further and the knee moves beyond 800 users (where we stopped increasing the workload).

This simple example illustrates how the LQN performance model can be used to predict the performance effect of different configuration changes. Here the actual diagnosis of performance problems (i.e., detecting the system bottleneck) and finding a solution for alleviating the problem was done by a human analyst. In the future, we intend to integrate our performance round-trip approach with performance diagnosis algorithms that will enhance the level of support offered to software developers by MDE tools.

![Figure 5-2 Response time for class1 users](image-url)
Figure 5-3 Throughput for class 1 users
6 Chapter: Incremental Change Propagation (ICP)

There is an urgent need to support the evolution of higher-level artifacts such as analysis and design models [13][111]. Model evolution is important for improving the system quality [66] as it provides continuous feedback to the designers. When the software models evolve during the development process, the traditional solution for keeping the NFP analysis model (ex. performance model) synchronized is to rerun the entire transformation each time the software model changes. Such a solution is expensive, especially in large-scale models. According to [15] incremental approaches minimize the effort to change a small part of the model, as the effort is proportional to the size of the change.

In this chapter, we propose an Incremental Change Propagation technique (ICP) to propagate changes from the UML+MARTE software model to the corresponding LQN. The entire process starts by automatically generating an LQN model with the previously developed ETL batch transformation [22]. During the development process, when the UML model evolves, we detect the changes with the Eclipse EMF Compare tool, identify the elements that need to be updated in the target model leveraging the generated traces and the mapping patterns, and then incrementally propagate them to the LQN model to keep it synchronized. Note that Epsilon does not support incremental model transformation. The proposed ICP is evaluated by applying it to a set of case studies. Section 6.1 discusses change detection phase. Section 6.2 presents three forms of mapping and section 6.3 discusses change propagation in more details.

6.1 Change Detection Phase

The change detection compares directly two versions, $S$ and $S'$, of the same UML+MARTE model, which is the source model for the batch transformation presented in chapter 5. The
result of the comparison between $S$ and $S'$ is saved and queried as an XMI file called *Differences* (see Figure 6-1 Example of difference file). The root node of its schema is *Comparison*, a metaclass in the EMF Compare metamodel that has all comparison information such as matched resources, matched objects and detected differences [73]. The root can have many *match* children, and each *match* can have many submatches. A submatch has a *left* and *right* node representing the matched resources and zero to many *differences*. The differences are classified as *RC, AC, RAC or SAC*.

- **RC (ReferenceChange)** is detected when a reference value is changed (i.e., added, moved or deleted). In a *RC* case, a difference has two children: a *reference* that is the name of the reference changed and the *value* of that reference.

- **AC (AttributeChange)** is similar to *RC*, but it refers to an attribute rather than a reference. In an *AC* case, the difference has only one child that is an *attribute* instead of a reference.

- **RAC (ResourceAttachmentChange)** difference is detected when one of the roots of the matched resources changes.

- **SAC (StereotypeAttributeChange)** is detected when a stereotype attribute changed.

  In a *SAC* case, the difference has discriminant that has the stereotype attribute.

Each *difference* has a *kind* that can be *ADD, CHANGE, DELETE or MOVE*. *ADD* includes two cases: a) adding a new element within the values of a multi-valued feature; or b) any change in a containment reference, even if that reference is mono-valued, represents a "new" element in the model. In the *CHANGE* case, the engine considers any modification to a mono-valued feature as *CHANGE* and excludes the containment references from this rule. *DELETE* follows the same logic as *ADD*, considering that a change to containment
reference is deleting, even if that reference is a mono-value. A change is considered as a MOVE in two cases: a) moving an object from one container to another; and b) reordering the values of a multi-valued feature is considered as a series of MOVE [73].

Figure 6-1 represents a simple example of differences file. The root Comparison matches two resources identified by leftURI=\text{S}' and rightURI=\text{S}. Submatches is matching left that is an element \text{e}' from \text{S}' with the right that is an element \text{e} from \text{S}. (In the document we will refer to the left as \text{e}' and right as \text{e} for simplicity). From Figure 6-1 we know that reference \text{node} has been added and its value element represents the name of the new node.

We can notice that \text{value}, \text{left} and \text{right} have UML identifiers denoted by \text{uml#}. Later we will explain how we can match each UML identifier in Differences file with its corresponding identifier in the UML source file to get the name and the type of the UML element.

```xml
<compare:Comparison xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:compare="http://www.eclipse.org/emf/compare">
  <matchedResources leftURI="\text{S}'.uml" rightURI="\text{S}.uml"/>
  <matches>
    <submatches>
      <differences xsi:type="compare:ReferenceChange" kind="ADD">
        <reference href="http://www.eclipse.org/uml2/5.0.0/UML#/ActivityPartition/node "/>
        <value href="\text{S}'.uml# _A-TSI BSgEeesewaezJHPIkQ "/>
      </differences>
      <left href="\text{S}'.e'.uml#_jBw4oDrLEeWw1YjMw9yGdw "/>
      <right href="\text{S}.e.uml#_jBw4oDrLEeWw1YjMw9yGdw "/>
    </submatches>
  </matches>
</compare:Comparison>
```

Figure 6-1 Example of difference file

6.2 Forms of mapping

In the proposed ICP, the mapping is necessary in order to identify the target elements that need to be updated according to a change received from its corresponding source element. In general, the mapping facilitates generating information about:
1. From what source element type(s) is obtained every target element type.

2. From what source element type(s), property(ies) and/or stereotypes is obtained every target element property.

3. What target element types correspond to each source element type, considering that one source element may have multiple target elements.

4. What condition(s) exists for every source model element type, if any.

5. To which location (i.e. container) a new element should be added or moved with respect to its relationships with other elements.

In chapter 4 we proposed some mapping patterns that provide a high-level specification of the mapping between source and target models and applied them in the context of UML+MARTE model as a source model and LQN as a target model. In chapter 5 we proposed a trace metamodel and extended the batch transformation to generate a trace model which is an instance of the trace metamodel. The traces are another form of mapping between source and target models. A tracelink between source element and target element is created automatically after each successfully executed rules. In chapter 2 we presented the transformation model as a third form of mapping.

The trace model, transformation model and mapping patterns are forms of mapping and each one has its own characteristics that make it preferable in certain situations. For instance, in the context of ICP when an element is changed in the source model, all its differences must be detected in the change detection phase; then its target element can be identified by following its tracelink and the actual update can be executed. However, in some cases the traces are not sufficient to identify which target element needs to be changed or which property needs to be updated. For example, if a rule has a guard for its source
element type and that guard was not satisfied when running the batch transformation, then no trace will be generated for that source element. If the same element has been changed and the guard evaluation was affected so that the guard is now satisfied, but the source element does not have a trace, we need to find its corresponding target element from another form of mapping. Another example could be when the change affects the property of an old element from the source model. By following its trace, we can get its corresponding target element only. However, the trace only gives information about the corresponding target type and the source type and the mapping between their instances but does not say anything about the mapping between source and target properties. In order to find out information about the changed properties and their mapping, we need to have higher-level mapping. In general, traces are useful only in case of:

- deleting an old element - as we find the source element type from which is obtained every target element type (point 1 above)
- what target element type corresponds to each source element type (point 3 above).

To overcome the limitation of traces, the transformation model helps in obtaining the information mentioned above (points 1 to 4). Having the transformation model has two advantages: first, it uses a transformation model that can be treated as any model, so it can be queried, instantiated and so on. Secondly, the transformation model is a higher-level abstraction that allows us to focus more on the properties of the transformation rather than its execution. Therefore, it is possible to reuse the mapping contained in the batch transformation rules, without duplicating the effort of remapping the source and target models. On top of that, generating the transformation model helps in realizing the ICP as a higher-order transformation, which takes such model as one of its inputs. This form of
mapping has its limitations too, as it is language dependent. The transformation model should conform to the transformation language definition metamodelling. Also, in some cases such as creating new element or moving an element to a another container requires to identify the container of that element and to preserve its relations with other elements. Such information can be found in the imperative part of an ETL rule and can not be generated from the transformation model because traversing the imperative part is an error prone and time consuming specially for large transformation that has large number of rules. The proposed mapping patterns overcome the limitation of the traces and the transformation model, as they are language independent and provide high level specifications that facilitate generating the implementation of the the transformation rules and provide all the necessary information mentioned above (points 1 to 5) as input to ICP.

6.3 Incremental Change Propagation (ICP)

This subsection presents the assumptions, general guidelines and multi steps process for the proposed incremental change propagation technique (ICP).

6.3.1 Assumptions

- The technique follows the same batch transformation assumptions as discussed in chapter 3, section 3.3.

- The Epsilon Object Language (EOL) is used to implement the proposed ICP technique. The following files are used as ICP input: two versions of the source model (original $S$ and changed $S'$), $LQN$ target model $P$ corresponding to the original source $S$, the trace model corresponding to the original transformation, the Differences file and the mapping patterns. The result of running the ICP is the synchronized $LQN$ model, with all propagated changes.
• The identification names for the generated differences file, trace file, UML model and LQN model in the run configuration should not be changed in order to run the ICP technique successfully.

• The changes are classified into Update, Move, Delete and Add. Update change is a set of composite changes consisting of add and delete.

6.3.2 Notation

In this section, we introduce some notation used in the thesis:

• $M_S$ is the original source model, which consist of original elements $e$ denoted by $M_S!e$.

• $e$ is an instance of type $E$ from the source model denoted by $e:E$.

• $M'_S$ is the changed source model, after applying a set of changes to $M_S$. $M'_S$ consists of elements $e'$ denoted by $M'_S!e'$. The differences between $M'_S!e'$ and $M_S!e$ is denoted by $diff$ and can be more than one $diff$. Set of $diff$ represents the differences between $M'_S$ and $M_S$.

• Every type $E$ in $M_S$ has set of original properties $p$ denoted by $E.p$. If $p$ changes, then $p'$ represents the changed property set $E.p'$

• $M_T$ is the original target model, which consist of original elements $t$ denoted by $M_T!t$.

• $M'_T$ is the changed target model, after propagating a set of $diff$.

• $t$ is an instance of type $T$ from the target model denoted by $t:T$.

• $t(e)$: an instance $t:T$ that is mapped to $e:E$.

• Tracelink $\lambda = (e:E, t:T)$: links $e:E$ and $t:T$ where $e \in M_S$ and $t \in M_T$.

• Trace $\Lambda = \{ \lambda_1, \lambda_2, \ldots \lambda_n \} = \{( e_1:E, t_1:T), ( e_2:E, t_2:T) \ldots \}$ : a collection of tracelinks contains a tracelink for each target element $t$.

• Mapping $\mu = (E, T)$: maps one source element type $E$ to one target element type $T$. 

87
• Mapping $\mu = (E, T_1, T_2, \ldots)$: maps one source element type $E$ to many target element types $\{T_1, T_2, \ldots\}$.

• Mapping $\mu = (e, t)$: maps one source instance $e$ to one target instance $t$.

• Mapping $\mu = (E.p, T.p)$: maps source element property to target element property.

EMF Compare matches pairs of corresponding elements that are different in $M'$ and $M$. Possible matches $\{e, null\}$, $\{null, e'\}$, $\{e, e'\}$ have the following meaning:

• $\{e', e\}$: $e'$ exists in $M'$ and $e$ exists in $M$, which means that the old element $e \in M$ has been changed to $e' \in M'$.

• $\{e', null\}$: $e'$ exists in $M'$ which means that $e'$ is a new element added to $M'$ and $e$ does not exist in $M$,

• $\{null, e\}$: $e$ exists in $M$ but $e'$ does not exist in $M'$, which means that $e$ is an old element deleted from the model.

Other notations used in the thesis are:

• The result of comparing the two sides of a match results in a set of differences $diff$.

  Each difference has a type $\epsilon \{RC, AC, RAC, SAC\}$, where: $RC$ means $ReferenceChange$, $AC$ means $AttributeChange$, $RAC$ means $ResourceAttachmentChange$, and $SAC$ means $StereotypeAttributeChange$.

• The change actions applied to $M$ can be of different kinds, where $kind \epsilon \{ADD, DELETE, CHANGE, MOVE\}$.

• $DiffVal$ is the value of a changed model element or reference.

• $DiffRef$ is a reference name if $diff.type = RC$.

• $DiffAttr$ is an attribute name if $diff.type = AC$.

• $DiffDiscriminant$ is a Stereotype attribute name if $diff.type = SAC$. 

88
6.3.3 ICP main steps

The change propagation has six major steps applied repeatedly for every match:

**Step 1:** Checking the existence of \( e \) in \( M_S \) and \( e' \) in \( M'_S \). As mentioned above, there are three cases denoted as Case 1: \{ \( e', e \) \}, Case 2: \{ \( e' \), \text{null} \}, Case 3: \{ \text{null}, e \}. See Appendix A, A.1, A.2 respectively.

Examples of such cases are shown in detail in chapter 7. For instance the activity partition \( A2 \) is an old element in \( M_S \) (see Figure 7-2 a) that is changed in \( M'_S \) (see Figure 7-2 b). On the other hand, the activity partition \( F \) does not exist in \( M_S \) (Figure 7-2 a) but it was added to \( M'_S \) (Figure 7-2 b). The activity “\( \text{CallOperationAction ph1} \)” exists in \( M_S \), but it is deleted from \( M'_S \). Note that the changes in UML diagrams are shaded in darker grey.

**Step 2:** Matching the UML elements identifiers from the difference file with the same identifier in both UML model versions \( M_S, M'_S \) to get the name and type of \( e \). The matching is done by operation \( \text{getName}(id: \text{String}) \) that receives the identifier of \( e \) as a parameter and returns its name. Similarly, we get the type of \( e \) by executing operation \( \text{getType}(id: \text{String}) \).

**Step 3:** Getting all differences between \( e \) and \( e' \). Each match between \( e \) and \( e' \) can have one or more (diff) differences. Therefore, we need to iterate among the differences to get the type, kind and value if it exists. For example, after we identified that \( A2 \) is an original element in step1 and get its type in step2, we get all its differences in step 3. \( A2 \) has three differences and each of them needs a specific action, as described in the next chapter, Example 1, match A.

**Step 4:** Getting and checking the trace of \( e \). There are three cases related to the existence of \( e \) and \( e' \) in \( M_S \) and \( M'_S \), respectively:
Case 1: If \( e \) and \( e' \) both exist, we get their differences and check if target element(s) \( t \) mapped to \( e \) exists in \( LQN \) by checking if there is a trace or not. If \( e \) has a trace (i.e., its target element exists in LQN) we can update it according to the differences (see Appendix A.1, case1.A). If \( e \) does not have a trace, then we need to get its target element type \( T \) from the mapping patterns. The creation of a new target element \( t \) type \( T \) depends on whether there are constraints on the source type or not; if there are and they are satisfied, then we can create \( t \) the target element in LQN (see Appendix A.1, Else part for Case 1.A). For example, in match C, Example2, \( SendSignalAction1 \) is an old element of type \( SendSignalAction \). It is changed in \( M' \)'s and it does not have a trace. From the S-T mapping pattern we find that its type has constraints and should be mapped to \( Activity \) in LQN. After verifying that the constraints are satisfied, the activity \( SendSignalAction1 \) is created in \( LQN \).

Case 2: If \( e' \) exists in \( M' \)'s but \( e \) does not exist in \( M_S \), this means that \( e' \) is a new element introduced in \( M' \)'s. Its corresponding target element \( t(e') \) needs to be created in \( LQN \), following the same reasoning as above. Later we will discuss how we can add \( t \) to its correct location by adding it to the right container (See Appendix A.2).

Case 3: If \( e \) exists in \( M_S \) but \( e' \) does not exist in \( M'_S \), then \( M_S \) was changed by deleting \( e \) from it. In consequence, we need to check if its corresponding target element \( t \) mapped to \( e \) exists in LQN and the delete it from \( LQN \) by following \( e \) trace (See Appendix A.3).

Step 5: Taking the corresponding action for each type of difference according to its Kind which can be Move, Add, Delete or Change when matching \( e \) with \( e' \) (see Appendices A.4, A.5, A.6, A.7, A.8,A.9).
**Step 6:** Update the traces by adding a trace for each new element created or delete the old trace for each deleted element.

In the next chapter we discuss how to apply those steps in more detailed examples.

### 6.3.4 Potential Extension

In this thesis, the proposed ICP is applied to synchronize UML+MARTE source model with LQN target model. However, it can be extended to propagate the changes from a software model to other analysis models for different non-functional properties (such as reliability, availability, safety) expressed in different formalisms (such as Petri nets variants, Markov Chains, fault trees). The goal is the same: to help integrate the analysis of different nonfunctional properties in the software development process and to improve the quality of the system.

For this purpose, ICP can be realized as a template algorithm with six major steps (presented in section 6.3.3), which represents its structure (see Appendix A). What needs to be specialized to the concrete pair of source and target model are the operations realizing the mapping between source and target (including the traces), as well as the actions for changing the target model (such as Move, Add, Delete or Change) that need to conform to the source and target metamodels.
7 Chapter: ICP Application Cases

In this section, we present the proposed ICP technique and its steps in more details, by discussing four case studies which present different cases of changes. We also discuss the actions needed to incrementally propagate each change to the target model.

7.1 Case study 1: Applying Façade Pattern

![Deployment diagram for Example 1](image)

a. Example 1: original deployment diagram

![Deployment diagram for Example 1](image)

b. Example 1: Changed deployment diagram

Figure 7-1 Deployment diagram for Example 1
a. Example1: Original activity diagram

b. Example1: Changed activity diagram

Figure 7-2 Activity Diagram for Example1
As a first step, we saved $M_S$ as the original model then applied different changes (highlighted in dark grey color) and saved it as $M'_S$. Figure 7-1 (a) and Figure 7-2 (b) represent the structural and behavioral views of Example1 old model, while Figure 7-1(b) and Figure 7-2 (b) represents the changed model $M'_S$. The differences detected by the EMF Compare are marked on the diagram with red numbered circles and discussed with respect to their order through the text.

The original model $M_S$ was transformed to the LQN model shown in Figure 7-3 by applying the batch transformation presented in [1].

The root element of type $LQNmodel$ contains two processors $D1$ and $D2$, and each processor contains a deployed task, $A1$ and $A2$ respectively. Each task has a phase-based entry, whose activities are contained in an element of type $Entryphaseactivities$. The entry of task $A1$ contains an activity called $CallOperationAction1$, which sends a $Synchcall$ to the entry of $A2$ and waits for a reply. The entry of $A2$ contains an activity $CallOperationAction ph1$. The entry ends by sending a reply back to the caller. While the reply is not explicitly modeled in LQN, it is implied by the $Synchcall$ semantics. However, the UML activity diagram from Figure 7-2 (a) that models the behavior of the entry of task
A2 contains a SendSignalAction1 that sends the reply back via a cross-border edge.

EMF Compare matches each element in $M_S$ with each element in $M'_S$ starting from the root. When the two elements are identical, EMF Compare does not detect any differences. We discuss only the matches that generate differences. They are in the same order as generated by EMF Compare.

A. **Change activity partition A2**: Differences in the form \{type, kind, DiffRef, DiffVal\} obtained by matching $M_S!A2$ with $M'_S!A2'$:

1. \{RC, ADD, node, OpaqueAction1\}
2. \{RC, DELETE, node, CallOperationAction ph1\}
3. \{RC, ADD, edge, goToEnd2\}.

This is an example of case 1 (see Appendix A.1) where A2 exists in both models $M_S$ and $M'_S$. As discussed before, the existence of the element in UML does not mean that the element exists in $LQN$. It depends if the element type has constraints or not and if those constraints have been met or not during the transformation. Therefore, we need to check if A2 corresponding target element $t$ (A2) exists also in LQN model by checking if it has a trace or not as in step 4. Since A2 trace has been found, then we get its target element which is Task A2. Please note that A2 is the virtual container of the contained node and edges in the ActivityPartition A2 by applying the Virtual Containers pattern (see Figure 4-13). The next step, as we are able to know the changes suffered by Task A2, is to update Task A2 according to the above differences type and kind (step 5). An update is a composite change that is performed as a set of operations on A2.

**Action for Diff 1**: the first step is to get the type of OpaqueAction1 which is OpaqueAction type and check that is not exists in LQN and does not have a trace. By querying the mapping
patterns specifically S-T Mapping pattern, we get the target type corresponding to $OpaqueAction$ type which is type $Activity$ and check if that type has constraints that need to be satisfied in order to create a new activity. Please note that this example represents one to one mapping (the source type is mapped to one target type). Since the type of the difference is $RC$ which means that the containment reference of A2 has been changed by adding new element named $OpaqueAction1$, we need to check if the target element of $OpaqueAction1$ can be created and added to the containment reference of $Task A2$. For that goal, we get the mapping patterns that are applied in the target model or between the source and target models by matching the structure in the source model ($Artifact, OpaqueAction$) to its corresponding structure in the target model ($Task, Activity$). If there are more than one pattern are applied in the target model or between source and target models as in this case, we need to check which pattern can be followed to find the container of the new activity that needs to be created in LQN model. As we see in Figure 7-4 in the top level, the Containment mapping pattern has been applied where $Task$ plays $tContainer collaborationRole$ and $Activity$ plays $tElement collaborationRole$. However, $Task$ is not the direct container of $Activity$. Therefore, we need to check the other pattern which is Mutually Exclusive Containers that has two mutually exclusive containers, $entry-phase-activities$ and $task-activities$ respectively and connected directly to the $Activity$. Based on LQN metamodel, $task-activities$ is the container of $Activity$ if entry type is $GRAPH$. On the other hand, $entry-phase-activities$ is the container of $Activity$, if entry type is $PH1PH2$.The entry type identifies the container type for the collection of activities modeling the entry behavior. Since we have identified the location or the container of the new $Activity$ which is $entry-phase-activities$. Then, we create a new LQN $Activity$ and initialize its name to
OpaqueAction1 and add it to its container. This is done by invoking operation addNewElement with parameter e’, whose type is Task, name is A2 and DiffVal is OpaqueAction1 and its target type is Activity. Inside the operation, we get the entry of A2 and check its type by invoking isGRAPHPattern operation that returns true if entry is of type GRAPH and false if entry is of type PH1PH2. Some changes to activity partition, for instance adding control nodes of type decision and merge, may affect indirectly the evaluation of isGRAPHPattern operation, which in its turn may affect the entry type, even if the source element for that entry did not change. For that reason, isGRAPHPattern is invoked inside addNewElement operation and update the type of entry according to its returned value (Appendix A.5 summarizing the steps for adding new element). Later, we will see such a case in Example 2. If the entry type is PH1PH2, then the container of the new activity is an entry-phase-activities block. We check if this block has already been created in LQN. If yes, then it already has a collection of activities to which we can add the new activity.

Figure 7-4 Mapping OpaqueAction to Activity

On the other hand, if entry-phase-activities is not defined, we need to create it, then add the new activity to it. Finally, we need to update the trace file, in order to add a new trace for OpaqueAction1 activity. In case of entry type GRAPH, we followed the same previous
steps and add the new activity to task-activities block (if it already exists) or create a new task-activities block and add the new activity to it.

**Action for Diff 2:** From the above discussion, we know that A2 is a Task in LQN, so we need to get the type of the deleted element from Ms, which is CallOperationAction. The next step is to check if CallOperationAction ph1 exists it in the LQN model by checking its trace. If it does exist, we can delete it by deleteElement operation. It takes DiffVal as a parameter and matches it with LQN activity then deletes it. Finally, the traces are updated by deleting CallOperationAction ph1 trace (see Appendix A.6 summarizing Delete steps). Figure 7-5 shows the LQN model after propagating the differences 1 and 2.

![Figure 7-5 Example 1: LQN after propagating diff 1 and 2](image)

**Action for Diff 3:** adding a new reference edge to A2 (the DiffVal is gotoEnd2). Its action has the same steps as the previous ones, except that type ControlFlow is mapped to three target types Synchcall, Asynchcall and Precedence with a set of disjoint conditions; the one that is satisfied will decide the target type of the element to be created (See Figure 4-21). In this case, control flow gotoEnd2 does not satisfy any of the conditions, so no action is needed on the LQN side.

**B. Add new partition F:** Differences in the form {type, kind, DiffRef, DiffVal} obtained by matching M's!F' with null:

4. {RC, ADD, node, AcceptEventAction2}
5. \{RC, ADD, node, SendSignalAction2\}

6. \{RC, ADD, node, NEWCallOperationAction2\}

7. \{RC, ADD, node, goToAction2\}

8. \{RC, ADD, node, goToSendSignal2\}

This is an example of Case 2 (see Appendix A.2) $F'$ is a new ActivityPartition introduced in $M'$, which did not exist before in $M$. We need to create the corresponding target element for $F$ in $LQN$. As discussed in the previous example, applying the Virtual Containers pattern results in having ActivityPartition the same name as the Artifact whose behavior is represented by the activities in that partition. Using this name matching, in the UML model the names of the artifacts from the deployment diagram appears as names of activity partitions in the activity diagram. Based on that, Task $F$ should be created as a target element of $F$ and added to its correct location (i.e., the container of $F$) in $LQN$. In order to do so, we query UML model to get the container of the Artifact which is in this case Device $D_2$. In the trace file, the target element $D_2$ of type Processor corresponds to the source element $D_2$ of type Device, therefore, in the LQN Processor $D_2$ is the container for Task $F$. Finally, we add Task $F$ as a contained element to Processor $D_2$ and update the traces by adding a new trace for Task $F$. After adding $F$ as a Task to $LQN$, we can take actions for its differences.

**Action for Diff 4:** In order to create the target element, correspond to AcceptEventAction2 element of AcceptEventAction type. We follow the same steps we followed in Diff 1. After getting AcceptEventAction2 type we check that it does not have a trace and not exists in LQN. By querying the mapping patterns, we get Entry is the target type for AcceptEventAction type and we get to know that its type does not have specific constraints.
Then we need to find the applied pattern by matching the structure from UML source model composed of \((Artifact, AcceptEventAction)\) with the structure from the LQN target model composed of \((Task, Entry)\). As we see in Figure 4-18 Containment mapping pattern is applied, and \(Task\) is a container of \(Entry\). Now, we can create a new entry and assign its name to \(AcceptEventAction2\) in \(LQN\) by invoking \(addElement\) operation with parameter \(e'\), with type \(Task\), name \(F\), \(DiffVal\) name \(AcceptEventAction2\) and its type is \(Entry\). The new \(Entry\) is added to the containment reference from \(Task\) to \(Entry\). Each entry has a type needs to be initialized to enumerated values (PH1PH2, GRAPH) according to the evaluation of the Boolean operation \(isGRAPHPattern\) which returns false as in this case if entry is type PH1PH2 or true if entry is type GRAPH. Finally, we create a trace for entry \(AcceptEventAction2\). Figure 7-6 shows the LQN model after adding F and propagating differences 4 and 6.

![Figure 7-6 Example1: LQN after creating new task F and propagating diff 4& 6](image)

**Action for Diff 5:** Adding the target element corresponding to the source element \(SendSignalAction2\) of type \(SendSignalAction\) as an \(Activity\) to \(Task\ F\) follows the same steps as for adding element types \(OpaqueAction\) or \(CallOperationAction\) since they are similar, as all are mapped to type \(Activity\). However, only \(SendSignalAction\) type has some constraints should be satisfied in order to create its corresponding target element such as it
is located with its incoming control flow in activity graph type \textit{GRAPH}. After checking those constraints in this case, \textit{SendSignalAction2} element does not satisfy them, therefore no corresponding action is needed to change \textit{LQN}.

**Action for diff 6:** As discussed before, to add \textit{NEWCallOperationAction2} element we need to get the container for \textit{CallOperationAction} type from the applied mapping pattern which is in this case mutually Exclusive containers (see Figure 4-19) and add the corresponding \textit{Activity} to the containment reference from \textit{Activity to entry-phase-activities} since we added in the previous difference a new entry \textit{AcceptEventAction2} of type \textit{PH1PH2}, then we have to get \textit{entry-phase-activities} (if defined), otherwise we have to create a new collection of them. The last step is to add activity \textit{NEWCallOperationAction2} to its container (\textit{entry-phase-activities}) and update the trace file by creating a new trace for \textit{NEWCallOperationAction2} activity. See Figure 7-6 for the modified LQN model.

**Actions for Diff 7 and 8:** are similar to Diff 3, as \textit{goToAction2} and \textit{goToSendSignal2} are both \textit{ControlFlow} typed and none of them satisfies any of the control flow conditions. Therefore, no action is needed to change \textit{LQN}.

**C. ControlFlow Elements without target:** Differences (9-12) result from matching the elements (goToPh1, goToSendSignal1, replyBackToCallOP1, gotoEnd2) that exist in both \textit{M_S}, \textit{M’_S’}. We discuss the above matches together as they are very similar. For all of them, both compared sides e and e’ exists in \textit{M_S} and \textit{M’_S’}, all of them is \textit{ControlFlow} type and the operation checking the traces returns no trace for each of them. This means that none of them has a target element in LQN. As we mentioned before in Diff 3,7 and 8, \textit{ControlFlow} type is mapped to three target types \textit{Synchcall}, \textit{Asynchcall} and \textit{Precedence}. Each type has
different constraints. One of them needs to be satisfied in order to identify the target element type for each control flow. None of the above control flows satisfies any of the its constraints, and then we do not need to take an action to change the LQN.

D. ControlFlow generating Synchcall: Difference in the form \{type, kind, DiffRef, DiffVal\} obtained by matching goToAcceptEventAction1 in \(M_S\) and \(M'_S\).

13. \{RC, CHANGE, target, AcceptEventAction2\}

**Action for Diff 13:** Although this match is similar to the previous ones as both compared sides e (goToAcceptEventAction1) and e’ (goToAcceptEventAction1’) exits in \(M_S\), \(M'_S\) respectively, but the trace checking operation returns true. Therefore, it was easy to identify its target element Synchcall from the trace file. However, the change affects this element's target property. The trace only gives us information about the corresponding target type and the source instance name but does not say anything about affected properties. In order to find out information about the changed properties and their mapping, we need to query the mapping patterns. Operation getPropertyFromMapping takes in two parameters: the changed property name (target) and target type (Synchcall) and queries the mapping patterns and returns the affected property in the target model (which is dest). Since we have all information we need, we can execute updateElement to update Synchcall and assigns its attribute (dest) to target value which is AcceptEventAction2 (see Appendix A.7). Figure 7-7 shows the changed LQN after propagating difference 13.

![Figure 7-7 Example 1: LQN after propagating diff 13](image-url)
E. **New ControlFlow Elements without target.** Differences (14-21) result from matching new elements (goToAction2, goToSendSignal2, replyBackToCallOP2) that exit in $M'$ but do not exist in $M_S$. This case is similar to C Differences (9-12) where there is no need for any action to change the $LQN$.

F. **New ControlFlow generating Synchcall:** Differences in the form $\{\text{type, kind, DiffRef, DiffVal}\}$ obtained by matching $\{M'! \text{goToAcceptAction2'},null\}$.

22. $\{\text{RC, CHANGE, target, AcceptEventAction1}\}$

23. $\{\text{RC, CHANGE, source, NewCallOperationAction2}\}$.

The control flow element $\text{goToAcceptAction2}$ is a new element that exists in $M'$ but does not exist in $M_S$ (see Appendix A.2 for case2). Since its trace checking operation returns false, we need to get its type in order to take action to change the $LQN$ accordingly. Element $\text{goToAcceptAction2}$ is type $\text{ControlFlow}$. Its target type and its constraints can be found by querying the mapping patterns (similar with difference 9). Since checking the constraints determine that the control flow parameter $\text{goToacceptAction2}$ follows the pattern for the Synchronous call, we have to take an action to create its target element (i.e., synchronous call) and add it to its correct container in $LQN$. As we see in Figure 4-21 Activity is the container of $\text{SynchCall}$. Operation $\text{addCall}$ takes control flow’s source and target as parameters, it creates a new $\text{Synchcall}$ target element and assigns its property $\text{dest}$. The source name represents the Activity name that is the container of $\text{Synchcall}$, then we add the new element to its container and update the traces. See Figure 7-8 for the modified LQN and Figure 7-9 shows the updated trace model.
The next step is to check the differences (22, 23). Action for diff 22 is similar to the action for diff 13. Diff 23 does not need an action as source property is not mapped to any property in LQN. We do not discuss other matches because their differences do not need any action to propagate the change to LQN. Some of them do not meet the guard conditions like SendSignalAction2 and SendSignalAction1. Other matches have differences on properties that are not mapped to LQN, such as the ControlFlow properties outgoing, incoming and inPartition.

G. **Container change:** Differences in the form {type, kind, DiffRef, DiffVal} obtained by matching Ms!D2 and M’!D2’:

45. {RC, ADD, nestedClassifier, F}.

D2 has changed when a new Artifact F was added to its nestedClassifier reference. Note that Processor is the container of Task in LQN metamodel (see Figure 4-17). D2 is of type Device and is an old element that exists in both Ms and M’\(_s\). Next, we have to check the
existence of its corresponding target element in $LQN$. The trace checking operation returns true. Since $D2$ has a trace, we can get its target element and target type from the trace. $D2$’s target element is $Processor$ and its name is $D2$. Then, we check if $F$ has a trace or not. The trace checking operation returns true, as $F$ has been added during the matching of $F$. Therefore, we do not need to change Processor $D2$ in $LQN$. If $F$ was not added before, we would get its target type (i.e., Task) from the mapping patterns and then add Task $F$ to Processor $D2$.

![Figure 7-9 Example1: Updated Trace Model](image)

7.2 Case study 2: Changing from sequence of activities to graph

Example 2 introduces other differences (shown in dark grey color) that lead to other kinds of changes in $LQN$ not shown in Example 1. An interesting observation is that some elements in LQN need to be checked even though their corresponding source element in UML has not been changed. The deployment diagram for example 2 is the same as in Figure 7-1 (b). Figure 7-10 shows the original activity diagram of Example 2, where none of the three partitions contains an activity Graph Pattern, and Figure 7-11 shows a changed
activity diagram with activity partition A2 as a Graph Pattern. Figure 7-12 shows the original LQN model in Emfatic Ecore format before propagating changes incrementally.
H. Changes inside a partition: Differences in the form \{type, kind, DiffRef, DiffVal\} by matching $M_s!A_2$ and $M'_s!A'_2$:

1. \{RC, ADD, node, OpaqueAction2\}
2. \{RC, ADD, node, OpaqueAction3\}
3. \{RC, ADD, node, OpaqueAction4\}
4. \{RC, ADD, node, DecisionNode1\}
5. \{RC, ADD, node, MergeNode1\}
6. \{RC, DELETE, node, ActivityFinalNode2\}
7. \{RC, ADD, edge, LeaveDecision1ToOp3\}
8. \{RC, ADD, edge, LeaveDecision1ToOp2\}
9. \{RC, ADD, edge, GoToMerge1FromOp2\}
10. \{RC, ADD, edge, GoToMerge1FromOp3\}
11. \{RC, ADD, edge, GoToOp4\}

12. \{RC, ADD, edge, goToDecision1\}.

**Actions for Diffs 1, 2, 3** are similar to difference 1 in Example 1, as \(A2\) exists in both sides of the match and it has a trace, which means its corresponding target element exists in LQN. The target element for \(OpaqueAction\) type is LQN \(Activity\); we need to create a target element for each source element \(OpaqueAction2, OpaqueAction3\) and \(OpaqueAction4\). \(A2\) Task has entry \(AcceptEventAction1\), of type \(PH1PH2\), so it has an \(entry-phase-activities\) block as container for the collection of activities. As shown in Example1, diff 1, some changes, like adding new control nodes of type decision or merge can affect indirectly the type of entry. For that reason, operation \(addNewElement\) invokes \(isGRAPHPattern\) operation in order to check the entry type. In this case, the type of entry should be updated to type \(GRAPH\), and a \(task-activities\) block should be created instead of the existing \(entry-phase-activities\) block. Since \(A2\) has already a collection of activities, we need to create a new \(task-activities\) block and move that collection to it. According to the Virtual Containers pattern, a \(task-activities\) block, and \(entry-phase-activity\) block cannot exist at the same time, so we delete the \(entry-phase-activities\) block. Figure 7-13 represents an intermediate step for deleting \(entry-phase-activities\) and creating \(task-activities\). As a last step, a new activity with the same name is created for each of the source elements \(OpaqueAction2, OpaqueAction3\) and \(OpaqueAction4\). The new activities will be added to the already existing collection of activities.
Action for Diff 4: the difference here is adding a new node with DiffVal being DecisionNode1. To take an action for this difference, we get the type of the new node (that is DecisionNode) and invoke the trace checking operation that returns false, which means it does not exist in LQN. In order to propagate the difference to LQN we invoke addNewElement operation, which in turn invokes other operation to creates the target elements for DecisionNode1 in the context of A2 task which is the existing container of task-activities block that is in its turn is the intermediate container of Precedence. Figure 4-20 shows containment mapping pattern is applied to map the structure from UML (Artifact, DecisionNode) to the structure from the LQN (Task, Precedence) and intermediate container pattern is applied in the target side to add Precedence and its children (pre, postor) to its intermediate container task-activities. It noteworthy that the
Figure 7-15 Example 2: LQN after adding new Precedence

mapping here is not one to one. Instead, each DecisionNode in UML has three target elements in LQN: Precedence, Pre and Postor elements. The last two are created as children of their Precedence container where Simple containers pattern is applied by default. Each Pre and Post contains elements of type ActivityR, referring to interconnected activities. Finally, we update the traces.

**Action for Diff 5**: This difference is similar to diff 4 since control nodes such as MergeNode and DecisionNode types are mapped to Precedence. MergeNode1 is mapped as a Precedence, Post and Preor in LQN. So, we follow the same steps to create a Precedence element and its children. Next, update the trace file. Figure 7-14 represents LQN after propagating diffs 1 through 5.

Differences from 6 to 12, do not need any action for changing LQN. DiffVal for diff. 6 is of type Activity FinalNode that is not mapped to any type in LQN. Regarding the
differences from 7 to 12, all of them are of type control flow and none satisfies any of their constraints.

A. **ControlFlow generating Precedence:** Differences in the form \{type, kind, DiffRef, DiffVal\} obtained by matching $M_s! \text{goToSendSignal1}$, $M'_s! \text{goToSendSignal1}'$:

13. \{RC, ADD, source, OpaqueAction\}.

![Figure 7-16 Example2: LQN after adding reply entry and replyactivity](image)

$\text{goToSendSignal1}$ of type ControlFlow exists in $M_s$ and has been changed in $M'_s$. However, it does not exist in $LQN$ as it has no trace. As discussed before ControlFlow is mapped to three different target types based on the evaluation of specific constraints. After testing each of them, $\text{goToSendSignal1}$ satisfies Precedence Constraints. Now we are able to create a Precedence and its two children Pre and Post in $LQN$ as target elements and create
new trace to update the trace model.

After creating the precedence for `goToSendSignal1` control flow (see Figure 7-15), we can check the generated difference on its property `source`. Since `source` is a property not mapped to any property of `Precedence` in `LQN`, there is no need for any action.

J. **SendSignalAction generating activity**: Differences `{type, kind, DiffRef, DiffVal}` obtained by matching `Ms! SendSignalAction1` and `M's! SendSignalAction1`:

14. `{RC, DELETE, outgoing, goToEnd2}`

Although `SendSignalAction1` is an old element that exists in both `Ms` and `M's`, its target element does not exist in `LQN` as there is no trace corresponding to it. As mentioned earlier, type `SendSignalAction` is mapped to `Activity` and has constraints should be satisfied. The reason behind such case is the fact that “`SendSignalAction1`” does not satisfy those constraints during the initial execution of batch transformation. In this case we need to check if the change affects them, so they became true. The constraints need to be satisfied are a) if property incoming of type `ControlFlow` (named `goToSendSignal1` in this case) is located inside the activity partition and b) its target element `SendSignalAction1` is part of a `GRAPH` pattern. As both conditions are satisfied, we need to create a new `LQN Activity` called `SendSignalAction1` and add it to the correct location (i.e. container) and create a `Replyentry` and a `Replyactivity` in `LQN`. This mapping is one to many because creating new activity corresponding to `SendSignalAction` type also triggers creating `Replyentry` and `Replyactivity`. It is considered a special application of S-T mapping pattern as it involves navigation of the target model since `Replyentry` and `Replyactivity` are children to `Task-activities` not to `Activity` as in the case of type `Precedence` and its children. In order to create the corresponding target elements for `SendSignalAction` type, first we need to find the
container of $SendSignalAction1$ that is type $ActivityPartition$ corresponds to an $Artifact$ which is mapped to a $Task$ in $LQN$ by applying Virtual container pattern. Then, we need to find the pattern that can be followed to find the container of the new $Activity$ by matching a structure from $UML$ source model ($Artifact$, $SendSignalAction1$) to as structure from $LQN$ target model ($Task$, $Activity$). Containment mapping pattern is applied in this case, $Task$ plays $tContainer$ collaborationRole. However, $Task$ is not the direct container of $Activity$; therefore, we need to search in the target side for other patterns. Mutually Exclusive containers pattern is applied in the target side. As discussed in Diffs 1,2,3 $Task$-activities is the direct container for the existing collection of activities and the new Activity $SendSignalAction1$ should be added to them. $Replyentry$ and a $Replyactivity$ should be created and added to $Task$-activities (See Figure 7-16). The Last step is updating the trace model as shown in Figure 7-18. The graphical representation of the $LQN$ model before and after propagating all changes is shown in Figure 7-17.

We do not discuss other matches in order to avoid repetition, as they are similar to those already discussed. Some differences do not need any action to propagate the changes to $LQN$ because the guard condition is not met. Other matches have differences on properties that are not mapped to $LQN$.

### 7.3 Case study 3: Changing container (Move)

As mentioned before, the proposed approach is one of few approaches that support MOVE. In this example, an $Artifact$ “F” is moved from $Device$ “D2” to $Device$ “D1”. Figure 7-19 and Figure 7-20 respectively represent the deployment diagram before moving $Artifact$ F and after moving it. Please note that activity diagram did not change, and it is the same as Figure 7-2 (b) in Example 1. The following change is detected by
Figure 7-17 Graphical representation of LQN model before and after propagating all changes

Figure 7-18 Example 2: Updated Trace model
A. **Move element to a new container:** Differences in the form \( \{ \text{type, kind, DiffRef, DiffVal} \} \) obtained by matching \( M_S!D2 \) with \( M'_S!D2 \):

1. \( \{ \text{RC, Move, nestedClassifier, F} \} \).

**Action for Diff 1:** From the above difference we can tell that element F is moved from its container D2. After retrieving type *Artifact* for F and *Device* for D2. The next step is to identify the new container of the moved *Artifact* F from \( M'_S \), which is *Device* D1, the namespace that represents the new container for F. As we discussed before, *Device* is mapped to a *Processor* and *Artifact* is mapped to a *Task* in LQN. From matching the structure \( (\text{Device, Artifact}) \) from UML to the structure \( (\text{Processor, Task}) \) from LQN, we find the Containment mapping pattern (see Figure 4-17) that we can follow to move *Task* F to its correct container. Since *Processor* is the direct container for *Task*, we can take the action by executing operation moveElement that takes *Artifact* F and matches it with *Task* F and adds it to the collection of tasks of \( D1 \) processor (Appendix A.4 summarizing Move algorithm). Figure 7-21 shows the original LQN and the new LQN after moving task F to processor D1.

![Deployment Diagram](image)

Figure 7-19 Deployment Diagram before moving Artifact
7.4 Case study 4: Updating stereotype attribute

In chapter 2 we mentioned that the UML source model is annotated with MARTE profile. In this example we show the capability of the proposed approach to propagate changes in stereotype attribute to its corresponding model element attribute in LQN. (Note that LQN does not use stereotypes). In Figure 7-22 the value of attribute Host demand from «PaStep» stereotype has been changed to 1.5. EMF Compare detects the following difference:

4. Change stereotype attribute value: Differences in the form \{type, kind, DiffDiscriminant\} obtained by matching $M_s!CallOperationActionph1$, $M_s'!CallOperationAction ph1$:

   1. \{SAC, CHANGE, hostDemand\}.

Action for Diff 1: hostDemand is an attribute for «PaStep» stereotype applied to the UML element type CallOperationAction, SendSignalAction or OpaqueAction. These three types
are mapped to LQN Activity type. By following the trace, we can find the corresponding target element and its type (Activity CallOperationAction ph1 in this case) but trace cannot help in finding the performance property (hostdemandmean) that corresponds to the changed stereotype’s property (hostDemand). Such information can be obtained by querying the mapping patterns where S-T mapping pattern is applied to map stereotypes properties to performance properties. The action for this difference is done by calling Operation updateStereotype() to update the Activity’s property (hostdemandmean) that matches DiffDiscriminant, which is hostDemand in this case. The operation gets the new value for hostDemand from M’s and assigns it to hostdemandmean property of the LQN Activity mapped to CallOperationActionph1 (see Appendix A.8 for Update stereotype algorithm). See the changed LQN model in Figure 7-23.

![Figure 7-22 Example4: Activity diagram after updating Host demand of CalloperationActionPh1](image-url)
Figure 7.23 LQN after updating Hostdemandmean of CallOperationActionph1 Activity
8 Chapter: ICP Evaluation

This chapter evaluates the incremental change propagation approach proposed in chapter 6. The ICP case studies already presented cover two kinds of changes: a) atomic changes (add or delete model elements one by one); and b) composite changes (add, delete, update and move more elements at a time). Section 8.1 of this chapter presents another two case studies by applying two design patterns to the source model: Secure Communication and Proxy patterns. Each design pattern was applied by hand to a UML model and then we used the version without and with pattern as $M_S$ and $M'_S$ models. For instance, the changes from Figure 7-2 (a) to Figure 7-2 (b) are inspired from the Façade design pattern. In this chapter also the correctness of LQN is evaluated in section 8.2 and the run time efficiency for ICP is measured in section 8.3.

8.1 Evaluating the Correctness of ICP

In this section, we describe the approaches followed to verify whether the proposed ICP produces correct LQN models after propagating the changes from the evolved UML model. Please note that the correctness of the evolved UML model should be verified by the designer in order to produce a correct LQN by ICP.

We verify syntactic correctness, which means that the target model conforms to its metamodel after propagating all changes, and semantic correctness, which means that a correct source model is transformed in a correct target model (i.e., the transformation is correctly executed) [112]. Syntactic correctness can be verified automatically with tools, as explained in the next section. However, semantic correctness cannot be verified automatically, only manually (a human analyst inspects the transformation result and decide whether the transformation produced the right outcome). For example, if an object
x is split into objects x1 and x2 (to solve some performance bottleneck), then verifying that the behavior of x is correctly handled by x1 and x2 needs a human analyst. The verification took place, either after propagating the changes, or during the ICP, as explained below.

**Verifying ICP correctness with EUnit testing framework**

This approach was developed using EUnit testing framework [11] provided by Epsilon to test model management tasks, such as model to model transformations. EUnit tests facilitate the comparison between models and the tests can be reused between different set of models. EUnit consists of ANT and EOL files. The ANT file provides an orchestration workflow solution to load the source and target models and their metamodels, to execute the transformation and load the expected target model. EOL files are used to invoke the helper target defined in the ANT script to execute the ETL transformation. The comparison between the actual and the target models is launched by an EOL file and it is done by calling EMF Compare internally.

After propagating all the changes from the evolved UML model to the LQN model, we execute EUnit tests with the following objectives: 1) to re-execute the batch transformation that takes evolved UML models as source and generates the corresponding LQN models and trace models; and 2) to compare two LQN models for each evolved UML model: one generated by re-executing the batch transformation and the other obtained via the ICP approach.

If the compared models are not identical, EUnit testing provides a graphical interface to visualize the differences between the compared models and shows a red bar to indicate the
failures of the testing. If the tests pass, EUnit shows a green bar to indicate success. For all cases we used to verify the ICP implementation, EUnit bar was green.

**Verifying the ICP correctness with the LQN solver**

Another approach used in this thesis to verify LQN correctness is by solving the updated LQN model with LQN solver as it generates errors if the model does not conform to the LQN metamodel. All cases were accepted by the LQN solver.

**Verifying the ICP correctness during the change propagation**

Please note that the updated LQN model is verified not only after completing the incremental change propagation, but also during the execution of ICP. EOL run configuration has the advantage that it checks every LQN model change against the LQN metamodel and if any action to update the LQN model violates the metamodel constraints, the EOL run configuration displays an error message.

### 8.2 Case studies

#### 8.2.1 Secure communication pattern

This pattern helps in securing the interaction between two entities by using security protocol (SSL) [113] that is common for securing web-based transactions [114]. In chapter 3, the e-commerce system has been discussed with its three scenarios *Browse Catalogue*, *Browse Cart*, and *Place Order*. In this case study, we apply the Secure Communication pattern to *Browse Catalogue* scenario (see Figure 8-1) to enhance the security between the user and the customer interface. This change occurs after LQN was generated (see Figure 8-4) and breaks the consistency between the UML software model and LQN. ICP has been executed to propagate all differences due to applying the Secure Communication pattern from UML+MARTE model to the LQN model to keep them synchronized. The tool EMF
Compare detected 137 differences between Ms (UML+MARTE before applying secure communication pattern) and M’s (UML+MARTE after applying secure communication pattern). The differences result from changing the deployment diagram by adding two new artifacts SSL-R1 to device RemoteProc and SSL-S1 to device UserDevice1, as well as new activity partitions matching the names of the new artifacts to represent their behavior. The new activity partitions SSL-S1 and SSL-R1 were added between User1 activity partition and CustomerInterface activity partition (see Figure 8-2). Please note that Figure 8-2 does not show CustomerProc and Catalogue Server partitions, as there were no differences detected in them. After propagating the differences, LQN model (see Figure 8-3) has two new tasks SSL-S1 and SSL-R1 that were added to processor RemoteProc and processor UserDevice1 respectively. Task SSL-S1 has a new entry Accept Call Request and Task SSL-R1 has a new entry Accept Call request2. Each entry created with a sequence of new activities (see Figure 8-3). Please note that Figure 8-3 shows part of LQN model in EMF format. Figure 8-5 shows the graphical representation of LQN after propagating the changes due to applying secure communication pattern.

![Activity Diagram of Browse Catalogue scenario before Secure communication pattern](image)

Figure 8-1 Activity Diagram of Browse Catalogue scenario before Secure communication pattern
Figure 8-2 Activity Diagram of Browse Catalogue scenario Secure communication pattern

Figure 8-3 LQN model after change propagating
Figure 8-4 Graphical representation of LQN before applying Secure Communication pattern
Figure 8-5 Graphical representation of LQN after applying Secure Communication pattern
8.2.2 Virtual Proxy pattern

Let consider that the designer needs to improve the performance of the system by applying Proxy pattern that helps in caching additional information about an object to use the memory only when is needed and postponing expensive computation [115]. To accomplish that purpose, we applied Virtual Proxy pattern to Browse Cart scenario (see Figure 8-7). Therefore, a new artifact called Proxy is added to User Device1 with a new activity partition matching the name of the new artifact to represent its behaviour. Proxy activity partition is composed of a graph with decision and merge node. It was added between Customer Interface activity partition and Customer Process activity partition. Change detection phase by EMF Compare results in 109 differences between Ms (UML+MARTE before applying proxy pattern Figure 8-6) and M’s (UML+MARTE after applying proxy pattern Figure 8-7). As a result of propagating the change the old LQN model (Figure 8-8) was updated to have a new Task named Proxy with new entry type Graph with graph of new activities (Figure 8-9).

![Activity diagram for Browse Cart scenario before Proxy pattern](Image)
Figure 8-7 Activity diagram for Browse Cart scenario after Proxy pattern
Figure 8-8 LQN model before applying Proxy pattern
Figure 8-9 LQN model after Incremental Change Propagations
8.3 Evaluation of ICP Run time efficiency

The aim in this section is to study the impact on the ICP execution time and different model properties:

- Performance impact of the *type of change*. For instance, we notice that creating new element is more expensive than updating an attribute of an existing element.
- Performance impact of the *number of changes*.
- Performance impact of the *size of the model*. As expected, the incremental transformation is faster than the batch transformation especially for large models.

In order to measure the execution time for the ICP and batch transformation, which are both implemented in Epsilon EOL and ETL languages, we used the “Epsilon Profiler“ a tool provided by the Epsilon framework to measure the performance metrics for an Epsilon code [116]. In all experiments we measured the ICP execution time while running it for the e-commerce model and its different versions. This model was proposed in [5] and our version has almost 300 model elements. We chose it since it is not a very small model, but not very large either. In all experiments, we focused in four types of changes: create element (Add), update attribute of an element (Change), update stereotype attribute (Change), and Move. These types of changes are common to occur and have different impact on the model. The experiments were executed on Intel(R) Core™ Processor i7 CPU, 64-bit OS, x64-based processor.

8.3.1 ICP execution time versus type of change

The goal of conducting such experiments is to study the effect of the type of change on the execution time of the ICP. We applied the same number of changes, measured the execution time for each type of change, and compared them. We conducted four
experiments. Each experiment has two rounds for the same type of change and for each round there are two versions of e-commerce model: one version with one change (V1), the second version (V2) with 10 changes. For each round, the experiment was repeated 10 times and we took the average value for the execution time.

As we see in Figure 8-10, the execution time for adding a new element is the highest. This result is expected since creating a new element is considered as an expensive change because the new element’s location needs to be identified, and then the element needs to be created in the target model. In addition, a new tracelink needs to be created for every newly created element. On top of that, creating new elements has a greater effect on other elements in the target model. For instance, in the next experiment we see that adding two new elements generates 16 differences between the old and the evolved models, which also need to be investigated in order to decide whether to take action or not for propagating them to the target model. The next expensive change type is “Move”. In this case, not only the moved element is affected but also the old container the element moved from and the new container the element moved to are also affected. The least expensive execution times are for the change type “update attribute” and “update stereotype attribute”. Those two types of change have an effect only on the element itself and do not affect other elements such as create or move.

8.3.2 ICP execution time versus number of changes

In these experiments, we conducted four experiments to propagate changes from the modified e-commerce model (M’s) to LQN. In each case, we applied and tested one type of change and for each type; we increased the number of changes. We focused on the same four types of changes presented in the previous section. Our aim is to study the effect of
increasing the number of changes on different types of change varies on their effect on the model. As a result, we noticed that the run time of the incremental approach increased with the increasing of the number of changes. Each experiment has three rounds with three versions (V1, V2, V3) of the e-commerce model. In each round, we increased the number of changed elements for the same type of change, which creates different version of the model in each round. For each round, the experiment was repeated 10 times and we took the average value for the execution time.

As mentioned before, ICP starts with the change detection by EMF Compare. Such a detection takes place only one time at the beginning of ICP and so its overhead does not slow down the incremental approach. EMF Compare is able to detect differences between two UML models that have around five thousands elements in a few seconds[117]. In the first round of the first experiment, the e-commerce model (Mₛ) was changed by creating a new element of type OpaqueAction and a new element of type control flow to connect the new action with the old actions. The change detection results in 16 differences as this type of change affects other elements that has relations with newly created elements in the model. In the second round, two new elements of type OpaqueAction and two control flows were created in Mₛ and thus the number of differences doubled to 32. In the last round three new elements with three control flows were created in M’ₛ, which tripled the number of differences to 48. In the second experiment, we applied another type of change, which moves an element from its container to a new container. We executed three rounds of experiments, moving 1, 7 and 10 elements, respectively, to new containers. For each moved element, EMF Compare found one difference. Again, we executed three rounds, updating the hostDemand value for 1, 6 and 12 elements. For each update, EMF Compare
detected two differences (Add value and Delete value). The number of differences in each round is 2, 12, and 24 respectively. In the last experiment, we chose update attribute name for an old element. In same the way as previous experiments, we increased the number of changed elements in the three rounds 1, 10, 30 respectively. As we notice

Table 8-1 Execution time for ICP based on number of changes

<table>
<thead>
<tr>
<th>Type of Change</th>
<th>#E-commerce model versions</th>
<th>#elements changed</th>
<th>#of differences generated by EMF</th>
<th>ICP Execution time(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>V1 2</td>
<td></td>
<td>16</td>
<td>207.5</td>
</tr>
<tr>
<td></td>
<td>V2 4</td>
<td></td>
<td>32</td>
<td>314</td>
</tr>
<tr>
<td></td>
<td>V3 6</td>
<td></td>
<td>48</td>
<td>412.7</td>
</tr>
<tr>
<td>Update Stereotype attribute</td>
<td>V1 1</td>
<td></td>
<td>2</td>
<td>67.8</td>
</tr>
<tr>
<td></td>
<td>V2 6</td>
<td></td>
<td>12</td>
<td>72.7</td>
</tr>
<tr>
<td></td>
<td>V3 12</td>
<td></td>
<td>24</td>
<td>312.8</td>
</tr>
<tr>
<td>move</td>
<td>V1 1</td>
<td></td>
<td>1</td>
<td>117.3</td>
</tr>
<tr>
<td></td>
<td>V2 7</td>
<td></td>
<td>7</td>
<td>408</td>
</tr>
<tr>
<td></td>
<td>V3 10</td>
<td></td>
<td>10</td>
<td>564</td>
</tr>
<tr>
<td>Update attribute</td>
<td>V1 1</td>
<td></td>
<td>1</td>
<td>51.7</td>
</tr>
<tr>
<td></td>
<td>V2 10</td>
<td></td>
<td>10</td>
<td>77.8</td>
</tr>
<tr>
<td></td>
<td>V3 30</td>
<td></td>
<td>30</td>
<td>286</td>
</tr>
</tbody>
</table>

Figure 8-10 Comparing ICP execution time for different types of change
from Table 8-1 the execution time for the incremental approach is increasing with the increasing of the number of changes.

8.3.3 ICP execution time versus the size of the model

In the previous experiments, we did not consider the execution time of the batch transformation because it does not care about the number of changes or the type of changes. For example, to update name of one element or to create a new element all transformation rules need to be re-executed. In this experiment, we compare the execution time of ICP with the batch transformation. We applied the same number and type of change on different model sizes: 20 elements, 300 elements and 1000 elements. The experiments were repeated 10 times for each model size, and we took the average value for the execution time. Figure 8-11 shows that ICP took only 154 ms in average to update name of one element when the model has 1000 elements and batch took 8691 ms in average to update name of one element for the same size of model. We challenged ICP and batch transformation by applying expensive change (create new element) on the largest model (1000 elements). As Figure 8-12 shows the execution time of ICP was 866 ms in average while the execution time for the batch did not change (8691 ms in average). The reduction of the execution time of the incremental approach was 90% with respect to the batch transformation.
Figure 8-11 Execution time for different size of models when apply update attribute change

Figure 8-12 Execution time for 1000 elements model size when apply create new element change
9 Chapter: Conclusion

This chapter summarizes the thesis contributions, discusses the limitations, and provides directions for future work.

9.1 Completed work

The overall goal of the thesis is to help with the integration of formal analysis models for Non-Functional Properties (NFP) into the Model-Driven Engineering process, in order to guide the design choices from an early stage and to ensure that the system under construction will meet its non-functional requirements. More specifically, this research focusses on the integration of performance analysis based on Layered Queueing Network (LQN) models in the model-driven engineering process using UML. The analysis models can be automatically derived by model transformations from the software models extended with MARTE profile performance annotations. The contributions of the thesis are concerned with three aspects: a) containment-based mapping patterns, b) performance analysis roundtrip and c) incremental change propagation to keep the software and performance models synchronized during model evolution.

In model transformations, the understanding of the source and target metamodels is an essential requirement for mapping the source model elements to target model elements. Once the mapping is identified at a high-level of abstraction, it can be implemented as a transformation with any available model transformation language. The main advantage of the separation of concerns between mapping and the implementation of the transformation is the independence of the mapping from the language used for implementing the transformation. Thus, the designers have the opportunity to focus more on the conceptual aspects of the relations between source and target (meta)models and have flexibility in
choosing the language for implementation. The thesis proposes a set of *containment-based mapping patterns* which are used to map a group of containment-related source model elements into a group of target model elements. Such mapping can be reused in different model management tasks. For instance, in the thesis the proposed mapping patterns are applied in a specific context to map UML+MARTE model to LQN performance model. The mapping was reused to design both batch and incremental transformations to help the designer to identify what should be transformed to what and to shift the focus from the implementation of the mapping to the mapping itself.

Another objective is to help designers to improve their design by feeding back the performance results to the software model via a *performance analysis roundtrip*. First, we extend the batch transformation to generate, beside the target model, a traceability model containing trace links between the mapped source and target elements. After generating the performance model and solving it with an existing solver, the performance results are fed back to the software model by following the cross-model trace links. Having the performance results on the same view with the UML software model helps designers to evaluate different alternatives and change the software model to improve the performance characteristics early in the software development.

The changes in the software model due to the design decisions break the consistency between the software model (UML+MARTE) and performance model (LQN). The proposed Incremental change propagation (ICP) main goal is to keep the source and target models consistent. ICP starts with detecting the changes with the EMF Compare tool and propagates them automatically from the UML+MARTE model to the LQN performance model. ICP propagates atomic and composite changes and keeps not only the LQN model
synchronized with the evolved UML model, but also updates the trace model, which was created at the beginning of the transformation process by a batch transformation.

The thesis evaluates the developed ICP according to different criteria:

a. Evaluate the syntactic and semantic correctness of the performance models produced by ICP, measuring the effect on the ICP execution time of the following model properties: type of change, number of changes and size of the model.

b. Evaluate the execution time of the ICP incremental approach compared with the batch transformation for different model sizes

9.2 Limitations

- So far, the mapping patterns have been applied “manually” to the design of transformation rules. However, steps could be taken toward automation.

- So far, the Incremental change propagation was implemented only in the context of the Epsilon family of languages. Other model transformation languages could be considered.

- The proposed ICP is applied between UML+MARTE as source model and LQN as target model. To prove the generality of the ICP algorithm, it needs to be applied to other pairs of source and target models.

- Find new kinds of grouping of the model elements based on different relationships other than containment that appear in other metamodels and may lead to new mapping patterns.

9.3 Directions for future research

- Extend the roundtrip analysis to other nonfunctional properties such as reliability, availability, security and safety.
• The mapping patterns have been used so far for “manual” design of the transformation rules. More work needs to be done to generate automatically the transformation model from the patterns by using a Higher Order Transformation (HOT) approach.

• Extend the incremental change propagation approach to consider other analysis models to help in improving other nonfunctional properties and the overall quality of the system. The proposed ICP algorithm is a generic algorithm that can be applied to other source and target models. For this purpose, we identified different kinds of transformation rules that generate target model elements from source model elements, identify the kinds of modifying actions corresponding to different rules and apply them to the target elements cross-linked with source elements that suffered a change.

• Extend the mapping patterns by adding new patterns to the catalogue of the proposed containment-based mapping patterns.

• A high-level grammar can be used for the definition and application of the mapping patterns which are specified by a UML-based notation in this thesis.
Appendices

Appendix A: Summarized ICP Algorithm

Appendix A.1: Summarized ICP Case 1 Algorithm

```
Matches.forAll (m | m.differences.isNotEmpty());

Get differences for each match m;

CASE 1: {m.e’ ∈ M_s’ and m.e ∈ M_s | m.e’ is changed}

If (e’. isDefined()) { // if e' exists in M_s’
  e’.name: =get e’.name;
}

If (e.isDefined()) { // if e exists in M_s
  e.name: =get e.name;
  E: = get e.type; // get e type from source model
  MappedSourceTypes: =getAllSourceTypes(); //get all source types mapped to target
  types from mapping patterns

  If (MappedSourceTypes.includes (E)) { //check if E is included in the collection of
    all mapped source types
      For (each diff ∈ differences) {
        //Case1.A
        If (∃ λ = (e’: E, t: T) and ∃ t(e’)) {
          // check if there is a tracelink for e’ and t (e’) exists in M_T
          // Take an action for every diff between e and e’ according to its type and kind

          If (diff.type = ReferenceChange) { // e’ changed by adding or deleting or moving an
            DiffElement: =DiffVal //The element added or deleted or moved and ∈ M’s
            DiffElementName=get the name of the DiffElement added or deleted or moved to/from
            the containment reference of e’;
            DiffElementType: =get the type DiffElement added or deleted or moved to/from the
            containment reference of e’;
            TargetTypeForSourceContainer: =Get T; //get target type T for E from the trace file
            TypeNameForSourceContainer: = Get t (e’); //get target element t for e’ from the
            trace file
            If (∃ λ = (DiffElement: E, t: T) and ∃ t (DiffElement)) and Kind =MOVE) {
              Follow algorithm Appendix A.4 to move element;
            } //End-if

            If (¬∃ λ = (DiffElement: E, t: T) and Kind =ADD) {
              Follow algorithm Appendix A.5 to add new element;
              Update the trace file by adding new tracelink for the new element;
            } //End-if

            If (∃ λ = (DiffElement: E, t: T) and ∃ t (DiffElement)) and Kind = DELETE) {
              Follow algorithm Appendix A.6 to delete element;
          }
        }
      }
    }
  }
} //End-Case1.A
```

140
Update the trace file by delete the tracelink for the deleted element;
} //End-if

Else (kind=CHANGE) {
    Follow algorithm Appendix A.7 to update element;
} //End-if
} // End-if

If (difference type ="StereotypeAttributeChange" and kind="CHANGE") {
    Follow algorithm Appendix A.8 to Update Stereotype attribute;
} // End-if

If (difference type =AttributeChange and kind=CHANGE) {
    Follow algorithm Appendix A.9 to Update Attribute;
} // End-if

Else (¬∃ λ = (e': E, t: T)) and (¬∃ t (e') ϵ M_T) {// Else part for case1.A when e' and e both exist in M's, Ms and t (e') not exist in M_r
    Check if the E type has constraints need to be satisfied;
    TargetTypeOfE: =Get target type T for type E from the mapping patterns;
    NameOfSourceContainer: = Find e’ container name from M’s;
    TypeOfSourceContainer: = Find e’ container type from M’s;
    TargetTypeForSourceContainer: =Get T; // get the target type of TypeOfSourceContainer from the mapping patterns
    TargetNameForSourceContainer: = Get t (e’); //get the name of target element of TargetTypeOfSourceContainer
    Match {E, TypeOfSourceContainer} with {TargetTypeOfE, TargetTypeForSourceContainer} to get the pattern name that can be applied to check if TargetTypeForSourceContainer can be the container of the element of TargetTypeOfE t(e’) that needs to be created;
    Call addNewElement () to create an element t(e’) of TargetTypeOfE and add it to its TargetNameForSourceContainer;
    Update the trace file by adding new trace link for the new element t;
    After adding t take an action for its differences according to its type and kind as in Case1.A;
} End-if

} //End-For-loop
} //End If
//End if
} //End if
CASE 2: match {m.e’ ∈ M’s, null}

If (e.isDefined()) { // if e’ exists in M’s
    e’.name: = get e’.name;
    E: = get e’.type;
}

If (e.isDefined()) { // if e does not exist in Ms
    TargetTypeOfE: = Get target type T for type E from the mapping patterns;
    TargetNameForSourceContainer: = Get t(e’); // get the name of target element of TargetTypeOfE;
    TargetTypeForSourceContainer: = Get T; // get the target type of TypeOfSourceContainer from the mapping patterns;
    Match {E, TypeOfSourceContainer} with {TargetTypeOfE, TargetTypeForSourceContainer} to get the pattern name that can be applied to check if TargetTypeForSourceContainer can be the container of the element of TargetTypeOfE t(e’) that needs to be created;
    Call addNewElement () to create an element t(e’) of TargetTypeOfE and add it to its TargetNameForSourceContainer;
    Update the trace file by adding new trace link for the new element t;
    After adding t take an action for its differences according to its type and kind as in Case1.A;
}  

TargetTypeOfE: = Get target type T for type E from the mapping patterns;
TargetNameForSourceContainer: = Get t (e’); // get the name of target element of TargetTypeOfSourceContainer

Appendix A.3: Summarized ICP Case 3 Algorithm

CASE 2: match {null, m.e ∈ Ms}

If (e.isDefined()) { // if e exists in Ms
  e.name: = get e.name;
  E: = get e.type;
}

If (e'. isNotDefined()) { // if e’ does not exist in M’s
  If (∃ λ = (e:E, t:T )and ∃ t(e))) { // check if e has a trace and its target element t exists in the MT
    Get t(e) from trace file; // get target element t for e from the trace file
    Delete t from M;
    Update the trace by deleting t trace link;
  }
}

Appendix A.4: Summarized Algorithm for Move Difference

MovedElementTargetType: =Get t (DiffElement) from the trace file; // get the target element t for the DiffElement that represents the moved element

NewContainerSourceTypeOfMovedElement: =Get the type of DiffElement’s new container from M’s;

NewContainerSourceNameOfMovedElement: = Get the name of DiffElement’s new container from M’s;

NewContainerTargetTypeOfMovedElement: = Get T; //get the target type T for

NewContainerTargetNameOfMovedElement: = Get t; //get the target element t for new container of the moved element from the trace file

Call moveElement() to move MovedElementTargetType and its name DiffElementName to its new container NewContainerTargetNameOfMovedElement;
Appendix A.5: Summarized Algorithm for Add New Element Difference

Check if DiffElementType has constraints needed to be satisfied;
NewElementTargetType: = Get the target type T for DiffElementType from the mapping patterns;
Match \{DiffElementType, E\} with \{NewElementTargetType, TargetTypeForSourceContainer\} to get the pattern name that can be applied to check if TargetTypeForSourceContainer can be the container for NewElementTargetType;
Call addNewElement() to create a new element of NewElementTargetType and add it to TargetNameForSourceContainer and assign the new element name to DiffElementName;

Appendix A.6: Summarized Algorithm for Delete Element Difference

DeletedElementTargetType: = Get the target type T for the deleted element of DiffElementType from the trace file;
DeletedElementTargetElement: = Get the target element name t for the deleted element DiffElementName from the trace file;
Call deleteElement() to delete DeletedElementTargetElement;

Appendix A.7: Summarized Algorithm for Change Difference

Check if e’ type has constraints that need to be checked after the change, so we check if it stills satisfies its constraints otherwise its target element t needs to be deleted;
mappedDiffRef: =Get the mapping of \{e’. DiffRef\} from the mapping patterns; // DiffRef is a reference name that its value (DiffElement) is changed
TargetElementName: =Get t (e’); // Get the target element of e’ from the trace file
Call updateElement() to update TargetElementName. mappedDiffRef.value in the target model with the value of DiffElementName;

Appendix A.8: Summarized Algorithm for Update Stereotype attribute Difference

Get DiffDiscriminant; //the name of the stereotype attribute that changed
TargetElementName: =Get t (e’); // Get the target element of e’ from the trace file
Get the TargetElementName.attribute that mapped to the changed stereotype attribute denoted by DiffDiscriminant;
Call updateStereotypeAttribute() to update TargetElementName.attribute with the value of changed stereotype attribute denoted by DiffDiscriminant;
Appendix A.9: Summarized Algorithm for Update attribute difference

Get DiffAttr; // the name of changed attribute
TargetElementName: = Get t (e'); // Get the target element of e' from the trace file
Get the TargetElementName.attribute that mapped to the changed attribute denoted by DiffAttr;
Call updateAttribute () to update TargetElementName.attribute with the new value of DiffAttr in the target model;
References


NFPinDSML@MoDELS, 2009.


