Synthesis of Execution Plans for the QVT Core Language

Horacio Hoyos Rodríguez, MSc.

EngD

University of York
Computer Science

July 2017

Abstract

Model transformation languages (MTLs) are important for Model Driven Engineering as they allow the automation of the engineering design process of hardware and software products, in particular at the preliminary and detailed design phases. However, the theories from compiler optimization have not been reused substantively in the development of MTLs. This makes the challenges associated with the implementation of declarative MTLs harder to overcome, in particular with respect to the synthesis of the execution plan (a representation of the control component of the transformation algorithm). The QVT Core MTL is a declarative language, part of a set of standards proposed by the Object Management Group® in order to support the adoption of Model Driven Engineering (MDE). This research presents how instruction scheduling theories can be used for the synthesis of execution plans, in particular for the QVT Core language. The main contributions are a novel approach for performing data dependence analysis on the QVT Core language and its use for the synthesis of execution plans, and the application of metaheuristics to solve the scheduling problem inherent to the synthesis of execution plans. The research demonstrated the feasibility of applying compiler optimization techniques in the design of MTLs and provides a methodology that can be used to construct efficient execution plans that result in correct transformations. The performance gains and correctness will help the widespread use of the QVT Core language and encourage the adoption of compiler optimization techniques in the implementation of other MTLs.

Contents

List of Tables

List of Figures

to my parents, for giving me the opportunities that led me here

"be anything you want to be" *they said*
"be a shoe shine boy if that makes you happy" *they insisted*
"but be the best-damn shoe shine boy!"

Acknowledgements

First and foremost, I would like to thank my supervisor Dr. Dimitris Kolovos who was incredibly supportive throughout my entire EngD. Second, thanks to my assessor Dr. Radu Calinescu who provided helpful insight and guidance in the early stages of the research.

Although away, my family was always supportive and encouraging, thanks a lot to them for believing in me and for their emotional boosts. Thanks to my partner, Emilia, for the sacrifices she made and for always supporting me and my long hours of work.

I'd also like to thank my desk-mate, Adolfo, for his ideas and comments on numerous discussions and for making office time (and off-time) entertaining.

I'd like to acknowledge the contributions of Dr. Edward Willink in the establishment of some fundamental concepts of this work and his role as industrial supervisor. I'd like to thank Willink Transformations for their financial sponsorship through the EPSRC EngD program. Finally, I'd like to thank the EPSRC whose financial support via the Large-Scale Complex IT System grant (EP/F001096/1) has made this research possible.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author, except where stated. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References. Section 2.4.2, Chap. 3 and Sect. 4.4 present collaborative work and clearly state the contributions of this author and the collaborators. Parts of this thesis have been previously published in the following:

[114] Willink, Edward, Hoyos, Horacio, Kolovos, Dimitris: Yet Another Three QVT Languages. In: Theory and Practice of Model Transformations. Ed. by K. Duddy and G. Kappel, pp. 58–59. Springer Berlin Heidelberg(2013).

[53] Hoyos Rodríguez, Horacio, Kolovos, Dimitrios: Declarative Model Transformation Execution Planning. In: Brucker, Achim D. Cabot, Jordi, Herrera, Adolfo Sánchez-Barbudo (eds.) Proceedings of the 16th International Workshop on OCL and Textual Modeling (OCL), Satellite event of MODELS. (2016).

Part I

Onset

1

Introduction

Some challenges associated with building larger and more complex systems, such as simulation, formal verification and modelling, can be solved by raising the level of abstraction used to represent the system as a whole or as a decomposition of sub-systems, to represent specific system artefacts, or to represent the system at specific design phases. An effect of the system's size and complexity, and of the differences in the design phases, is that it is often the case that different abstraction levels are used as part of the abstraction effort. In order to bridge the semantic and syntactic gap between different abstraction levels the system design process should be automated as much as possible. Automation is key in order to receive the benefits in productivity gain of the abstraction effort [41].

Model Driven Engineering (MDE) is a type of development process that can ease the implementation of system artefacts and to automate the design phases of these artefacts [87]. MDE promotes two activities as part of the development process [60]. First, the use of models as the realization of an abstraction of the system under design[1]. That is, a model is used to represent a particular abstraction of the system. The main goal of MDE is to allow mod-

[1]For the rest of the discussion when referring to the *system* it means the system as a whole or its components.

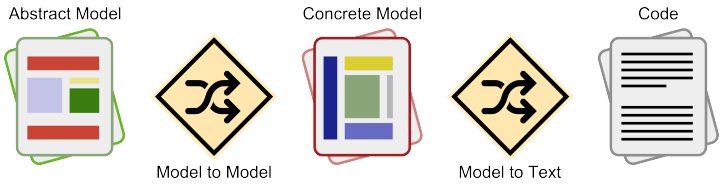


Figure 1.1: Typical Workflow of MDE Software Construction

els to be used to design, test, analyse, explore and implement the system and its components. Second, the use of model management languages (a particular type of programming languages) in order to validate, analyse, construct and transform models. Of the various existing model management languages, model transformation languages (MTLs) are of particular interest as they allow the automatic synthesis of alternative model representations [87]. In the context of this project, synthesis is understood as the combining of separate entities into a single or unified entity. Given that models can belong to different abstraction levels, MTLs are vital to enable MDE to support automation.

A typical use of MDE is the automation of software construction, as presented in Fig. 1.1. The application code is generated from a model that describes a software application at a high abstraction level, by a series of Model–to–model (M2M) and model–to–text transformations. This research only considers M2M transformations. In its simplest form, a M2M transformation takes as input a source model and produces as output a target model. Note that in general, the models in a M2M transformation are not required to be at different abstraction levels.

1.1 Motivation

A MTL is a Domain Specific Language (DSL) tailored specifically to the domain of model transformations. Most of the existing MTLs follow an imperative, declarative or hybrid paradigm. In imperative languages, the program describes both the logic component (what to do) and the control component (how to do it). In

declarative languages, the program only describes the logic component and the control component must be synthesized before or during execution of the program. Hybrid languages support mixed descriptions.

In some scenarios, declarative MTLs are preferable because "particular services such as source model traversal, traceability management and automatic bidirectionality can be offered by [the] underlying reasoning engine." [72]. This research project focuses on the challenges associated to the execution of declarative MTLs, in particular to the synthesis of the control component. This research centres its attention to the particular case of the QVT Core [3] (QVTc)[2] - declarative - transformation language, one of the languages proposed by the Object Management Group® (OMG®) in their framework for MDE: Model Driven Architecture (MDA) [2, 17]. The OMG® is an international, open membership, not-for-profit technology standards consortium, that develops enterprise integration standards for a wide range of technologies and industries. Another notable OMG®'s modelling standard is the Unified Modelling Language®(UML®).

In a QVTc program the logic component specifies a set of constraints between the involved (source/target) models. The model transformation is the result of modifying the involved models in order to satisfy these constraints. The aim of this work is to automatically compute (synthesize) control components that, when merged with the logic components and executed, are guaranteed to modify the models such that the constraints are satisfied. That is, they result in correct transformations.

One of the expected advantages of using DSLs, in general, is runtime efficiency [93]. The result of an optimization process in computer science is a software system that works more efficiently and/or uses fewer resources. Optimization can be done in different aspects of the system. The interest of this research is optimizations

[2]QVT is an initialism for Query/View/Transformation, which comes from the full name of the OMG's specification that defines the language: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification.

that improve the performance of the execution of QVTc programs, in particular with respect to execution time. In the case of MTLs, the control component is accountable for the execution time of the QVTc program. This translates to designing synthesis strategies that exploit particular aspects of the MDE domain in order to produce efficient control components. In the context of this project, a synthesis strategy is the plan or method used for combining the separate entities into a single or unified entity. Thus, the challenge is to synthesize *correct* and *efficient* control components.

Efficiency requires a base–line to compare to. For the QVTc language, given that during the field survey of this project no other implementations of QVTc were found, a brute-force approach is used as the base–line. The brute-force approach, referred hereafter as the *naïve* approach, consists in executing the QVTc program's logic component in an arbitrary order. At this point it suffices to consider a QVTc program as a set of operations. Thus, in the *naïve* approach these operations are invoked in an arbitrary order, an indefinite number of times, until a predefined termination condition is encountered.

Correctness is achieved by guaranteeing that the synthesized control components respect the semantics of QVTc. Thus, achieving efficiency is the main drive of this research project. This research explores how compiler theory for General Purpose Languages (GPLs) can be adapted and applied to the synthesis of the control component. In particular, we will explore the theories related to instruction scheduling. Instruction scheduling aims at modifying the order of execution given by a control component in order to improve the efficiency of a program. Control component synthesis can be viewed as a special case of instruction scheduling, in which the control component does not define an initial order of execution. Given that the instruction scheduling problem is NP-hard [109, 44], this research explores the use of metaheuristics to synthesize the control component. The use of heuristics means that the best possible schedule is not found, but rather a good-enough

solution.

Data dependence analysis is at the core of instruction scheduling in compiler design [111, 10, 44]. The adoption of this principle is a key component in the synthesis of the control component for QVTc programs proposed in this research. The novel idea of this research is that by formulating a strategy to perform data dependence analysis on QVTc programs it is possible to construct their associated control component.

The choice of working with the QVTc language is due to its role in the MDE efforts. The QVT Specification [3] is part of a theoretical underpinning that is envisaged to allow the construction of an "agreed-on systematic theory of model transformation design" [89] and execution [114]. We anticipate that providing a fully compliant implementation of the Meta Object Facility (MOF) Query/View/Transformation (QVT) declarative languages would greatly benefit the MDE community. The benefits include, but are not limited to:

- Writing and executing model transformations using a language defined in a standard.
- Other transformation languages/engines could be benchmarked against a standard.
- Future research can concentrate on formulating a systematic and reliable theory of constructing model transformations [89] instead of being spread across the definition of non-standard languages.

Systematic synthesis of the control component for programs written in the QVTc language is a first step towards positioning QVTc as one of the pillars of the systematic theory of model transformation design.

Model transformations are key to the MDE effort [28, 70, 61]; the ability to execute transformations efficiently would greatly improve the benefits associated to using model transformation in dif-

ferent aspects of the MDE process, including the automation of the design process of hardware and software products.

1.2 Research Question

The question this research answers is:

> *Can correct and efficient control components for programs written in the QVTc language be systematically synthesized for QVTc transformations?*

where efficiency is measured against the *naive* approach.

In order to answer this question, this research project has three hypotheses. The first two are directly related to the research question. The third is a consequence of the use of metaheuristics in the proposed approach.

Hypothesis 1 *[Hypothesis] For a given QVTc program, execution of the synthesized control component is guaranteed to be correct.*

Hypothesis 2 *[Hypothesis] For a given QVTc program, execution of the best synthesized control component has a better performance than the* naive *approach.*

Hypothesis 3 *[Hypothesis] For a given QVTc program, the best synthesized control component has a good performance among all the synthesized control components for that program.*

Hypothesis 3 mentions *all* the synthesized control components as for a given QVTc program it is possible to construct multiple control components.

The objectives of this research project are:

- To identify how data dependence analysis can be applied to declarative MTLs, in particular QVTc.
- To design and implement a data dependence analysis tool that can extract the data dependence information of any

(and all) Model Transformation Program (MTP) written in QVTc.

- To design a model to represent execution plans for QVTc programs.
- To design and implement a tool that can construct efficient execution plans by exploiting the data dependence information.
- To implement a tool that merges the logic component with the execution plan to provide an executable QVTc program (compilation).
- To evaluate the correctness of the constructed execution plans, and use the knowledge gained from the evaluation to propose improvements to the synthesis of execution plans.
- To evaluate the efficiency of the constructed execution plans, and use the knowledge gained from the evaluation to propose improvements to the synthesis of execution plans.

1.3 Thesis Contributions

The primary contributions made in this thesis are summarised below:

- A novel methodology for synthesizing the control component for MTPs written in the QVTc language.
- The demonstration of the feasibility of using the systematic synthesis to construct control components for a variety of MTPs written in the QVTc language.
- The formalization of *transformation correctness* for the QVTc language.
- The formulation of a novel approach of identifying data dependencies in MTPs written in the QVTc language.

- The mapping of the QVTc control component synthesis problem to the Ant Colony Optimization (ACO) metaheuristic.

The secondary contributions made in this thesis are summarised below:

- The definition of a novel representation for the control component of MTPs written in the QVTc language.
- The formulation of the control component synthesis problem as a scheduling problem.
- The implementation of a metaheuristic control component synthesis framework for the QVTc language.
- The implementation of a merge operation to merge the synthesized control component with the base QVTc MTP.

1.4 High–Level Overview

Figure 1.2 presents a high–level overview of the proposed solution. The QVTc MTP is first translated① into QVTm in order to eliminate bi-directionality and normalize the transformation rules, which is required by the data dependency analysis. The details of these step can be found in Sect. 2.4.2. The data dependency analysis② is responsible for extracting the data dependency information from the MTP. This process results in two artefacts: the data dependence graph (DDG) discussed in Sect. 4.2 and the input variable derivation information discussed in Sect. 4.5. The DDG is then transformed③ into a foraging area (FA), which is a representation more amenable for the ACO algorithm④. The details of the FA can be found in Sect. 5.2.1, while Sect. 5.3 presents the implementation details of the ACO algorithm. The ACO algorithm produces an execution plan (the best found) which is then merged⑤ with the derivation information and the original QVTm MTP to produce the complete QVTi MTP. The details of the

merge are discussed in Sect. 5.5. The evaluation of the approach is presented in Chap. 6, where the performance of the synthesized plans are compared against the *naïve plan* approach, and also against each other in order. For the former, even small performance gains are important particularly in cases when the MTP is going to be executed repeatedly. The latter is used to determine if the best solution found is in fact the best performing solution.

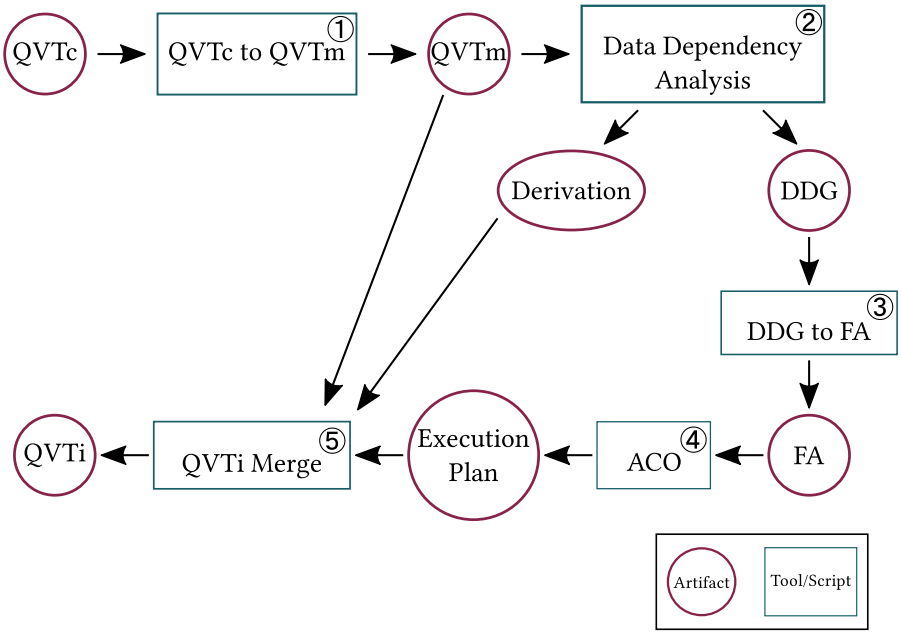


Figure 1.2: High-level overview of the proposed solution.

1.5 Thesis Structure

Chapter 2 presents a review of related literature. The chapter starts by introducing the MDE domain, discussing the role of model transformations within MDE, the key characteristics of model transformations and presenting the QVT Specification (which defines the syntax and semantics of the QVTc language). Next, it presents a review of other transformation languages and highlights the state of the infrastructure used as a base for the development effort of the research project. Finally, it provides an introduction

to the problem of instruction scheduling and describes the metaheuristics used in the proposed approach.

Chapter 3 provides a more detailed description of the QVT language, with particular focus on the features relevant to the control component synthesis problem. The chapter then presents an in–depth analysis of the QVT language semantics in order to propose a model to represent the control component for QVT transformations. This chapter also provides a formal definition of *transformation correctness* for QVT transformations.

Chapter 4 defines the control component synthesis problem as a scheduling problem. It describes how data dependence analysis is applied to a QVTc transformation in order to provide a solution for the control component synthesis problem. The chapter introduces the concept of *thoroughness* as an additional condition for constructing valid control components. Finally, it shows how data dependence analysis can be extended in order to provide local optimizations.

Chapter 5 further specializes the control component synthesis problem by representing it as a combinatorial optimization problem. The chapter shows how data dependence analysis is used to define an objective function for the combinatorial optimization problem and then presents how the ACO metaheuristic can be used to solve the combinatorial optimization problem. Next, the chapter provides details on how the problem can be represented for use with the ACO and then describe in detail the ACO implementation. The results of running the ACO implementation on a set of development examples are summarized and discussed. Additionally, the chapter presents the naïve control component used as a baseline to evaluate the solutions found by the ACO implementation.

Chapter 6 presents an evaluation of the control component synthesis. The evaluation consists of two main parts. In the first one, the focus is on the observations of the behaviour of the ACO implementation with respect to its ability to find a solution in

a reasonable amount of time and its exploration of the solution space. In the second part the focus is on the evaluation of the correctness and the performance of the synthesized control components. The performance analysis also provides validation of the objective function that is defined in Chap. 5.

Chapter 7 summarizes the thesis contributions, discusses the results and presents potential new/future research areas that emerge from the work presented here.

2

Field Survey and Review

This research presents a novel approach to the synthesis of the control component of Model Transformation Program (MTP)s written in the QVT Core [3] (QVTc) language. The proposed approach builds on proven methodologies from the domain of compiler design, in particular on the use of data dependence analysis for instruction scheduling. The main contribution of this research is the adaptation of these methodologies to the domain of model transformation languages (MTLs), in particular for the QVTc language.

This chapter provides an overview of Model Driven Engineering (MDE) and MTLs, of the use of data dependence analysis for instruction scheduling in compilers, metaheuristics and the use of metaheuristics for instruction scheduling. The concepts introduced are considered necessary to understand this thesis.

The purpose of Sect. 2.1 is to provide an introduction to MDE and an overview of the characteristics of model transformations. Section 2.2 introduces the Meta Object Facility (MOF) Query/View/Transformation (QVT) languages followed by a review of existing implementations and other model transformation languages/engines in Sect. 2.3. Next, Sect. 2.4 presents the details of the existing infrastructure used for the development of our solution. Section 2.5 gives an overview of instruction scheduling

in compilers and the use of data dependence analysis, and Sect. 2.6 introduces the use of metaheuristics for instruction scheduling. Finally, Sect. 2.7 summarizes and concludes the discussion.

2.1 Model Driven Engineering

MDE is a type of development process that can ease the implementation of system artefacts and to automate the design phases of these artefacts. MDE promotes two activities as part of the development process [87]. First, the use of models as the realization of an abstraction of the System under Design (SUD). Second, the use of model management languages (a particular type of programming languages) in order to validate, analyse, construct and transform models.

A model of a system is regarded as an abstract representation of the system that suppresses uninteresting details. Thus, the abstraction process can be considered as a process in which models of the SUD are constructed at different abstraction levels. The benefits of representing the system through various models is that these models can be used, among other things, to predict the SUD properties and behaviours [31], and for SUD analysis, synthesis and verification [41].

In order for models to be effective they must be well-defined and provide clear and unambiguous semantics. In the Model Driven Architecture (MDA) [2] framework proposed by the Object Management Group® (OMG®), the semantics for models follows a 3+1 layer architecture as illustrated in Fig. 2.1.

> A model represents a system at level M1. This model conforms to its metamodel defined at level M2 and the metamodel itself conforms to the meta-metamodel at level M3. The meta-metamodel conforms to itself [15].

In the automation process, metamodels are used to define the concepts, semantics and restrictions of the different abstraction

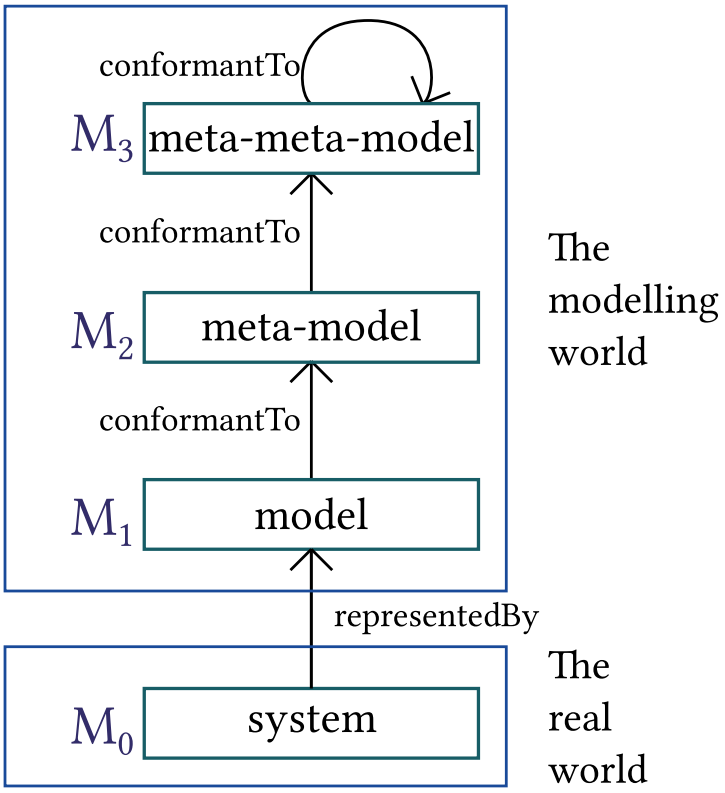


Figure 2.1: The 3+1 level MDA architecture (taken from [14])

levels. The Eclipse Modelling Framework (EMF) [95] is a very mature and widely used framework (M_3 level) for specifying metamodels and models. The EMF can be considered a pragmatic implementation of the Meta Object Facility (MOF) [4] standard, which is recommended by MDA for the M_3 level.

The relations between two adjacent layers in the MDA define type-token distinctions, as observed in the object oriented paradigm [64], as follows: If a model (M_a) conforms to metamodel (MM_a), then for a given model element $X \in M_a$ and its associated metamodel element $Y \in MM_a$, the type-token relation is given by $meta(X, Y)$, where $meta(X, Y) \leftrightarrow type(X) = Y$. The various relations between model and metamodel elements are summarized by the relation $conformsTo(M_a, MM_a)$ [15]. This relation holds at the global level. Further, subtyping is also supported at the metamodel level, allowing the notion of substitutability to be observed at the model level. That is, anywhere where an element X with

$type(X) = Y$ is expected, an element Z with $type(Z) = W$ can be used iff W is a subtype of Y.

The main aim of MDA is to define an approach that will provide the tools to materialize the "idea of separating the specification of the operation of a system from the details of the way that system uses the capabilities of its platform" [2]. Thus, in the context of MDA, the SUD is designed at two levels: the platform–independent model (PIM) and the platform–specific model (PSM). However, abstraction can happen not only vertically (PIM to PSM) but also horizontally (PIM to PIM, PSM to PSM) [72]. Horizontal abstraction can represent a refinement process, vertical abstraction a realization process and, when the PSM to PIM direction is considered, the vertical abstraction can also represent a reverse engineering process [60].

2.1.1 Model Transformations

As mentioned previously, MDE enables the automation of the design process by promoting the use of models and of model management languages. Model management languages, in particular model transformation languages, play a key role in this endeavour as they provide the tools to define and execute scripts that describe how a model of the SUD at an abstraction level can be automatically transformed to a model of the SUD at another (or the same) abstraction level.

At its essence, a model transformation maps concepts from a metamodel (read as abstraction level) to concepts of another metamodel. If object o_a belongs to a model that conforms-to metamodel MM_a, and it has to be transformed into an object o_b that belongs to a model that conforms-to metamodel MM_b, then a transformation specifies how the attributes of o_a (defined by MM_a) can be used to construct o_b and populate its attributes (defined by MM_b).

In simple terms, execution of a model transformation "takes as input a model conforming to a given source metamodel and produces as output another model conforming to a target meta-

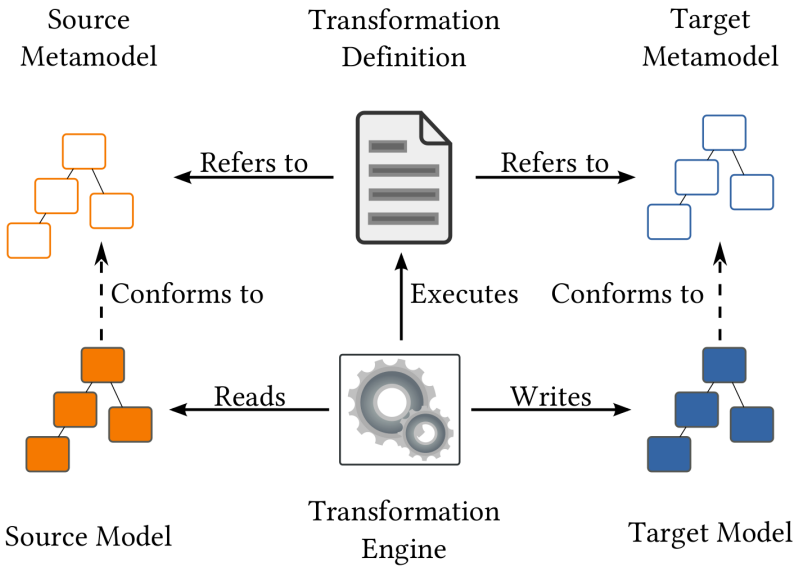


Figure 2.2: Model Transformations.

model" [30], as presented in Fig. 2.2. The transformation is specified in a *transformation definition*, or simply the MTP. MTPs are executed in a transformation engine (the term execution engine is used interchangeably).

MTLs are used to express MTPs. In general, MTLs are Domain Specific Languages (DSLs), although some approaches use General Purpose Languages (GPLs) to express model transformations (see Sec. 2.3.2). The benefits of using a DSL for model transformations are the same as those observed for DSLs in other domains: Expressiveness is preferred over generality; this expressiveness facilitates ease of use, with gains in productivity and (arguably) reduced maintenance costs [73]. DSLs also make it easier for domain experts to work with the language as their notations and constructs are usually closely related to the domain.

The features and characteristics of MTLs (and their engines), have been summarized and classified by Mens et al., and Czarnecki et al. The taxonomy proposed by Mens et al. [72] defines a set of qualities that can be used to group tools, techniques or formalisms used for model transformations. On the other hand, Czarnecki et

al. [28] carry out an analysis of the model transformation domain and provide an in-depth overview of the variabilities and commonalities of existing model transformation tools. The remainder of this section discusses the features and characteristics of the QVTc language in the frame of the aforementioned classifications. In particular, it focuses on the classifications that are closely related to the data dependence analysis and the correctness of the transformation.

2.1.2 Model Transformation Mechanisms

This categorisation [72] refers to the techniques from major programming paradigms which are used by the transformation language (and its execution engine or interpreter). Is the transformation language declarative or imperative? Is the transformation executed via interpretation or compilation?

The QVTc language is a domain specific, declarative, rule based, MTL. The declarative aspect is the main feature that drives this research. Being declarative, a QVTc MTP only describes the logic component (what to do) of the transformation algorithm, and the control component (how to do it) must be synthesized either at compilation or during execution. This research proposes a method to synthesize the control component during compilation, following the approached presented in Sect. 2.4.2. This allows the use of existing compiler design techniques, specifically instruction scheduling, during the synthesis process.

An important aspect of the language is that it is rule based. By considering rules as the smallest units of a transformation [28] this research will consider them as the starting point for data dependence analysis. Section 4.2.1 defines the data dependence relations between rules.

Another important aspect of the transformation mechanism is tracing support.

This is concerned with the mechanisms for recording

> different aspects of transformation execution, such as creating and maintaining trace links between source and target model elements [28].

In some transformation languages the tracing is provided in the form of a trace model, which holds the information of links between source and target model elements. The trace model can be implicit or explicit. If implicit, the transformation engine is responsible for defining its structure (possibly by defining a proprietary trace metamodel) and for maintaining it. Further, the implicit case precludes the use of existing trace models as additional information available to the execution engine. That is, the implicit model is always constructed from the ground up during execution.

If explicit, the programmer is responsible for defining the trace model's structure and for maintaining the information it contains. Maintaining the information implies that the MTP must correctly create the links between source and target model elements as required. In this case, the trace may contain information from previous executions (or created otherwise). In the QVTc language, the trace is explicit. Section 2.2 describes how the language supports the use of an explicit trace model.

2.1.3 What Needs to be Transformed into what

This classification refers to the format of the source and target models (MOF models, graphs, trees, databases, text, etc.), the number of candidate models, the technical space (the domain of the M3 level), endogenous versus exogenous (transformations of models expressed in the same language versus expressed in different languages), horizontal versus vertical (referring to the level of abstraction of the models) and syntactical versus semantical transformations [72]. From these, the format of the source and target models and the technical space are of interest to data dependence analysis. The former will place restrictions on the information that

can be extracted from the metamodels and how models are accessed. The latter will determine the information that is available for data dependence analysis.

Regarding the technical space, the selected QVTc execution engine (see Sect. 2.4) supports the EMF at the M3 level. The EMF provides an Application Programming Interface (API) that allows clients to query a metamodel for information about the concepts it defines (types), their attributes and the relations between them (e.g. subtyping). This information will be crucial for the data dependence analysis. The use of the EMF also determines the format of the source and target models. As with the M3 level information, the EMF provides an API that allows clients to query models for information about their contents (elements), and also modify existing elements and/or create new ones. Each individual element can be queried about its attributes, including its *meta* relations. The available interactions with a model will determine how the synthesized control component will access these models.

Another important aspect is that the QVTc language allows the specification of *n-way transformations* [28], that is, transformations between more than two models. To support this, the QVTc language provides a syntax in which each of the models accessed by a rule is distinguishable from the others but in which there is no clear distinction between source and target models. Although during execution a single model must be specified as the target model since any model can be the target hereafter all models involved in the transformation will be referred to as the candidate models. When an execution direction is defined, we will then refer to the specific source model and the target model.

Additionally, the language uses syntactic typing [28], where variables are associated with a metamodel element and as a result defines the type of elements it can hold. This will facilitate the data dependence analysis because it allows identification of the elements' *meta* relations by using static analysis. Static analysis is fundamental in order to perform the control component synthesis

during compilation. Section 3.1 provides more details about the QVTc language syntax, Sect. 4.2 shows how the language features are used to perform the data dependence analysis and Sect. 2.4 presents details of the selected QVTc execution engine.

2.1.4 Verifying and Guaranteeing Correctness of the Transformations

This aspect refers to questions such as, does the transformation produce well-formed models from well-formed input models? Does the transformation always complete, and are its results deterministic? Considerations regarding this aspect include:

- Can the transformation be tested (or its correctness proved if the transformation language has a mathematical underpinning).
- How are incomplete or inconsistent models treated (failure/error classification, user assistance).
- Can the transformation be executed as a series of smaller transformations.
- Is the transformation bi-directional.
- Is a trace mechanism provided (useful for debugging).

The QVTc language is a bi-directional MTL. However, the QVT Specification [3] states that a QVTc transformation must be executed in a particular direction. Further, although it supports defining *n-way transformations*, during execution the selection of a single model as the target model implies that the QVTc transformation is in practice a *many-to-one* transformation. As such, data dependence analysis must be performed in such a way that the direction is taken into consideration.

A model will be classified as inconsistent when it does not conform to the metamodel it claims to conform-to. Inconsistency can

refer to the model containing elements of types that are not defined in the metamodel, elements missing required attributes, and references to missing elements, among others. Dealing with inconsistent models is delegated to the EMF as part of the loading process of the source model(s) (the EMF will fail to load inconsistent models). As a result, during evaluation, this research assumes that models used as source model(s) is(are) consistent. As a result, assuming that the MTP is free of errors, execution of the transformation should produce the expected target model.

Other important aspects are composition and tracing. The QVTc language does not support transformation composition and thus we do not consider the case of execution as a series of smaller transformations. Since in QVTc the trace model is explicit, this means that trace metamodel types must be also considered for the data dependence analysis.

Given that the QVTc language does not have a formal semantics, correctness of the synthesized control component cannot be proved. Therefore, evaluation of correctness can only be achieved empirically. Finally, it is important that the synthesized control components guarantee that the execution will always complete and that the results are deterministic.

2.2 The QVT Specification

Model Transformations are fundamental to MDA.

> It is important that transformations can be developed as efficiently as possible. A standard syntax and execution semantics for transformation [a transformation language] is an important enabler for an open MDA tools chain [42].

To this end, the OMG® issued the QVT Request For Proposals (RFP) [6] in 2002. The QVT RFP brought forth eight submissions. From these, Gardner et al. [42] presents a set or recommendations

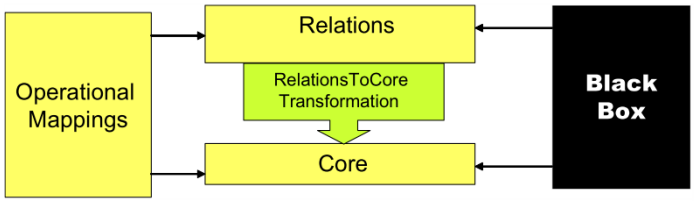


Figure 2.3: QVT Architecture (taken from [3])

for the final QVT specification. Some of these recommendations, like support for hybrid transformations, a simple declarative specification language, bi-directionality, composition and reuse, are present in the final QVT specification. It is important to notice that although the specification is still called Query/View/Transformation, the query and view components are not defined in the specification. It could be argued that both are special cases of transformations.

The QVT specification "has a hybrid declarative/imperative nature, with the declarative part being split into a two-level architecture" [3], as presented in Figure 2.3. The declarative nature is provided by the QVT Relations [3] (QVTr) and QVTc languages, while the imperative nature is provided by operational mappings language. In the rest of this thesis, the QVT Declarative (QVTd) acronym is used when referring to the Relations and Core languages.

The QVT languages are all intended for model-to-model transformations. Transformations can be endogenous or exogenous as well as horizontal or vertical. The languages also support multiple input and output models and in the case of the QVTd also support bi-directional transformation specifications.

The QVTr language is more abstract than the QVTc and QVT Operational Mapping [3] (QVTo) languages, and intended for end users. The QVTc language is defined using minimal extensions to the Essential MOF (EMOF) and Object Constraint Language (OCL) and is intended to provide the semantics of the Relations language. This was one of the main reasons that the QVTc language was selected for this research. By providing an execution

engine for QVTc, then QVTr should be, in principle, supported too. Further, the QVTc could be used by other transformation languages as a compilation target language in order to provide execution capabilities [114]. The *Operational Mappings* language provides OCL extensions with side effects. Operational Mappings can be invoked from the Relations or the Core languages. Finally, the *Black Box* mechanism adds extension support by allowing operations to be provided by external libraries.

Usually models involved in a transformation are referred to as input/output or source/target models, denoting an explicit direction in the transformation (the input/source model is transformed into the output/target) model. However, considering the multi-directionality and multiplicity of models involved in a QVT transformation, the term *candidate* models is used throughout this thesis to refer to the models involved in the transformation (as defined in the specification). In places where a differentiation is necessary the input/output or source/target models naming is used. Input/source and output/target will be used interchangeably depending on the references cited.

2.2.1 The Relations Language

As specified in the QVT specification,

> In the relations language, a transformation between candidate models is specified as a set of relations that must hold for the transformation to be successful. A candidate model is any model that conforms to a model type [3],

where, as defined in the specification, a *model type* is defined by a metamodel. The QVTr language is a multi-directional declarative language where "traces between elements involved in the transformation are created implicitly" [3].

A QVTr MTP consists of *Relations* that specify constraints that must be satisfied by the elements of the candidate mod-

els. The multi-directionality is defined by the number of domains defined in a Relation (minimum two). Two language constructs, **when** and **where** clauses, define the relationships that must hold between the elements of the candidate models. The *when* predicate expresses pre-conditions and the *where* predicate expresses corollaries (statements and rules that should execute after execution of the rule's body). The QVTr language does not support refinement or extensions of Relations; only an override semantics is provided. That is, transformation rules cannot be refined or extended.

The QVT specification defines the semantics of Pattern Matching (how variables in a Relation are bound to elements in a model), Enforce and Checking. The last two are defined as QVTr execution modes. In *Enforce* mode, the candidate models are modified so that they satisfy the relations in a MTP; in *Check* mode the candidate models are checked to assess if they satisfy the relations in a MTP.

2.2.2 The Core Language

In the QVTc language,

> a transformation is specified as a set of mappings that declare constraints that must hold between the model elements of a set of candidate models and the trace model. The candidate models are named, and the types of elements they can contain are restricted by a model type [3].

The QVTc language is a multi-directional, declarative transformation language where the trace model must be explicitly defined and the mappings in a transformation refer to this model directly. Thus, a QVTc MTP consists of mappings that specify the constraints between the candidate models and the trace model.

As in QVTr, multi-directionality is defined by the number of domains defined in a mapping, with the difference that in QVTc mappings can have zero or more domains. This particular distinc-

tion allows a QVTc mapping to define constraints between elements of one (or more) model types and the trace model, or even constraints between elements of the trace model only. A QVTc transformation can also be executed in Check or Enforce mode.

In a nutshell, QVTc is a textual, declarative, rule based model transformation language that reuses OCL as an expression language for model navigation and modification. A QVTc mapping (rule) is built around the concept of areas, as illustrated in Fig. 2.4. Each area consists of a *bottom* pattern and a *guard* pattern. The guard pattern defines the constraints that a set of elements must satisfy in order to be processed by the mapping. The bottom pattern defines additional constraints the set of elements must satisfy and modifications that the elements must undertake as part of the transformation.

An area is related to a particular model. When the area is related to a source or target model the area is called a domain. In Fig. 2.4 there are two domains, one for the L (left) model and another for the R (right) model. Each mapping is in itself an area. All mappings (Middle Area) are related to the trace model and provide the mechanism for managing the (explicit) tracing.

As presented in Sect. 2.1.2, the programmer must define its structure (via a metamodel) and is responsible for creating the links between the source and target model elements. The *bottom* pattern and a *guard* pattern allow the user to define constraints between the elements of the trace model, and between elements of the trace model and the candidate models. Although explicit, the QVTc semantics do not make it mandatory to use a trace model. As a result there are no restrictions either on transferring information directly between candidate models.

Patterns are specified as a set of variables, predicates (constraints) and assignments (modifications). The QVT specification defines the semantics for pattern matching and bindings:

> A match of a pattern results in zero or more valid bindings. A binding is a unique set of values for all

> variables of the pattern. A valid binding of a pattern is a binding where all the variables of the pattern are bound to a value other than undefined, and where all the predicates of the pattern evaluate to true [3].

The arrows in Fig. 2.4 define the dependencies between patterns and help define the context of bindings. These relations are important for data dependence analysis as they provide an initial guide on what relations are important and a possible order in which dependence should be performed, e.g. start at the guard patterns.

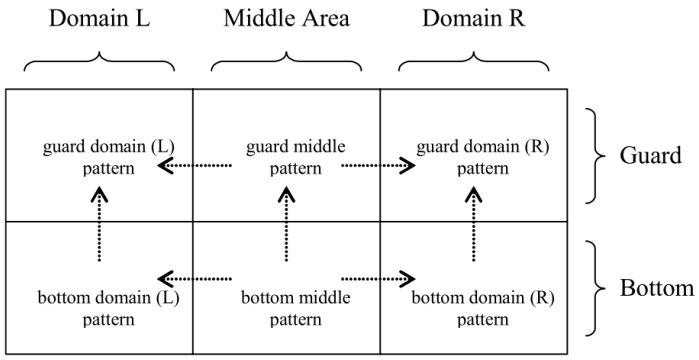


Figure 2.4: QVTc Areas and Pattern Dependencies (taken from [3])

2.2.3 Operational Mapping

This research focuses on the declarative subset of the QVT specification (QVTr and QVTc) and thus a detailed presentation of QVTo is out of the scope of this survey. For completeness this sections presents a small overview and highlight the key differences from the QVTd languages.

The QVTo language allows "to define transformations either using a complete imperative approach or allows complementing relational transformations with imperative operations" [3]. A QVTo transformation is specified as a set of *mapping operations* in which objects (elements) can be created and modified. The QVTo language supports both transformation and mapping composition.

The language also supports a *forEach* structure that allows iteration over a block of expressions. The language provides implicit tracing facilities and language constructs to access the trace information.

2.3 A QVT Landscape

Almost 14 years after the adoption of the QVT specification and the initial interest in the RFP, the available QVT transformation engines are scarce, and none of them support all three languages. In 2008, eight QVT tools where reported [68]; one for QVTc, four for QVTr and two for QVTo. Of those, the following provide QVTc or QVTr support:

- The OptimalJ (QVTc) project was abandoned by Compuware that same year.
- MediniQVT [62] provides a partial implementation of QVTr but its performance has been reported to be unsatisfactory [21].
- MOMENT-QVT claims to have conformance with QVTr (except with respect to SyntaxExportable[1] [85]) but it is no longer available from the project website[2].
- ModelMorf is a proprietary tool from Tata that is under development, the latest publicly available version is a Beta release[3]. Tata has recently authorized the use of their test cases in the Eclipse QVTd Project.

[1] "SyntaxExportable: An implementation shall provide a facility to export a model-to-model transformation in the concrete syntax of the language given by the language dimension"[3].

[2] Moment: `http://moment.dsic.upv.es/content/view/34/75/`, last accessed 23/03/2017

[3] The company has been contacted as there was a possibility to get a demo version of the latest release, `http://www.tcs-trddc.com/trddc_website/ModelMorf/ModelMorf.htm`, last accessed 23/03/2017

- The Atlas Transformation Language (ATL) tool was conceived as a QVTr tool [68], however, the ATL evolved in parallel to QVTr and, although they share similar architectures and have a similar operational context [57], ATL does not conform to the QVTr standard.

The Eclipse MMT project[4] provides a QVTo and a QVTd sub–projects. At the beginning of this research the QVTd project provided editors for QVTr and QVTc.

As a result of a growing interest in MDE in the software engineering domain ATL is not the only project that provides support for model transformations that has developed in parallel to QVT.. Other projects like the Epsilon Transformation Language (ETL) (part of the Epsilon framework [80]), Kermeta [40], GReAT [8], VIATRA [103] and MOTOE [61], have entered the model transformation scene. Some of these come from the software engineering domain, although not all fall under the umbrella of the MDA, and others from other disciplines like graph grammars (VIATRA, GReAT) and linear optimisation (MOTOE).

2.3.1 QVT Issues

As with other specifications, the QVT specification has been subject to revisions that aim at fixing issues that have been identified in the available releases. A small number of issues have been reported against the version used as reference for this research (1.2).[5] However, there are some issues against the languages' semantics that are not officially reported. Some of these unreported issues have been outlined by Stevens [96, 97]. Most of the issues identified are relevant to bi-directionality and bijectivity. Further, the difficulty in understanding the semantics of the QVTd languages has played an important role in the lack of available implementa-

[4] Eclipse model to model transformation Project, `http://wiki.eclipse.org/MMT/`

[5] The issues reported against version 1.2 can be found here: `http://issues.omg.org/issues/spec/QVT/1.2`.

tions [96]. From reading the specification as part of this literature review, the existence of typographical errors, complex grammar constructs and ambiguities is evident. The use of the OCL as the language for writing expressions in QVT is a cause of some of the issues [97]. For example, the use of OCL results in the lack of a proper mechanism to test transformations and rule composition [97].

However, Stevens concludes that most of these issues are related to lack of detail in the specification and could be addressed by revising and adding the missing information. The same could be argued for the possible incompatibility between QVTr and QVTc semantics identified by Stevens [96]. This survey was unable to find any work that explored the reasons for the lack of a working implementation of the QVTd languages, the extent of adoption of the set of QVT languages and if there is a relation between the two.

2.3.2 QVT Alternatives

Interestingly,

> it sometimes happens that even those tools which use 'QVT' in their marketing literature do not actually provide any of the three QVT languages, but rather, provide a 'QVT-like' language [96].

Further, from the available tools that claim to support QVTr identified by Stevens [96], few provide support for bi-directional execution and only work in *enforcement* mode.

The landscape of model transformations is growing, with tools that claim some conformance to QVT, that support the QVT languages by compiling them into another language, or that provide bespoke MTLs. Since the list of all the available tools is quite extensive, this section lists some of them in different domains and provide a brief description of the approach they take to model transformations. The tools presented where selected based on the

domain of the implementation, whether the tool implements the QVT specification or another (e.g. bespoke language), and the implementation strategy. This section presents tools that follow particular approaches that could provide interesting ideas that could be adapted in order to support the control component synthesis.

In Greenyer et al. [43] Triple Graph Grammars (TGG) are used to provide execution of QVTr transformations. They use the transformation from QVTr to QVTc (given in the QVT Specification), and a TGG transformation to map QVTc into TGG. Finally, the resultant TGG program is executed by their TGG engine. They support the EMF as the M3 level. The execution in their TGG engine follows an interpreted approach. Further, they also provide a mapping from QVTr to TGG, again using TGG. As part of the mapping of QVT to TGG, the authors also identify aspects of the QVT specification that leave "room for interpretation": rule execution order and bindings of variables across mappings/rules.

GREAT [26] uses programmed graph rewriting to provide support for model transformations. It is independent of QVT and uses the Optimix language to define the patterns between model elements. However, the framework is limited to the transformation of the UML class diagrams.

GReAT [8] (notice the small '*e*') is another transformation language based on graphs. Its authors propose a language similar to the UML class diagrams to define the patterns used for matching. Transformations are then described using a visual language that allows the definition of patterns used in a transformation. Transformations consist of rules and each rule is a block that receives inputs and produces outputs (signals). Rules are cascaded by interconnecting input and output signals. Of interest is how signals are used to interconnect rules, given that in data dependence analysis it is important to identify the flow of data through the program.

Model transformations implemented using a linear programming (LP) approach are discussed by Callow et al. [23]. To be able to use LP, the authors propose a method for representing QVTr

Table 2.1: Comparison of QVT alternatives.

Tool	Domain	Uses QVT	Implementaiton Strategy
Greenyer et al. [43]	Triple Graph Grammars	Yes	QVT tranformed into TGG for execution
GREAT	Graph Rewrite Systems	No	Patterns specified using Optimix language and then executed
GReAT	Graph Rewrite Systems	No	UML like visual language which can be executed
Callow et al. [23]	Linear Programming	Yes (partial)	QVTr is transformed into LP problems
MOTOE	Population based meta-heuristics	No	A transformation pattern is derived form a set of examples. The pattern can then be aplied to a new problem.
Xmorph (Tefkat)	Aspect driven	No	Source and target patterns written in Xmorph which are then executed
Kermeta	GPL (objet oriented)	No	Patterns specified as classes which are then executed.
FunnyQT	Declarative GPL (Clojure)	No	Transformation written in Clojure and then executed
ATL	Model Management Langauges	No	Transformation written in ATL language and then executed.
ETL	Model Management Langauges	No	Transformation written in ETL language and then executed.

transformations as sets. Different sets are used to represent rules and patterns, and are also used to represent the relations in a pattern. The transformation execution itself is a complex process, requiring the execution of two mixed integer linear programs. This work has some shortcomings: some aspects of MOF meta-models and the QVTr language are not captured by the defined sets, inheritance and arbitrary OCL statements are not fully supported, and an automated transformation from QVTr to the aforementioned sets is not provided (i.e. the sets have to be composed manually). The LP approach provides insights into the definition of a cost function for QVTr transformations. Such a function might be of interest in order to compare different synthesized control components in order to determine, for example, if a given synthesis method provides better results.

MOTOE [61] also treats model transformations as an optimization problem, but as with GReAT, its authors provide a bespoke language to define patterns. Patterns are called mapping blocks: "A mapping block gives the translation of a cohesive piece of a source model to its equivalent in the target model" [61]. To execute the transformation they use PSO (a parallel population–based computation technique) originally inspired from the flocking behaviour of birds. Briefly, the algorithm finds the best solution by evaluating how well the elements of the model fit the mapping block. This approach is interesting, given the use of a metaheuristic to find a solution. However, there is no mention of support of bi- or multi-directional transformations and it is not clear what is the modelling technology supported. Of interest is how swarm intelligence methods can be applied to the problem of executing declarative MTLs.

XMorph [36] is a transformation language that was proposed in response to the QVT RFP. However, XMorph is influenced by an aspect-driven approach which makes its semantics unique in the handling of target objects, as creation of target objects is implicit. This allows for several rules to set properties of a single

target object without the need to specify which rule creates it. The transformation is also based on patterns (matched for input elements and used as a template for output ones). In XMorph (like in QVTc) traceability is explicit, with the use of *trackings* to declare relationships between source and target elements.

Tefkat [94] is a model transformation engine that supports XMorph. The engine has two primary components: a source-pattern matcher and a target-pattern resolver. The source-pattern matcher "checks the existence of object instances and the values of their attributes while the target-pattern resolver may create objects or set attribute values to ensure a target pattern holds" [94]. Of interest is the distinction between a source and a target match, which in the QVTd languages could be used to decompose rules into smaller blocks. This decomposition can facilitate the synthesis of the control component by separating access to the candidate models.

Kermeta is the core language of a model-oriented platform [77]. It consists

> of an extension to the Essential Meta-Object Facilities (EMOF) 2.01 to support behavior definition. It provides an action language to specify the body of operations in metamodels. The action language of Kermeta is imperative and object-oriented [76].

Different from other approaches, Kermeta is a fully fledged object oriented programming language; "Kermeta has nothing specific to model transformation, but being quite general purpose, it can be used to implement mechanisms to support model transformations" [76]. To specify a model transformation a new class must be defined for each pattern. The class has attributes to capture the details of the patterns and a transformation operation is defined which receives an input object and returns an output object. The classes are then used by the Kermeta engine to execute the complete transformation.

FunnyQT [51] is a model query and transformation language designed as an extension to Clojure, a Lisp Dialect. FunnyQT is a Clojure API and as a result does not provide a specific syntax for model transformations. Thus, transformation rules are essentially Clojure expressions. For execution, the MTP execution is delegated to the underlying Clojure virtual machine. "Clojure provides a large set of features including higher-order functions and control structures that can be used directly" [51].

The ATL [56] is a domain-specific language for specifying model transformations. The ATL is inspired by the OMG®'s QVT requirements, but "the actual ATL requirements have changed over time as this language matured" [57]. The ATL provides both declarative and imperative constructs and is therefore a hybrid MTL. The ATL is a textual language in which transformations are composed of rules. Each rule defines a pattern by specifying the relation between elements of the input and output metamodel. Transformations are uni-directional. For execution, the transformation is compiled and run in the ATL Virtual Machine. As in QVT, relations are expressed using OCL expressions. In the architecture alignment between ATL and QVT presented in Jouault et al. [57], declarative ATL is considered less abstract than QVTr as it is restricted to unidirectional transformations and hybrid ATL is considered at the same level as QVTr and QVTo ("ATL with imperative features has a lot of overlap with [QVTo] language " [57]).

The Epsilon framework [80] provides a set of languages to work with models, among which there is a transformation language: the ETL [63]. The ETL has a textual syntax similar to the ATL, in which a transformation is composed of rules and each rule specifies a matching pattern between elements of the input and output models. The ETL is unidirectional but supports multiple input and output models. The ETL is a hybrid language and supports annotations to extend and modify how patterns in the rule are matched, to support abstract rules (useful for rule inheritance/extension) and to enhance traceability support.

The landscape of QVTc alternatives presented in this section, although not complete, provides an insight into the multiple approaches to model transformation. The selected alternatives cover languages developed in different domains and with different technologies. The landscape omitted languages for which no implementation details where publicly available. From the surveyed approaches a consensus on the need for pattern matching capabilities emerged. Pattern matching is needed to identify source model elements of interest in the transformation and to define templates that specify how output model elements are created. The challenges of supporting multi-directionality are widespread and most of the referenced languages are limited to uni-directional transformations. Further analysis and testing will be required to identify additional general limitations or de-facto assumptions. One of these assumptions, at least from our experience with the ATL and the ETL, and from the semantic explanation of other approaches, is that each input element is transformed only once by each rule. This may be an implicit requirement to prevent the transformation from entering an infinite loop (i.e. when all input elements matched by input patterns have been transformed the transformation ends).

The aforementioned alternatives are all concerned with model–to–model transformations. Another important area of research is model–to–text transformations. Such transformations are usually performed at the lowest level of abstraction to generate, for example, the code of an application (in a particular programming language) or the SQL statements to initialize a database. Since the QVT standard does not define a language for model-to-text transformations this survey does not include languages that support this type of transformations.

2.4 Existing Infrastructure

Given that there are no available open source tools that support the QVTc language, part of the current efforts of the Eclipse QVTd project aim at providing execution capabilities for the declarative languages. This section discusses some basic aspects required to provide a tool that supports the QVTc language and then shows the current state of the Eclipse QVTd project QVTc transformation tool.

2.4.1 Language Execution

In order to execute a transformation, i.e. create or update the candidate models, the transformation language must be executed on a machine (computer). The process of enabling a language to run on a specific machine is known as an *implementation*. Implementations can be divided into interpretation– and translation–based approaches [111]:

Interpretation–based The main characteristic of the interpreted approach is that the program (transformation) and the data (candidate models) are evaluated simultaneously. "Every construct c of the program is analysed and interpreted as it is encountered during the execution – even if c has already been encountered" [111].

Translation–based In this approach the program and the data are processed at two distinct phases of the execution. In the first phase (compile–time), the program is translated to another representation (e.g. assembler language or bytecode) which can be directly executed by the machine. In the second phase the alternative representation is executed on the machine and data is accessed as required (run–time). Also known as *compiled*, the translation–based approach provides a more efficient execution than interpretation–based approach.

As an alternative to hardware machines, the concept of virtual machines was proposed as early as during the implementation of early ALGOL60 compilers [111]. A virtual machine provides benefits to compilation and interpretation implementations, as it provides instruction sets (machine core instructions) that are tailored to the needs of the language. Thus, providing a virtual machine for a transformation language is desirable as specific instructions for model manipulation can be provided [56, 1].

The QVTc language is specified as an extension to the OCL, which adds new elements to the language such as block interaction mechanisms, semantic constructs and syntactic sugar [93]. Thus, a virtual machine for QVTc can be conceived as an extension to the OCL virtual machine:

> Efficient evaluation of OCL is useful, but increasingly OCL is now used within extended contexts such as QVT or MOFM2T. These languages define an extended the [OCL] Abstract Syntax Tree (AST) and so in principle are amenable to the same tree-walking evaluation as the basic OCL AST. Nonetheless the same basic interpreter can support native evaluation and the code generator can be extended to the extended AST [112].

2.4.2 The Eclipse QVTc VM

The QVTc execution engine available from the Eclipse QVT Declarative project[6] was used for the development of the proposed solution. The main reason for this choice is that one of the conclusions of the field survey (see Sect. 2.3) is that this engine is the only existing available implementation of a QVTc execution engine freely available and actively developed. Since the Eclipse QVT Declarative project was under active development at the time this research was conducted, the development was branched

[6] `https://projects.eclipse.org/projects/modeling.mmt.qvtd`

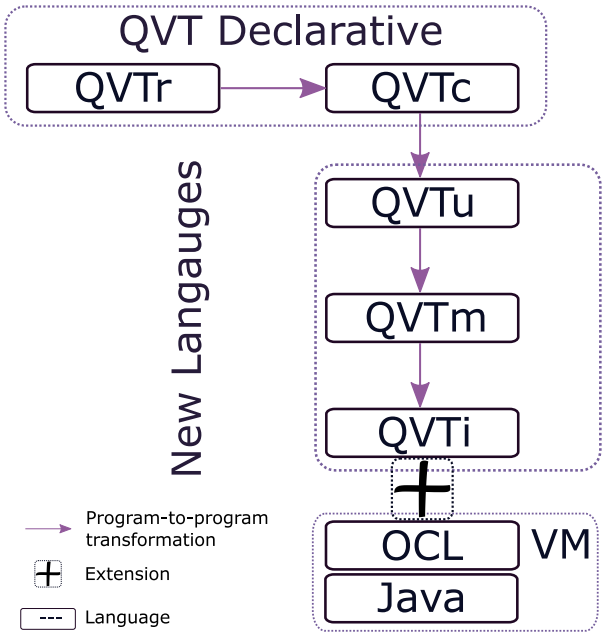


Figure 2.5: Bridging the QVTd to OCL semantic gap.

form the 0.13 RC4 release (7 June 2016). Further, modifications have been made to the QVTi virtual machine, mainly to fix small bugs in the QVTc to QVTm translation and in the interpreted execution, and to change some of the existing behaviour. The reasons for the changes in behaviour where mainly that the QVTd project development was exploring alternative approaches to the synthesis problem. The available engine provides two modes of execution: compiled and interpreted. Given that at the time the interpreted version was more stable and mature, this research uses the interpreted execution mode.

The semantic differences between the QVTd languages and OCL makes the implementation of the QVTc engine a complex task. One way to solve the challenges of the *semantic gap* is to progressively bridge the gap by using intermediate representations and perform the required translations in a series of steps [41].

An initial solution following this approach driven by Edward Willink was published [114] in the International Conference on Model Transformation (ICMT). The solution is based on the intro-

duction of three additional QVTd languages: QVT Unidirectional (QVTu), QVT Minimal (QVTm) and QVT Imperative (QVTi). QVTu is intended to be the closest to QVTc and QVTi the closest to OCL as presented in Fig. 2.5. Execution is provided by a QVTi virtual machine.

QVTu is semantically a subset of QVTc and its intention is to remove bi-directionality (by selecting an execution direction) and explicitly define the execution mode (check or enforce). For the QVTc to QVTu translation to take place, the user must select an execution direction and an execution mode. Bi-directionality is removed by transforming all statements (assignments) that modify the source model into constraints (predicates). If the execution mode is *check*, all modifying statements are transformed into constraints.

QVTm is semantically a subset of QVTu and its intention is to normalize the transformation. Normalization is done by eliminating mapping refinement and nested mappings. Refinement elimination is done by the combination of the refined mapping with the refining mapping as defined by the QVT Specification [3][7]. After refinement elimination all nested mappings are removed by lifting domains in nested mappings to their parent mapping. The lifted domains are merged with their respective domains in the parent mapping.

QVTi supports the same syntax and semantics of QVTm, but adds new imperative language constructs. The additional language constructs include iteration and mapping invocation. The imperative language constructs must be synthesized as part of the translation step.

The QVTi virtual machine is responsible for three main activities:

- Source model traversal to query all elements of a given type (*allInstances*).

[7]Section 9.14 Mapping Refinement

- Mapping invocation for a set of variables bound to model elements.
- Target model modification via new element creation and element modification (absent in OCL).

With the translation between the new languages in place and an initial implementation of the QVTi virtual machine, this research contributes to the Eclipse QVTd Project effort by proposing the mechanism for synthesizing the imperative language constructs.

2.5 Control Component Synthesis

In this section we present how data dependence provides a synthesis strategy for control component synthesis in the case of instruction scheduling. This approach is the base of the solution proposed in this project. We will consider the transformation from QVTm to QVTi as a compilation step. This compilation is a pre–processing activity required to generate the imperative (QVTi) MTP that is executed by the QVTi virtual machine. The QVTi MTP can be then executed either by interpretation or compilation.

A MTP is a description of an algorithm that specifies how to transform a source model (or models) into a target model (or models). A *program* is an algorithm representation that is meant to be readable by a machine. Algorithms can be regarded as consisting of a logic (L) and a control (C) component [65]:

$$A = (C, L) \tag{2.1}$$

The logic component describes the knowledge about the problem being solved by the algorithm. The control component determines the way in which the knowledge can be used to solve the problem. If C does not affect the meaning of the algorithm, then "different algorithms A_1 and A_2, obtained by applying different methods of control C_1 and C_2 to the same logic definitions L, are equivalent

in the sense that they solve the same problems with the same results" [65].

Compilation is described by Wilhelm et al. as a process in which a program p_K, which is written in a language K, is translated into a program p_M, which is written in a machine (or assembly) language M, of the real or virtual machine that will execute the program [111]. Compilation can be then understood as a process in which the algorithm $A_{p_K} = (C_{p_K}, L_p)$ is transformed into the algorithm $A_{p_M} = (C_{p_M}, L_p)$. Since both algorithms share the same logic, they must produce the same results. Program p_K, has a structure defined by language K, and its *semantics* (the meaning of the program) is described by this structure. The job of the compiler is to collect the semantics of a program in a *semantic representation* and rephrase it in terms of the target language [44].

An MTP p_m written in QVTm can be represented, using (2.1), as $A_{p_m} = (C^*_{QVT_c}, L_p)$. Strictly speaking, given that the QVTm language is declarative, the algorithm does not include a control component. However, there exists an implicit control component C^* given by the semantics of the language. The compilation of p_m to a QVTi version p_i, must then produce $A_{p_i} = (C_{QVT_i}, L_p)$, with $A_{p_i} \equiv A_{p_m}$. The resulting algorithm A_{p_i} uses the imperative constructs of the QVTi language to produce the same results.

The next section describes how the semantic representation of an algorithm can be constructed. We will first go over some basic compiler theory to introduce the concepts of instruction scheduling and the role of data dependence analysis as part of compiler optimizations.

2.5.1 The Abstract Syntax Tree (AST)

The first step in a compilation is to parse the text of the program. The result of the parse stage is usually a syntax tree that represents the program in terms of the grammar (or concrete syntax) of the language. Given that syntax tree is closer to the grammar, an additional step is taken to shift the emphasis from the syntax to

the semantic content of the program [44]. The result of this step is the AST. This section introduces a simple arithmetic expressions language that is used to explain the construction of the AST, how the AST is used to schedule the execution of a program written in this language, and how data dependence analysis is used for scheduling optimizations.

Listing 2.1: Simple arithmetic expression grammar.

```
1 expression → expression '+' term | expression '−' term | term
2 term → term '∗' factor | term '/' factor | factor
3 factor → identifier | constant | '(' expression ')'
```

The grammar for the simple arithmetic expressions language (K) is presented in Listing 2.1, written in a simplified BNF syntax. Each line is a derivation rule, where the name to the left of the arrow is a *symbol*. To the right of the arrow the sequence of symbols determines a possible substitution for the symbol to the left. A "|" is used to indicate different choices for the replacement.

In language K, an `expression` can be replaced by a `term`, or by an `expression` and a `term` separated by either a + or − character. This allows the construction of addition and subtraction expressions. Similarly, the derivation rule of a `term` allows the construction of multiply and divide expressions. Finally, the derivation rule of a `factor` allows using identifiers (names) or constants (numbers) to define the factors (operands) of the expression. Such grammar allows the construction of expressions such as `b+2*(3/a)-c` or `b*b-4*a*c`. For the rest of this exposition $p_K = b * b4 * a * c$.

A parser builds a parse tree by determining the best way to replace parts of the expressions with the grammar derivations. The parse tree of p_K is presented in Fig. 2.6 and the equivalent AST presented in Fig. 2.7. In the AST the information related to the grammar is removed, leaving only the information relevant to the syntactic structure. Notice that operator precedence defined in the grammar is highlighted by the structure. Further, since each operator has a clear semantic meaning, the expression can be executed (a result calculated).

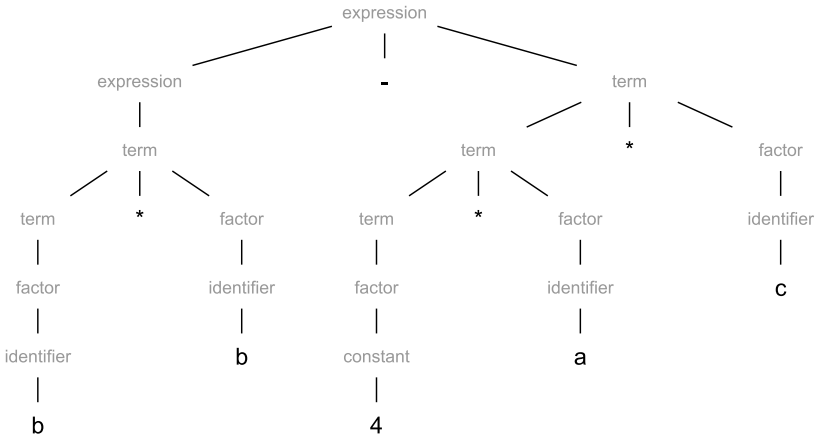


Figure 2.6: Parse tree for expression `b*b-4*a*c`.

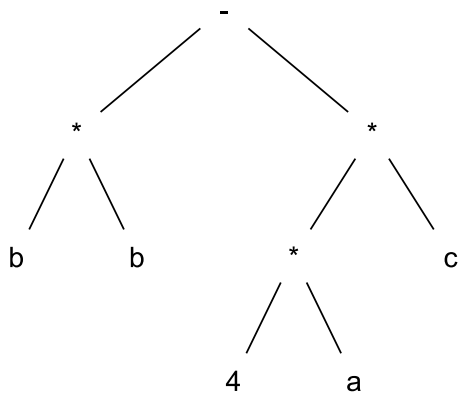


Figure 2.7: AST for expression `b*b-4*a*c`.

2.5.2 Instruction Scheduling

The AST is a good representation of the semantic meaning of an algorithm. However, the compiler still needs to do an additional step to produce the program p_M that can run in the target machine. This additional step maps semantic constructs of the source language K to specific executable instructions in the target machine and generates the target machine program p_M.

Lets consider the case in which the target machine is a *pure register machine*. A pure register machine can store values in a set

Table 2.2: A simple register machine instruction set.

Instruction		Action
load_C	c, R_n	$R_n := c$
load_M	x, R_n	$R_n := x$
store_R	R_n, x	$x := R_n$
add_R	R_m, R_n	$R_n := R_n + R_m$
sub_R	R_m, R_n	$R_n := R_n - R_m$
mul_R	R_m, R_n	$R_n := R_n \times R_m$
div_R	R_m, R_n	$R_n := R_n \div R_m$

of registers and provides a basic set of instructions that can be performed on the registers. Table 2.2 describes a simple register machine that will be used for the remaining of this exposition. The first set of instructions describe operations between memory (or a constant) and a register: load the register with a value and store the value in a register in memory. The instructions in the second set perform an operation between two registers, leaving the result in the second one.

Since the register operations map nicely to the semantics of language K, the machine–ready AST can be constructed by replacing each of the nodes in the AST by its corresponding machine instruction. Assuming the target machine has an infinite amount of registers, the translation is straightforward. A possible machine–ready AST is presented in Fig. 2.8.

The second part of the step for generating p_M is responsible for instruction scheduling. For this, note that each node can only be computed after the results of its children have been computed. For example `mul_R` $\mathtt{R}_2, \mathtt{R}_1$ cannot be computed until `load_M` $\mathtt{b}, \mathtt{R}_2$ and `load_M` $\mathtt{b}, \mathtt{R}_1$ have been computed. This suggests that the instruction scheduling can be obtained by a depth-first traversal of the machine–ready AST. In depth-first traversal, the child of a node is explored before exploring its siblings. The resulting p_M is presented in Listing 2.2.

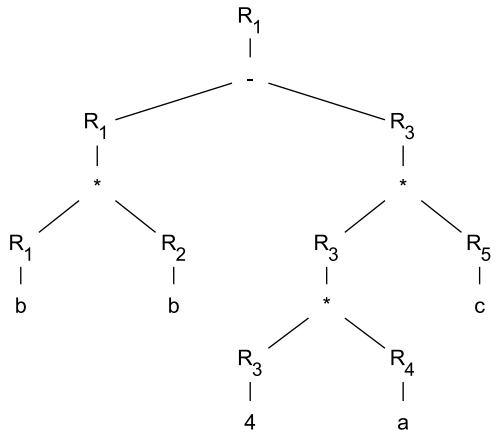


Figure 2.8: AST for expression `b*b-4*a*c` using the machine registers for storage.

Listing 2.2: Register machine code for the expression `b*b-4*a*c`

```
load_M  b, R1
load_M  b, R2
mul_R   R2, R1
load_C  4, R3
load_M  a, R4
mul_R   R4, R3
load_M  c, R5
mul_R   R5, R3
sub_R   R3, R1
```

2.5.3 Optimization and Data Dependence Analysis

The assumption of infinite registers is unsound, given that a hardware or virtual machine is bound to have a limit to its memory size, and as a result, a limit to the number of registers it provides (or alternative storage mechanism for computation, e.g. a stack). In general, it is preferable that a program a) uses as few registers (memory) as possible, and b) executes as fast as possible. Most modern compilers will perform some sort of optimization during generation of the machine program to reduce its memory use and boost its performance (e.g. reduce execution time).

Careful analysis of the register machine code program p_M presented in Listing 2.2 reveals that it is possible to use only three

registers instead of five and that one instruction can be eliminated. The result of making these optimizations is presented in Listing 2.3. Using fewer registers was possible by a simple optimization that generates code for the child that requires the most registers first. Elimination of the double loading of `b` is done with a more advanced optimization technique called *common subexpression elimination*.

Listing 2.3: Optimized register machine code for the expression `b*b-4*a*c`

```
load_C  4, R2
load_M  a, R3
mul_R   R3, R2
load_M  c, R2
mul_R   R2, R3
load_M  b, R1
mul_R   R1, R1
sub_R   R3, R1
```

Instruction selection, register allocation, common subexpression elimination, and other optimization techniques, are key to finding an optimal machine program. However, these three actions are intertwined and finding an optimal machine program is an $\mathcal{NP}$-complete problem [44, 9]. $\mathcal{NP}$-complete problems are, in essence, problems for which a solution cannot be found in polynomial time, that is, the time to solve the problem grows exponentially as the size of the problem increases. From the different optimization techniques, instruction scheduling optimization is of interest because of its role in increasing processor utilization [75, 10], which results in reduced execution times. This research investigates how instruction scheduling techniques can be adopted in order to synthesize the control component for programs written in QVTc.

Instruction scheduling is usually performed on atomic blocks of code, or *basic blocks*. A basic block is defined as a section of code (one or more instructions) that has only one entry point (the first line of code) and in which control will leave the block without halting or branching [10]. The instruction scheduling problem is then to find a schedule for the instructions within the block so

it executes in the minimum number of clock cycles. In general this problem is $\mathcal{NP}$-hard [13, 48]. For $\mathcal{NP}$-hard problems, as with $\mathcal{NP}$-complete problems, a solution cannot be found in polynomial time. Additionally, once a solution is constructed for a $\mathcal{NP}$-hard problem, it may not be possible to verify it in polynomial time.

Different intermediate representations of the AST proposed to solve this problem are: control flow graphs, def-use chains, data dependence graphs, program dependence graphs [39], static single assignment form [18], among others [82]. Although an ideal program representation should include information about the flow of control and the flow of data [82], some of these representations only contain information about one of the two. This is the case for declarative languages in which the control component is absent.

The alternate representation is constructed by analysing the AST to identify control and data dependencies between instructions. The analysis can also be done on the machine-ready AST [48]. The data dependence analysis is usually performed on the AST because this representation makes data dependencies easier to identify between constructs of the user programming language.

Coming back to our example arithmetic expression language, this language can be considered to be declarative. The program (equation) only describes the logic: the calculation the programmer wishes to compute. There are no control constructs to indicate how the calculation must be computed. If the semantics of the language are respected, the compiler is free to execute the subexpressions in any order. For example, additional to the two cases presented earlier, all register loading could be done first as in Listing 2.4. Note that in this last example `mul_R` $\mathtt{R}_4, \mathtt{R}_3$ is evaluated first. These three programs will produce the same result.

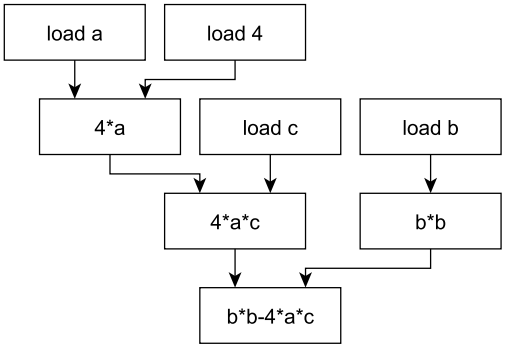


Figure 2.9: DDG for expression `b*b-4*a*c`.

Listing 2.4: Alternative register machine code for the expression `b*b-4*a*c`

```
load_C  4, R3
load_M  c, R5
load_M  b, R1
load_M  b, R2
load_M  a, R4
mul_R   R4, R3
mul_R   R2, R1
mul_R   R5, R3
sub_R   R3, R1
```

A plausible data dependence graph (DDG) analysis for our example expression `b*b-4*a*c` is presented in Fig. 2.9. This graph reveals important information:

- There are no dependencies between the operations that load the variables and constants. Hence, as in Listing 2.4, the load operations can be invoked in any order with respect to each other.

- The rest of the dependencies are due to the semantics of the language: an operation needs its operands to be available before it can be executed. Hence, `a*4` depends on a and 4 to be loaded and `a*4*c` depends on `a*4` being evaluated and c being loaded.

Instruction scheduling is built around the concept of predecessors and successors in the DDG:

> In a [data dependence graph], the *predecessors* of a node i are the set of nodes j excluding i for which there exists a directed path from j to i. Conversely, the *successors* of i are the set of nodes j excluding i for which there exists a directed path from i to j [47].

The instruction represented by a node i cannot be scheduled until all its *predecessors* have been scheduled. All *successors* of the instruction represented by a node i must appear after node i in the schedule. Note that when used in conjunction with other optimization techniques additional considerations need to be taken. For example, *static single assignment form* optimization can result in some instructions being discarded, in which case the predecessor/successor scheduling constraints must be refined. Other approaches to scheduling using data dependencies can be found in Wolfe et al. [116], Halbwachs et al. [45], Sweany et al. [100], Battacharyya et al. [12], and Ferrante et al. [39].

2.5.4 Existing Solutions

The theories from compiler optimization (including instruction scheduling) have not been reused in a substantive way in the development of model transformation languages [89]. This is supported by the very few articles that were found during this literature review that detail how the control component is synthesized for the existing declarative MTLs.

In the case of the ETL [63] and the ATL [58] languages the control component is synthesized as proposed by Jouault et al. [58]: Rules are scheduled in the order they are defined in the MTP, with the complete set of rules executed twice. In the first iteration each rule is analysed, and new elements created according to the target (output) elements of the rule. In the second iteration the logic of each rule is executed. No further analysis is done, for example, to execute the rules in a different (optimal) order. In the case of FunnyQT [51], the author mentions that there is no compilation of

the users MTP before delegating execution of the program to the Clojure execution engine[8]. The Clojure interpreter may do some optimizations, but no specific information about the transformation domains is available to the Clojure interpreter so these cannot be considered optimizations of the transformation.

Although functional languages can be considered a subclass of declarative languages, an important difference is that since there exists an explicit relation between functions (i.e. one function is computed by invoking a set of other functions), there is an initial implicit ordering. This is similar to the relations defined by operators in the arithmetic expression language introduced previously, that is, an operator can't be evaluated by itself: its operands must be evaluated first. Thus, optimization of functional languages focuses on finding the correct set of function invocations to obtain a result as opposed to finding an order in which to invoke these functions.

There exist declarative languages used in other domains. Examples of pure declarative languages include the Structured Query Language (SQL), Prolog and Triple Graph Grammars (TGG). The SQL has become the de facto language for data insert, query, update and delete in relational data base systems. Prolog is a general purpose programming language with applications in symbolic computation such as natural language processing and artificial intelligence. TGGs are a techinique for defining the correspondence between two graphs and to translate one graph to the other (bi–directional) [88].

The SQL compiler is responsible for query optimization. This process includes determining the order in which queries should be invoked (query scheduling) and finding an optimal path for computing *join* operations when the query spans multiple tables. From the techniques applied to optimization of SQL queries the method for selecting the optimal path for *join* computations are of interest to this research. Optimal path selection involves evaluating

[8]Personal communication, May 2016

the cost of each path to find the path with the least cost. The cost is a function of the required I/O access (e.g. disk access) and CPU utilization (instructions executed) [90, 55]. In SQL optimizations the complete solution space of possible paths is explored, and the best is selected.

Section 2.5.3 presented how multiple alternative schedules can be constructed for the same program written in a declarative language. Given that all versions produce the same (correct) result, by assigning a cost to each it can then be used to pick the *best* solution. The cost function should measure aspects relevant to the particular application domain and according to which of these aspects are to be optimized.

A Prolog program consists of a set of clauses, where each clause is either a fact or a rule about the problem to be solved. In Prolog, optimizations such as intelligent backtracking [67] use data dependence analysis to tag previous clauses and provide smarter backtracking choices. The backtracking mechanism aims at reducing the number of solutions constructed for evaluation. Instead of construction all possible solutions, the interpreter keeps a partial order of clauses that satisfy the solution. If adding a new clause to the solution does not provide a correct result, the compiler/interpret can backtrack to the last best known partial solution and start from there. Although this type of optimization is done at runtime and this research focuses on providing a compile time solution, the idea of constructing a partial solution and reduce the number of explored alternatives is interesting but was not adopted because the semantics of QVTc are very difficult to validate in partial solutions.

Triple Graph Grammars (TTG) are of particular interest as they can also be used for model transformations. Data dependence analysis can be used for deriving test cases for complete coverage testing in TGG. However, the data dependencies are per pattern (block), as opposed to individual variable instructions [50]. This approach is interesting because many instructions are grouped

together, reducing the number of nodes in the data dependence graph. Further, since in this research the interest is in the execution order of the QVTc mappings, not the individual statements inside the rule, considering each the complete MTP a basic block and each mapping an instruction, seems a very plausible approach.

In the domain of graph transformations, Lauder et al. [70] present an optimization to TGG execution called Precedence TGGs. The authors identify data dependencies between rules and use this information to topologically sort the nodes in a source graph (model) as opposed to defining an schedule for the rules. This is due to the fact that TGG execution is defined by source graph traversal as opposed to rule traversal. Further, there is no mention of the different execution alternatives and if the proposed algorithm results in an optimal (sufficiently good execution) for a given TGG program.

Although not directly related to declarative languages, the problem of pattern matching has some similarities to that of instruction scheduling. In pattern matching, it is desirable to find an optimal solution in which the constraints defined in a pattern should be satisfied. A pattern consists of a set of variables, and a set of constraints that place restrictions between the variables. The size of the variable set is referred to as the arity. Varró et al. [104] use metaheuristics to provide a solution to the problem of pattern matching. Their approach proposes the use of a search plan to determine the best way to match variables in the pattern. The best search plan optimizes the order in which the constraints are satisfied, by finding an optimal order in which elements of the types involved in the pattern should be retrieved from the model(s). Thus, their approach is also in the lines of TGG optimizations in which the solution is a schedule on the model queries rather than on the grammar/pattern statements.

2.6 Ant Colony Metaheuristic

Recall that the instruction scheduling problem in compiler optimization is $\mathcal{NP}$-hard. Given the inability to find an algorithm that can solve these types of problems in polynomial time, an accepted approach is to use an algorithm that can find a *good enough solution* in polynomial time. In this case of instruction scheduling, a *good enough solution* is a schedule that results in a correct (produces expected result) execution of the problem. Heuristics and approximation algorithms are two methods that can be used for finding a such a solution.

The main differences between the two approaches is that approximation algorithms can provide solutions that are within a guaranteed factor of the optimal solution and that the run time bounds can be defined [105]. On the other hand, heuristic algorithms can only claim that the solution is good-enough and that it can be found fast, where *good-enough* and *fast* are loosely defined. This research focuses on using a heuristic approach, as our objective is to demonstrate that the control component of a QVTc transformation can be synthesized systematically, as opposed to proving that a solution is within a guaranteed factor of the optimal.

2.6.1 Metaheuristics

Metaheuristics are high-level problem-independent approaches for solving optimization problems. A metaheuristic is an algorithmic framework that provides a method to find a sufficiently good solution to an optimization problem. Metaheuristics tackle the problem of finding a near-optimal solution to a problem by efficiently exploring the search space [20] (the set of all possible solutions to the problem). One of the key features of metaheuristics is that a particular metaheuristic can be applied to a wide range of problems. This is possible thanks to their ability to be tailored to a specific domain with the use of heuristics, that is, the ranking

method to differentiate better from worse solutions is configurable. An important aspect of working with metaheuristics is selecting one that fits the problem to be solved. The key aspects to consider are the general strategy of the algorithm, the spread of the search and the problem representation.

The general strategy of the algorithm can be classified into *constructive* and *local search* methods [20]. Local search algorithms usually provide better quality solutions than constructive ones (closer to the optimal value), but to do so they require the existence of an initial solution. The initial solution is then iteratively replaced by a better one, selected from solutions that are close (in the search space) to the latest best–found solution. Constructive algorithms can construct a solution from scratch. The solution is built by adding components until a solution is complete. The result of adding a component can be evaluated to determine if adding the component will result in a better or worse solution. For the QVTc language the control component must be constructed from the ground up and hence this research focuses on constructive algorithms.

The spread of the search is related to the balance between *diversification* and *intensification* [20]. Diversification accounts for how much of the search space is explored, while intensification is related to the use (or not) of the search experience at later search stages. More specifically, intensification locks an algorithm on examining solutions that are close to the best solution(s) found so far. Diversification on the other hand, allows the algorithm to break from this lock and explore different areas of the search space. Ideally, metaheuristics must have an optimal balance between diversification and intensification [20].

Finally, the problem representation relates to the required model, if any, that must be used to represent a problem in order to use a particular algorithm. In *trajectory methods* (one grouping classification for metaheuristics) the problem does not need a special representation. The only requirement is that the quality of

each solution can be measured. In *population-based methods* (the other grouping classification for metaheuristics) the problem representation is important as the search algorithm must manipulate the current best solution by applying mutation operators, or must be able to construct and evaluate partial solutions. Given that the state of the art on instruction scheduling optimizations is done with the use of graph representations (see Sect. 2.5.3), the use of population-based methods seems more appealing. A graph structure is easily encodable in order to apply mutation operators [69, 52] and is also amenable to partial construction and evaluation.

2.6.2 $\mathcal{MAX} - \mathcal{MIN}$ Ant System

The Ant Colony Optimization (ACO) [33] is a cooperative heuristics searching algorithm in which the agents replicate the behaviour of ants, originally introduced by Dorigo et al. [34]. In nature, when exploring a foraging area for food, individual ants deposit a pheromone on the paths they wander. Foragers then use the pheromone information to make decisions on what direction to take, preferring paths in which the pheromone is stronger. This method of communication is known as *stigmergy*. Foragers will reinforce the paths they traverse with their own pheromone. Since pheromones decay over time, given two paths visited by the same number of ants the shorter path will eventually be preferred as its pheromone strength will be higher (ants take longer time to walk the longer path and hence pheromone decays more). As a result, although any single ant moves randomly, a collective knowledge of the shortest paths to the food sources emerges.

The ACO is a *constructive*, *population-based* metaheuristic. To be precise, it is a *swarm intelligence* metaheuristic, a sub-group of population-based metaheuristics, that replicate the behaviour of a particular animal as found in nature. Other examples of *swarm intelligence* metaheuristic are Particle Swarm optimization [37] and the Artificial Bee colony [59], among others. The ACO draws from the behaviour of ants, where a group of cooperating agents ex-

plore the search space to find a good enough solution. The details of the problem domain can be used to model the foraging area (the foraging area can be an alternate representation of the problem) or to place further restrictions on the paths the ants can follow during exploration. The ACO has been used to solve a number of $\mathcal{NP}$–hard problems, such as the Travelling Salesman Problem (TSP) [32], vehicle routing problem [46], process planning problem [110], minimum spanning tree problem (MST) [78], minimum–weight rooted spanning arborescence (MWRSA) problem [19, 71] and instruction scheduling problem for imperative languages [109].

The ACO works as follows. Initially the ants are placed on a random component (part) of the solution. Each ant builds a solution by applying a *state transition rule* to pick the next component that is added to the solution. The next component is selected using a probabilistic choice which is biased by the pheromone information τ_{ij} and by local heuristic information η_{ij}. The local heuristic information is usually a measure of the cost (effort) of selecting component j after selecting component i. For example, in the traveling salesman problem[9] (TSP), η_{ij} is a function of the distance between cities. Ants will prefer components with a high pheromone value and low heuristic value, choosing the next component to add to the solution with probability:

$$p_{ij} = \begin{cases} \dfrac{[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}{\sum\limits_{l \in \mathcal{N}_i}[\tau_{il}]^{\alpha}[\eta_{il}]^{\beta}}, & \text{if } j \in \mathcal{N}_i \\ 0, \text{ otherwise} \end{cases} , \tag{2.2}$$

where α and $\beta (\alpha, \beta \in \mathbb{Z}^{+})$ are two configuration parameters to assign weights to the relative importance of the pheromone and the heuristic information, and $\mathcal{N}_i$ is the feasible neighbourhood, that is, the components that the ant can add next to the solution. Selection of the feasible neighbourhood is critical to guarantee that the ants produce correct solutions. For example, in the TSP

[9]The traveling salesman problem is a problem requiring to find the tour that visits n interconnected cities in the least total distance.

problem, $\mathcal{N}_i$ is given by the cities the ant hasn't visited yet. If the ant could to visit a city twice, the solution would not be correct.

During construction, the ant deposits pheromones on the selected component, and then selects the next one. The ant continues adding components until a complete solution is built. Once all ants have terminated constructing their solution, the pheromone on the components is updated using a *global pheromone updating rule*. The updating rule is of the form:

$$\tau_{ij} = \rho\tau_{ij} + \sum_{s \in S^* | c_i^j \in s} g(s), \tag{2.3}$$

where $\rho \in (0, 1]$ is the evaporation rate, c_i^j is the component associated with the pheromone, S^* is a set of good solutions and $g(\cdot) : S \longmapsto \mathbb{R}^+$ is a function that determines the quality of a solution. This rule determines the rate of decay of the pheromone.

> ACO algorithms typically differ in the way pheromone update is implemented: different specifications of how to determine $S*$ result in different instantiations of update rule [(2.3)]. Typically, $S*$ is a subset of $S_{iter} \cup \{s_{gb}\}$, where S_{iter} is the set of all solutions constructed in the current iteration of the main loop and s_{gb} is the best solution found since the start of the algorithm (gb stands for global-best) [35]

Mimicking ants found in nature, components that are picked by a larger number of ants will have higher pheromone values and thus will be more likely to be part of a good solution.

The first ant colony metaheuristic, known as the Ant System [34], has been studied in depth [32, 35] and improved versions have been proposed, in particular to optimize the balance between *diversification* and *intensification*.

The Ant Colony System (ACS) [32] focused on increasing the importance of exploitation by introducing a *pseudo-random proportional rule* that alternates between exploration and exploitation: with probability $q_0, 0 \leq q_0 < 1$ the ant will move to com-

ponent j for which the pheromone and heuristic information is maximum. The *pseudo-random proportional rule* is of the form:

$$j = \begin{cases} \arg \max_{c_i^j \in \mathcal{N}_{s_p}} \{[\tau_{ij} \times [\eta_{ij}]^\beta\}, & \text{if} \quad q \leq q_0 \\ S, & \text{otherwise} \end{cases} \tag{2.4}$$

where S is a random selection according to the probability given by (2.2), and $\mathcal{N}_{s_p}$ is the set of solution components that may be added while maintaining feasibility.

The $\mathcal{MAX} - \mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) introduced by Stützle et al., improves on the ACS and optimizes the balance between *diversification* and *intensification* by using a stronger exploitation of the best solutions found during the search and incorporating a mechanisms to avoid search stagnation [98]. Search stagnation happens when the algorithm focuses on intensification, and as a result, solutions are only searched in the search space close to the best solution found.

In the $\mathcal{MM}$AS algorithm, only one ant is allowed to deposit pheromones. This ant "may be the one which found the best solution in the current iteration (iteration–best ant) or the one which found the best solution from the beginning of the trial (global–best ant)." [98]. This ant is either the ant that found the best solution in the current iteration (s^{ib}), or the ant that has found the best solution from the beginning (s^{gb}).

"The use of only one solution, either s^{ib} or s^{gb}, for the pheromone update is the most important means of search exploitation" [98]. This approach is referred to as the *best ant feedback strategy*. Using only s^{gb} it is possible that results concentrate too fast around this solution, probably resulting in low quality solutions. By allowing s^{ib} to be used and considering that the iteration–best solution may differ considerably between iterations, a larger area of the solution space can be explored. In fact, Stützle et al. [98] show that a dynamical mixed best ant feedback strategy where s^{gb} is used more frequently as the search progress produces the best results. Their proposed approach is as follows: use s^{ib} to update

the pheromone information but at given intervals use s^{gb}. The value of the interval is gradually decreased (i.e. use s^{gb} more frequently) to veer the search from an early exploratory phase to a late exploitation of the overall best solution phase.

Second, to avoid stagnation the amount of pheromone is limited to an interval $[\tau_{min}, \tau_{max}]$. By "limiting the influence of the pheromone trails one can easily avoid the relative differences between the pheromone trails from becoming too extreme" [98]. This has the effect that there are no components with significantly higher pheromone values than the others. In ACS algorithms, stagnation occurs when at each choice point, the pheromone value in one of the trails is significantly higher than for the rest. When stagnation occurs, the probabilistic choice from (2.2) will lead the ants to prefer the invocation with the highest pheromone over all the others. When (2.4) is applied after each iteration, further reinforcement will be given to this invocation. Eventually, the ants will construct the same solution over and over again and the exploration of the solution space stops. Stützle and Hoos [98] proved that for $\mathcal{MM}$AS to converge the values of $\tau_{\min}$ and $\tau_{\max}$ depend on s^{gb} and therefore must be updated after each iteration. As a result, for iteration t, the upper and lower bounds [98] are given by (2.5) and (2.6)

$$\tau_{\max}(t) = \frac{1}{1-\rho}\frac{1}{f(Q^{best}(t-1))} \tag{2.5}$$

$$\tau_{\min}(t) = \frac{\tau_{\max}(1-p_{\text{dec}})}{(avg-1)p_{\text{dec}}} \tag{2.6}$$

where p_{dec} is given by

$$p_{\text{dec}} = \sqrt[n]{p_{best}} \tag{2.7}$$

where n is the number of components and $avg = n/2$. Without pheromone limits, as a result of the *state transition rule* components with significantly high pheromone values will always be preferred by the ants, resulting in stagnation.

2.6.3 Instruction Scheduling using the $\mathcal{MAX} - \mathcal{MIN}$ Ant System

Wang et al. [109] showed that the $\mathcal{MAX} - \mathcal{MIN}$ Ant System is able to generate consistently good scheduling results when compared to other heuristic methods [109]. Further, instruction scheduling using the $\mathcal{MAX} - \mathcal{MIN}$ Ant System can outperform more popular scheduling methods and scales well over different applications and problem sizes [108]. In these approaches the data flow graph (DFG) is used as the foraging area of the ant colony [109, 108]. That is, the ants build a solution by visiting the nodes in the DFG. Each node in the DFG is an operation and edges represent dependencies between operations. Given that a DFG only represents the flow of data, this representation matches the requirements of our problem, given that a QVTc program has no explicit control information.

However, the main goal of our approach is to find solutions that result in correct executions while the goal in [109] is to minimize the execution time of the program. Another important difference is that they use a set of well-known heuristics used for instruction scheduling. These heuristics are constructed with the goal of reducing the execution time. For example, the *instruction mobility* heuristic gives a lower value to instructions that are more urgent and must be executed soon; the *successor number* heuristic gives a lower value to nodes with more successors as scheduling such nodes has a higher probability of satisfying the greater number of dependencies [109]. This research formulates a heuristic that will allow the ant to pick mappings in an order that will result in a correct execution. As in the instruction scheduling problem, these heuristics must evaluate characteristics of the solution that are particular to the model transformation domain. Section 5.3 presents a detailed description on the use of the $\mathcal{MAX} - \mathcal{MIN}$ Ant System to find a solution to the QVTc control component synthesis problem.

2.7 Summary

This chapter introduced the basic concepts of Model Driven Engineering (MDE), the QVT Core [3] (QVTc) language as proposed by the Object Management Group® (OMG®) and the existing implementations of the language. The chapter followed with an overview of the instruction scheduling problem and the use of data dependence analysis as a means to solve it. Finally, the chapter discussed how metaheuristics are applied to solve $\mathcal{NP}$-hard problems, such as instruction scheduling (for imperative languages), and introduced the $\mathcal{MAX} - \mathcal{MIN}$ Ant System, a state-of-the art approach that draws inspiration from the behaviour of ants to solve $\mathcal{NP}$-hard problems. Although the $\mathcal{MAX} - \mathcal{MIN}$ Ant System has been successfully used for instruction scheduling in other domains, its application to a new domain requires a precise formulation of the problem representation, a careful selection of the local heuristics and a correct construction of the feasible neighbourhood. The remainder of these thesis presents a methodology to perform data dependence analysis on QVTc programs, proposes a problem representation based on data dependence analysis and the semantics of the QVTc language, and presents how the synthesis of QVTc control components can be achieved using the Ant Colony System (ACS) metaheuristic.

Part II

Systematic Synthesis

3

The Control Component Model

The previous chapter characterized a Model Transformation Program (MTP) as an algorithm and discussed that algorithms are composed of a logic and control component as defined by (2.1). In the case of a declarative language it was discussed that the control component must be synthesized before execution. This chapter looks at the problem of understanding the logic component in a MTP written in QVT Core [3] (QVTc) (referred to onwards as a *QVTc transformation*). Understanding of the logic component is a fundamental aspect of control component synthesis as it allows us to determine the purpose of the logic and the knowledge embedded in it.

This chapter gives an overview of the syntax and semantics of QVTc to enable readers who are not familiar with the language to follow the discussion on reasoning about the purpose of the logic component. Then, the chapter presents an analysis of the purpose of the logic from an execution point of view, which is used to define QVTc transformation execution correctness. This definition is crucial in the synthesis process as it constitutes the main condition for validating synthesized control components. Finally, analysis of the required execution facilities of a control component is used to

define a model for representing it. This model facilitates abstract interpretation and use of algorithms for analysis and synthesis of the control component. Understanding the knowledge embedded in the logic component is presented in the next chapter.

Section 3.1 provides an overview of the syntax and semantics of QVTc and Sect. 3.2 introduces a running example used in the remainder of the thesis. The objective of Sect. 3.3 is to analyse the purpose of the logic component by relating the language semantics to the runtime execution of a transformation and to provide a definition for a *mapping invocation* during execution; this definition is used to formally define transformation execution correctness for QVTc transformations. In order to facilitate analysis and use of algorithms towards the definition of the synthesis process, Sect. 3.4 explores the basic construction blocks required to provide a model for representing the control component and then continues to present this model. Finally, Sect. 3.5 summarizes and concludes the discussion.

3.1 QVTc Syntax and Semantics Overview

This section presents an overview of the syntax and semantics of the QVTc language (for a more detailed discussion please refer to the Meta Object Facility (MOF) Query/View/Transformation (QVT) Specification [3]). Note that in practice the data dependency analysis and instruction scheduling are done by analysis if the QVTm translation of the QVTc transformation. However, the relevant language constructs and semantics are shared by the two languages and thus the discussion is kept at the QVTc level.

In QVTc a transformation is defined as "a set of mappings that declare constraints that must hold between the model elements of the candidate models and the trace model" [3]. First, this section looks at how the constraints are declared (the syntax), and later at how the constraints are checked and enforced (the semantics).

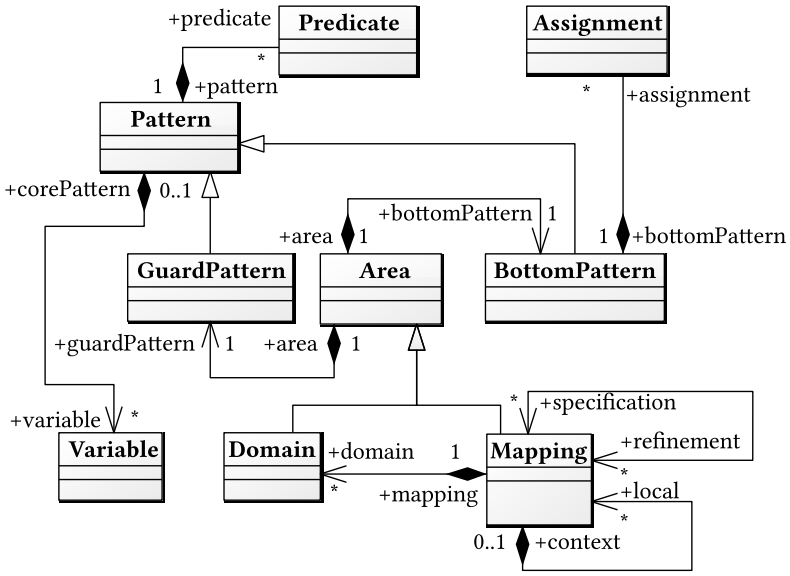


Figure 3.1: Simplified QVTc metamodel

3.1.1 QVTc Syntax

A simplified diagram of the QVTc abstract syntax is presented in Fig. 3.1. In QVTc rules are called mappings (these terms are used interchangeably). A `Mapping` can have multiple domains. Each `Domain` represents a candidate model. Domains can be checkable, enforceable or both (or none). Mappings and domains have a `GuardPattern` and a `BottomPattern`. Patterns contain variables. A `Variable` in a domain pattern must have a type from the candidate model represented by the domain. Variables in the mapping's patterns must have a type from the trace model. A realized variable is a variable that creates a new instance of the type of the variable in the respective model (candidate or trace).

Patterns have predicates. A `Predicate` is a boolean–valued Object Constraint Language (OCL) expression. Predicates are used to define relations between variables or conditions on the values of the properties (attributes/references) of a variable. A `BottomPattern` has assignments. An `Assignment` sets the value of a property of an element or of a variable. For the former, the element can be specified using an OCL expression, and in both

cases the value is specified by an OCL expression.

Example 1. Consider the mapping P2S presented in Listing 3.1. At this stage it is only important to consider the syntax, not the purpose of the mapping. The mapping has two `Domains`: domain *uml* (line 2) and domain *rdbms* (line 5). The `GuardPattern` of the domain *uml* has one variable (p) and one predicate. The predicate will evaluate to *True* iff the value of the *kind* attribute of variable p is equal to the string 'persistent'. The `BottomPattern` of domain *rdbms* has one `RealizedVariable`: s. The mapping's bottom pattern has one `RealizedVariable`: $p2s$, and two assignments. The first assignment sets the *name* attribute of variable $p2s$ to the value of the *name* attribute of variable p. ◫

Listing 3.1: Mapping p2s showing the QVTc syntax

```
1 map p2s in demo {
2     uml(p:Package | ) {
3         p.kind = 'persistent';
4     }
5     enforce rdbms() {
6         realize s:Schema
7     }
8     where() {
9         realize p2s:PackageToSchema |
10        p2s.umlPackage := p;
11        p2s.schema := s;
12    }
13    map {
14        where() {
15            p2s.name := p.name;
16            s.name := p2s.name;
17        }
18    }
19 }
```

3.1.2 QVTc Semantics

In QVTc there are two modes of execution: *checking* and *enforcement*:

> In checking mode, a transformation execution checks whether the constraints hold between the candidate models and the trace model, resulting in reporting of errors when they do not. In enforcement mode, a transformation execution occurs in a particular direction, which is defined by the selection of one of the candidate models as the target model. The execution of the transformation proceeds by, first checking the constraints, and secondly attempting to make all the violated constraints hold by modifying only the target model and the trace model. [3]

In both modes, a mapping defines a one-to-one relation between the bottom pattern of the mapping and the bottom patterns of the mapping's domains. That is, the bottom patterns define the constraints that must hold between elements of the candidate models and the trace model. This means that if the constraints of one of the bottom patterns hold, then all the other bottom patterns' constraints must hold too. The one-to-one constraint between the bottom patterns is only checked or enforced if the constraints for each guard pattern of that mapping hold.

"When a transformation is executed in checking mode, all the mappings of the transformation are executed, by matching the patterns, to check the one-to-one constraints" [3]. In checking mode, all realized variables behave like normal variables and assignments are treated as predicates (e.g. the value of a property must match the value to be assigned). In enforcement mode:

> When a transformation is executed in enforcement mode in the direction of a target model, each mapping is executed to enforce the one-to-one constraint. Firstly by matching the patterns (the same as in checking mode), secondly by enforcing the one-to-one constraint if it is violated. Enforcement will only cause changes to model elements of the trace model and the

> target model associated with the domain and its model type [3].

If the mapping has multiple enforced domains, then all domains that do not represent the direction are treated as checkable.

> **Example 2.** Recall the mapping P2S introduced previously (Listing 3.1). The mapping describes the relation between a package from a UML model and a schema in a relational database (RDBMS) model. Thus, the purpose of the mapping is to transform packages into schemas or vice–versa. Consider that the transformation is executed in the direction of the RDBMS domain. That is, the RDBMS model is the target model. Further, consider that there exists a `Package` for which the predicate in line 3 is `True` (the package is `persistent`) and thus, the guard pattern of the UML domain holds. As a result, the bottom patterns must be checked/enforced. The check phase would involve finding a `PackageToSchema` and a `Schema` for which the assignments (treated as predicates) in lines 15 and 16 hold. If the checking phase fails, then the enforcement would depend on what elements are present in the trace model and candidate models. For example, if a `PackageToSchema` exists that is related to the package via the *umlPackage* attribute, then a new `Schema` will be created, and its attributes assigned accordingly. ◫

This research only considers the case of transformation execution in enforcement mode. Further, it only considers *rewrite executions* in which the trace model and target model are considered to be empty at the beginning of the transformation. As a result, bottom patterns in the mapping and target domain are not checked, only enforced. This restriction is a result of the implementation of the Eclipse QVTc virtual machine which does not fully implement the check–enforce semantics of the language. For example, in the mapping P2S presented in Listing 3.1 a new `PackageToSchema` and a new `Schema` would be created for each package that satisfies

the guard pattern. Finally, regarding the QVT specification [3] on bindings: "one valid binding per instance of the type" of the variable, the Eclipse QVTc virtual machine interprets this to mean that variables are assigned elements (instances) of the models that are of the type or of a subtype of the variable.

3.1.3 Transformation Correctness

The correctness of model transformations, namely to guarantee that certain semantic properties hold for a transformation, is a crucial aspect of transformation engineering. The typical correctness properties of a model transformation are termination, confluence (uniqueness) and behaviour preservation [38] Given that the QVT Specification does not provide a formal definition of the semantics of the QVTc language and as a result the correctness of the synthesized control component cannot be proved. However, it is important to note that termination depends on both the semantics of the language and the MTP, and confluence and behaviour on the MTP. Since the control component synthesis does not modify the logic of the original QVTc MTP, the resulting control component respects the termination, confluence and behaviour properties of the original MTP.

Termination condition is partially given by the semantics of the QVTc language due to its declarative nature, that is, the lack of imperative constructs limits the possibility of writing MTP that do not terminate (e.g. via infinite loop). However, an in–place transformation (source and target model is the same model) can still be non-terminating (a rule can produce and consume elements of the same type, creating an infinite loop). By assuming that the MTP is terminating, then the termination condition depends solely on the language semantics. Section 3.3.2 presents how the language semantics are used to define the concept of *transformation correctness* that will be used to guide the control component synthesis in this research.

This section presented an overview of the syntax and seman-

tics of the QVTc language. The level of detail should be sufficient to discuss the contributions of this thesis. Where necessary, additional insights or discussions on the details of the syntax and semantics will be given. The next section introduces the running example used for the remaining of the thesis.

3.2 Running Example: UML to RDBMS Model Transformation

The Unified Modelling Language (UML) to Relational Database Management System (RDBMS) transformation is one of the most used examples in the literature related to model transformations, ranging from language implementations and triple graph grammars [115, 101] to verification and testing [22, 74, 94]. Arguably this happens because it involves two well –known domains of software engineering, because it can be described by a very clear set of rules and finally because it involves a non–trivial degree of complexity. This example is also used in the QVT specification [3] for both the QVT Relations [3] (QVTr) and QVTc languages. This section gives an overview of the transformation and presents some code snippets to explain its structure. The transformation used is based on the one presented in the QVTc language examples in the specification.

Although the original transformation (as presented in [3]) is written in a bi–directional style, for the rest of the discussion it is assumed that the transformation is executed in the RDBMS direction and hence the code has been rewritten as a uni–directional transformation to facilitate the discussion (all assignments in UML domains are re-written as predicates where appropriate, and predicates in the bottom pattern of RDBMS domains have been removed). Effectively, the code used is what would have been obtained from the QVTc to QVTu translation. The UML to RDBMS name can be misleading, in effect, it is a transformation from a minimal UML Classes/Package model to a minimal RDBMS

schema model. Further, for most of the discussion a trimmed down (simplified) version is used for readability[1]. The simplification is not done because of limitations of our approach, but rather because the removed types and attributes in the metamodel (and their corresponding mappings) would introduce duplication in the discussion.

The simplified version does not support inheritance or associations in the UML model, and does not create PrimaryKeys for the RDBMS tables. Additionally, attributes can only be of a PrimitiveDataType. Hereafter the simplified version is referred to as the *UML2RDBMS* transformation. When compared to the original complete transformation the simplified version is labelled as *UML2RDBMS Minimal* and the original transformation as *UML2RDBMS Complete*. Next sections present the metamodels to which the trace model and candidate models conform to, before discussing the rationale behind the mappings.

3.2.1 The Candidate Models' Metamodels

Since in QVTc the middle/trace model has to be explicit, Fig. 3.2 presents the three metamodels used in the transformation. Different colours are used to identify the different metamodels. This section outlines the characteristics of the metamodels.

The simplified minimal UML metamodel has a hierarchical structure with a `Package` containing `Classes` and `PrimitiveDataTypes`, and `Classes` containing `Attributes`. An `Attribute` has a `PrimitiveDataType` as a type, thus it is only possible to model primitive attributes. The simplified minimal RDBMS metamodel also exhibits a hierarchical structure with a `Schema` containing `Tables`, and a `Table` containing `Columns`. The transformation is straightforward: a `Package` becomes a `Schema`, each `Class` becomes a `Table`, and each `Attribute` of a `Class` becomes a `Column` in the respective `Table`.

[1]The complete, original transformation is available in the QVTd Eclipse Project as an example.

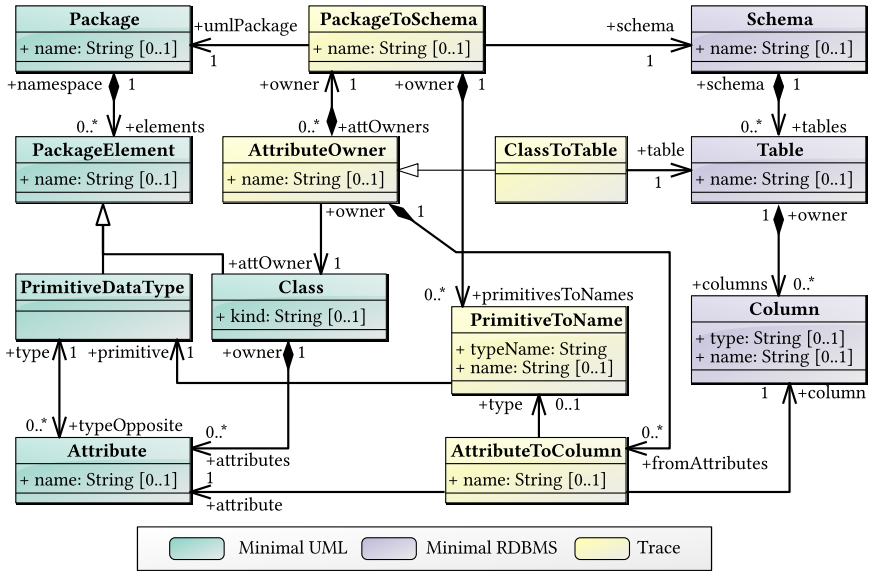


Figure 3.2: Metamodels for the Simplified UML to RDBMS example.

3.2.2 The Transformation Mappings

This section gives an overview of the UML2RDBMS MTP. The accompanying models in Fig. 3.3 present a small example of a UML model and the result of executing the transformation in the direction of the RDBMS domain. The UML model is to the top (green) and the RDBMS model to the bottom (purple). The UML model depicts the structure of a University and its students, courses, and lecturers. For readability the mappings have been named using an acronym of the full name, the listings have a comment with the full name for guidance.

Listing 3.2: Mapping PACKAGETOSCHEMA in UML to RDBMS.

```
11 /** packageToSchema */
12 map p2s in umlRdbms {
13     uml(p:Package | ) { }
14     enforce rdbms() { realize s:Schema }
15     where() {realize p2s:PackageToSchema |
16         p2s.umlPackage := p; p2s.schema := s; }
17     map {
18         where() { p2s.name := p.name; s.name := p2s.name; } }
19 }
```

Listing 3.2 presents the code for mapping PACKAGETOSCHEMA (m_{p2s}). This mapping creates a new `Schema` (line 14) and a `Pack-`

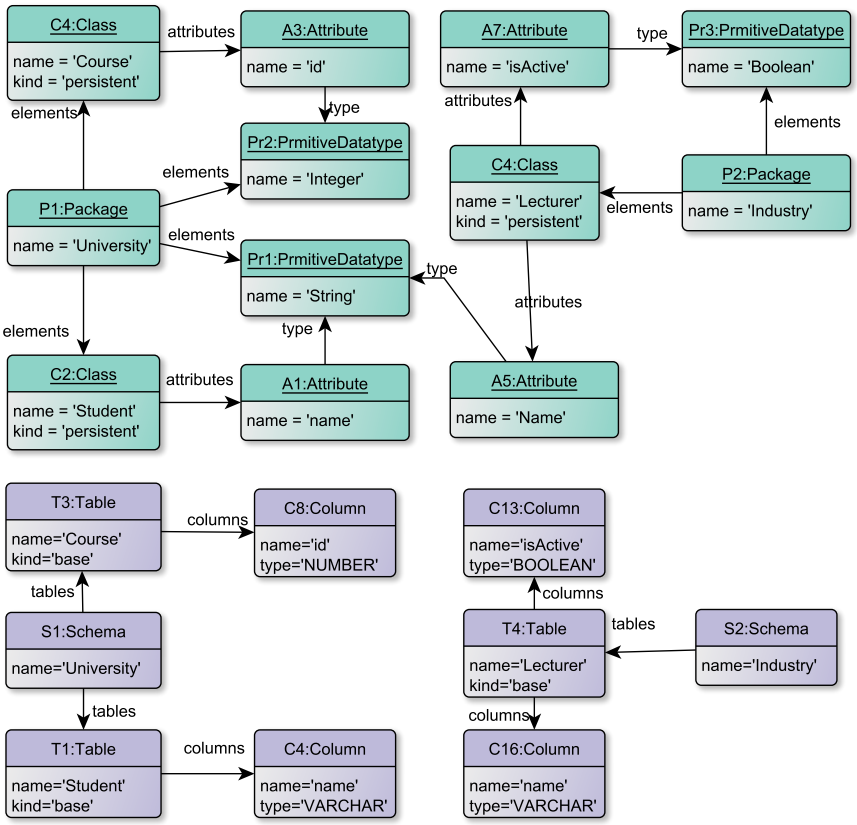


Figure 3.3: Models for the UML (top) to RDBMS (bottom) example.

`ageToSchema` (line 15) for each `Package`. The schema will have the same name as the package (line 18). In the example models in Fig. 3.3 there is a resulting schema with name 'University' and another with name 'Industry'.

Listing 3.3: Mapping integerToNumber in UML to RDBMS.

```
20 /** integerToNumber */
21 map i2n in umlRdbms {
22     uml(p:Package, prim:PrimitiveDataType |
23         prim.namespace = p; prim.name = 'Integer';) { }
24     check enforce rdbms() { sqlType:String | sqlType := 'NUMBER'; }
25     where(p2s:PackageToSchema |
26         p2s.umlPackage = p;) {
27         realize p2n:PrimitiveToName |
28         p2n.owner := p2s; p2n.primitive := prim;
29         p2n.typeName := sqlType; }
30     map {
31         where() { p2n.name := prim.name + '2' + sqlType; } }
32 }
```

Mappings INTEGERTONUMBER (m_{i2n}), STRINGTOVARCHAR (m_{s2v}) and BOOLEANTOBOOLEAN (m_{b2b}) perform the mapping between UML types and RDBMS types. Note that in RDBMS types are only strings stored in the type attribute of the `Column`, thus there is no realized variable in the *rdbms* domain. For example, Listing 3.3 presents the INTEGERTONUMBER mapping, in which a `PrimitiveType` named Integer will be mapped to the *rdbms* string 'NUMBER'. The other two mappings have an identical structure but guard the primitive data type (line 23) by matching is name to 'Boolean' and 'String' respectively, and will result in RDBMS string values of 'BOOLEAN' and 'VARCHAR' respectively.

Listing 3.4: Mapping CLASSTOTABLE in UML to RDBMS.

```
59 /** classToTable */
60 map c2t in umlRdbms {
61     uml(p:Package, c:Class |
62         c.kind = 'persistent'; c.namespace = p;) { }
63     check enforce rdbms(s:Schema |) {
64         realize t:Table |
65         t.kind := 'base'; t.schema := s; }
66     where(p2s:PackageToSchema |
67         p2s.umlPackage = p; p2s.schema = s;) {
68         realize c2t:ClassToTable |
69         c2t.owner := p2s; c2t.attOwner := c; c2t.table := t; }
70     map {
71         where() { c2t.name := c.name; t.name := c2t.name; } }
72 }
```

Listing 3.4 presents the code for mapping CLASSTOTABLE (m_{c2t}). The guards in the mapping validate two conditions: The `Class` must have a 'persistent' *kind* (line 62) and the class' *namespace* (p) must be related via the trace (p2s) to the `Schema` (s) (line 67). In line 69 the trace information is updated and in line 71 the table *name* is assgined the *name* from the class. In the example models in Fig. 3.3, there is a `Table` for each `Class` (all classes are persistent), the resulting tables are correctly contained in the appropriate `Schema`, and each `Table` has one `Column` for each `Attribute` of the `Class`.

Primitive attributes of the Class are transformed by the mapping FROMATTRIBUTE in Listing 3.5 (m_{fa}). Similarly to the CLASS-

ToTable mapping, the guard validates the containment and trace relations between the class, the attribute and the primitive type. Notice that this mapping does not have a *rdbms* domain. The trace `AttributeToColumn` stores all information of the attribute, including a reference to the trace for the attribute's type (line 80).

Listing 3.5: Mapping fromAttribute in UML to RDBMS.

```
73 /** fromAttribute */
74 map fa in umlRdbms {
75     uml(c:Class, t:PrimitiveDataType, a:Attribute |
76         a.owner = c; a.type = t; ) { }
77     where(fao:AttributeOwner, p2n:PrimitiveToName |
78         fao.attOwner = c; p2n.primitive = t;) {
79         realize fa:AttributeToColumn |
80         fa.attribute := a; fa.owner := fao; fa.type := p2n; }
81     map {
82         where() { fa.name := a.name; } }
83 }
```

Finally, mapping attributeToColumn (m_{a2c}) in Listing 3.6 creates the columns in the table for each of the attributes of the class. The `Column` takes its *type* for the trace `PrimitiveToName` and its *name* from the `AttributeToColumn` (line 94).

Listing 3.6: Mapping attributeToColumn in UML to RDBMS.

```
84 /** attributeColumn */
85 map a2c in umlRdbms {
86     check enforce rdbms(t:Table |) {
87         realize c:Column |
88         c.owner := t; }
89     where(c2t:ClassToTable, a2c:AttributeToColumn,
90             p2n:PrimitiveToName |
91             c2t.table = t; a2c.owner = c2t; a2c.type = p2n; ) {
92         a2c.column := c; }
93     map {
94         where() { c.name := a2c.name; c.type := p2n.typeName; } }
95 }
```

3.3 Purpose of the Logic Component

Recall that this research only considers the case in which the QVTc transformation is executed in enforcement mode. In this mode of execution, the purpose of a QVTc transformation is, as stated in

the QVT Specification, to check the constraints and attempt to make all the violated constraints hold by modifying only the target model and the trace model. Given that this is not a formal definition, this section breaks down this definition to the mapping level and formalizes the concept of mapping invocation. This concept is used to define *transformation correctness* such that the definition is in accordance to the purpose of the logic component.

3.3.1 Mapping Execution

In the QVTc syntax (Sect. 3.1.1) a domain specifies a set of model elements of exactly one of the candidate models and the mapping itself (in the mapping's guard and bottom patterns) specifies elements from the trace model. This specification is done via variables. This section presents how the relation between metamodels and models link to the notions of a *type system* [81]. This link will allow to provide a definition of the execution of a mapping that clearly states how the variables from the syntax can be associated to elements of the candidate models and the trace model.

Section 2.1 presented how the relation *conformsTo* between models and metamodels defines type-token relations between elements of the model and elements of the metamodel. As such, each metamodel element Y can be considered to be a class and each model element X can be considered to be an object, where $meta(X, Y) \leftrightarrow type(X) = Y$ and X is an instance–of Y, denoted $X \ltimes Y$. Further, if Z is a subtype of Y, denoted $Z \lhd Y$, and $meta(W, Z)$, then $type(W) = Z$ but $type(W) \neq Y$. However, W is an instance of both Z and Y: $W \ltimes Z$ and $W \ltimes Y$.

Definition 1 (Transformation Mapping). A *transformation mapping* $m = (T, V, S)$ consists of finite sets of T types of interest, V variables, and S statements. A variable can hold a model element as its value and $\forall v \in V, type(v) \in T$. A statement can either be a predicate or an assignment. ⊛

At runtime, the relation $value(v, e)$ denotes that a variable v holds an element e of the candidate models or the trace model as its value, such that given $type(v) = t$ then $value(v, e) \rightarrow e \ltimes t$. Thus, a mapping can be considered a subroutine, where the statements in S are the program instructions and variables in V the parameters.

In general, a subroutine can have input and output parameters. It follows that some mapping's variables can be labelled as input and others as output parameters. In checking mode, all variables are considered inputs. In enforcement mode and rewrite execution (which is the interest of this research), it is possible to identify which variables in the mapping are inputs and which are outputs. Output variables are realized variables in the domains related to the target model and realized variables in the mapping's bottom pattern (trace model). All other variables are inputs.

Definition 2 (Input and Output sets.). Given a transformation mapping me use $IN(m)$ to denote the set of input types of m (types of the input variables of m), and $OUT(m)$ to denote the set of output types of m (types of the output variables of m). ⊛

Example 3. For the UML2RDBMS example the IN and OUT sets for enforcement execution in the RDBMS direction are presented in tabular format in Table 3.1. Note that the $type(v) = t$ relation is depicted in the QVTc concrete syntax by a colon ':', $v : t$. ◫

Definition 3 (Input and Output variable sets.). Given a transformation mapping $m = (T, V, S)$, $V^{IN}(m) = \{v_k \in V | type(v_k) \in IN(m)\}$ and $V^{OUT}(m) = \{v_k \in V | type(v_k) \in OUT(m)\}$ are the set of input and output variables of m, respectively. ⊛

Having defined the IN and OUT sets, the definition of the execution of a mapping in checking and enforcing mode is presented next. Although this research is focused on the enforcing mode the

Table 3.1: IN and OUT sets for the UML2RDBMS example.

	IN	*OUT*
m_{p2s}	Package	PackageToSchema, Schema
m_{i2n}	Package, PrimitiveDataType	PrimitiveToName
m_{b2b}	Package, PrimitiveDataType	PrimitiveToName
m_{s2v}	Package, PrimitiveDataType	PrimitiveToName
m_{c2t}	Package, Class, PackageToSchema, Schema	ClassToTable, Table
m_{fa}	Class, PrimitiveDataType, Attribute, AttributeOwner, PrimitiveToName	AttributeToColumn
m_{a2c}	ClassToTable, AttributeToColumn, PrimitiveToName, Table	Column

checking mode is also defined for completeness and to support definitions introduced later. Given that a mapping m is considered a subroutine, $m(\sigma)$ is used to represent an invocation of a given mapping with a list of arguments σ. The arguments represent an assignment of a value for each of the mapping parameters.

Definition 4 (Checking Mapping Invocation). In checking mode a mapping $m = (T, V, S)$, can be applied to the Cartesian product $\Sigma_m^{\Phi} = E_1 \times \cdots \times E_n$, where $n = |V|$. Each set E_k represents all the possible values $(v_k, e)$of $v_k \in V$, such that $\forall e \in E_k, e \ltimes type(v_k)$. We will use $m^{\Phi}(\sigma)$ to represent the invocation of a mapping in checking mode. ⊛

Definition 5 (Enforcing Mapping Invocation). In enforce mode a mapping $m = (T, V, S)$, can be applied to the Cartesian product $\Sigma_m^{\dagger} = E_1 \times \cdots \times E_n$, where $n = |V^{IN}(m)|$. Each set E_k represents all the possible values of $v_k \in V^{IN}(m)$, such that $\forall e \in E_k, e \ltimes type(v_k)$. We will use $m^{\dagger}(\sigma)$ to represent the invocation of a mapping in enforcement mode. A mapping will generate (new elements) the Cartesian product $\Lambda_m^{\dagger} = R_1 \times \cdots \times R_j$, where $j = |V^{OUT}(m)|$. Each set R_k represents all the possible values of $v_k \in V^{OUT}(m)$, such that $\forall r \in R_k, r \ltimes type(v_k)$ ⊛

Example 4. For mapping m_{c2t} in the UML2RDBMS example, given the models in Fig. 3.3 then $V = \{p, c, s, p2s\}$ and some of

the invocation argument tuples are $(P1\colon Package, S1\colon Schema, C2\colon Class, P2S1\colon P2S)$, $(P1\colon Package, S1\colon Schema, C4\colon Class, P2S1\colon P2S)$, $(P1\colon Package, S2\colon Schema, C2\colon Class, P2S1\colon P2S)$, and $(P1\colon Package, S2\colon Schema, C2\colon Class, P2S2\colon P2S)$. ◫

3.3.2 Transformation Correctness

The definition of the execution of mappings can be related to the purpose of the logic component based on transformation execution . For both *checking* and *enforcing* mapping invocations there are three possible outcomes: NA (not applicable), True or False For both invocation modes, evaluation is NA if $\exists v_k \in V \,|E_k = \emptyset$, i.e. there are no elements available to be assigned as variable values. If applicable then: For *checking* mode, the result of invocation is True if all constraints in the mapping hold and False if not; for *enforcing* mode, the result of invocation is True if all guards in the input domains hold and False if not.

For example, an invocation of mapping m_{c2t} (Listing 3.4) with a `Class` element as an argument, for which its *kind* attribute is not equal to 'persistent´ will result in False. Further, any tuple with $value(p, P1)$ and $value(s, S2)$ will also result in False, given that schema $S2$ is not the schema created to represent package $P1$ and hence the guard of the mapping's bottom pattern will fail. the elements in σ are not applicable to the mapping.

In enforce-rewrite execution, at the start of the execution all mappings with a guarded domain for the target model will evaluate to NA; variables in the target and trace models have undefined values. At the end of the transformation there should exist at least one parameter tuple for each mapping such that invocation of the mappings results in True (assuming the transformation is free of errors and the source model(s) contain at least one element of the input types of interest of each mapping). Hence, $m^{\dagger}(\sigma) = True \implies m^{\Phi}(\sigma) = True$. That is (in a defect-free implementation of a QVTc execution engine), if there are no runtime

exceptions, it is assumed that enforcement invocation always results in the necessary modifications in the target and trace model so that the constraints in the transformation hold. Having formalized the concept of mapping invocations and related it to the purpose of the logic component the definition of *transformation execution correctness* is introduced next, in order to formalize the purpose of the logic component at the transformation level.

Definition 6 (QVTc Transformation). A QVTc transformation $Q = (O, M)$, consists of finite sets of O model–types and M mappings. A model–type is defined by one or more metamodels, which in turn define the set of classes and property elements that are expected by the transformation. At runtime, a candidate model is paired with a model–type. This pairing implies that the candidate model contains elements of types defined by the model–type's metamodels. A domain represents the candidate model linked to the model–type. Additionally, the trace model is also paired with a model–type and consequently the trace model contains elements of types defined by the model–type's metamodels (the trace metamodel(s)). A mapping represents the trace model linked to the model–type. ⊛

Definition 7 (Transformation execution correctness). Given a QVTc transformation $Q = (O, M)$, the transformation execution is *correct*, if $\forall m \in M.\, \forall \sigma_i \in \Sigma_m . m^{\Phi}(\sigma_i) = m^{\dagger}(\sigma_i)$ at the end of the executionin enforcing mode. ⊛

This section introduced an initial set of definitions that allow reasoning about the purpose of the logic component at the mapping level and from this define the concept of transformation execution correctness. This definition will be fundamental in validating any systematic process for synthesizing control components.

3.4 Control Component Model

An abstract representation of the problem can be useful to perform analyses and to apply optimization algorithms or techniques. This section presents the *Control Component Model*, an abstract representation defined for the control component. This model facilitates "abstract interpretation to be employed when designing algorithms, which facilitates systematic algorithm development and proof of correctness" [82]. For example, this model can be used to validate that a specific control component observes transformation execution correctness.

The proposed model exhibits the properties suggested by Pingali et al. [82]:

- It should be executable.
- The representation must be efficiently traversed for data dependence information.
- Loops should be represented explicitly, and the representation should be compact.

Executability is particularly important as the main goal of the control component is to execute the QVTc transformation. This type of representation has been proposed before for other declarative languages, such as SQL [25, 55]. This section first identifies the functionality that is provided by the existing QVTi virtual machine (see Sect. 2.4.2) and from this determines what functionality must be provided by the control component. The functionality needed will be used to determine what needs to be modelled.

The QVTi virtual machine is capable of executing mappings, i.e. evaluate all the statements in the mapping for a given set or arguments, and implements the QVTc semantics as defined in the specification [3]. Further, it provides access to the candidate models and trace model and their elements (for both query and alteration), and also provides functionality to retrieve all elements that are instances of a given type. However, it does not provide

a control component. Given the functionality provided by the existing QVTc engine, the synthesized control component must be responsible for calculating Σ for each mapping and invoking each mapping for all the tuples in Σ. We refer to the process of calculating Σ_m as element *ascription*. In the context of this project, ascribing is understood as the action of assigning a model element as the value of a variable.

Definition 8 (Control component Objective). The objective of the control component is to guarantee that all possible permutations of elements for every mapping are considered for ascription and that each mapping is invoked for all possible permutations. ⊛

Based on the above, the model for representing a control component must support the abstraction of its two main functions. We will refer to these functions as *execution actions*. The two available execution actions are: *All-Of-Kind Loop* (all-loop for short) and *Invoke Mapping* (map for short). The all-loop action iterates over the elements of the candidate models that are of a given type or any of its sub-types (denoted as $A(type)$); the map action represents an enforce invocation of a given mapping (denoted as $I(m)$). The term *execution plan* is used hereafter to refer to an abstract representation of a control component built using execution actions, as presented in Fig. 3.4.

The execution plan is a graph in which nodes represent execution actions and edges represent calls[2] from one action to another. The execution plan is a directed graph with a specially designated *root* node that represents the execution entry point. A call to a map action results in the invocation of the mapping associated to the action. A call to an all-loop action results in the iteration over all the elements of the given kind and all calls outgoing from the all-loop action are called once for each iteration. Outgoing calls from a map action are executed only if $m^{\dagger}(\sigma) = \text{True}$.

[2]The term *call* is used to to differentiate execution of the actions from execution (invocation) of the mapping.

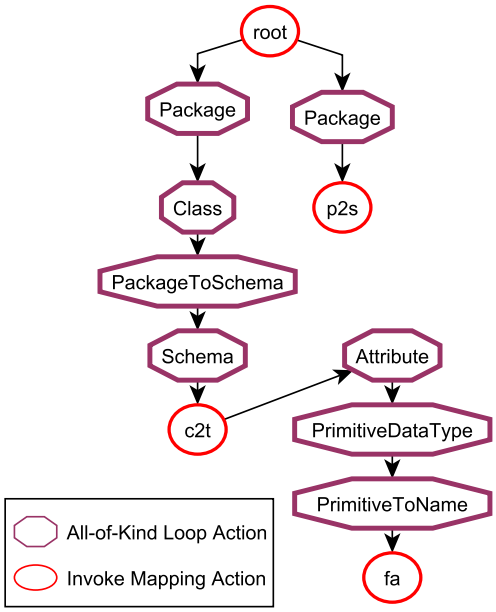


Figure 3.4: An example abstract representation of the control component.

The following constraints are defined for the execution plan:

- *All-Of-Kind Loop* actions can only have one incoming and one outgoing edge.
- *Invoke Mapping* actions can have multiple incoming and outgoing edges.
- The execution plan is not a multigraph, and hence an action can only invoke another action once.

Definition 9 (Invocation path). An *invocation path* $ip(m_s, m_t) = \{I(m_s), c_1, A(t_1), c_2, \ldots,$
$A(t_k), c_n, I(m_t)\}$ is a finite sequence of alternating actions and calls (c), which begins at map action $I(m_s)$ and ends at map action $I(m_t)$.

Invocation paths define the sets that are available for ascription for $I(m_t)$ as follows:

Definition 10 (Map action ascription sets). In an execution plan, given the path $ip(m_s, m_t)$ ascription for invocation $I(m_t)$ can be applied to the Cartesian product $L_1 \times L_2 \times \ldots \times L_k \times R_{s_1} \times R_{s_2} \times \ldots \times R_{s_p}$, where L_i represents the elements iterated by all-loop action $A(t_i)$ in the path, and $R_{s_1} \ldots R_{s_j}$ are the output elements of m_s (from Definition 5). ⊛

Execution plans are executable, with execution semantics defined by outgoing edge order and edge traversing. Thus, execution can be guided by a depth-first traversal starting at the root node. To make execution repeatable, the execution plan graph guarantees outgoing edge traversal order. For example, the execution plan model can be easily used for interpreted execution or as a model for code generation and compiled execution.

The execution plan is distinctly different from a physical operator tree used for SQL optimization [25], mainly by the fact that the physical operator tree is designed to allow tree reduction, i.e. the leaves represent data access in the database, and branches represent how the data is merged, joined, looped, etc., to produce the complete SQL query result. Hence, physical operator trees are executed (reduced) bottom-up.

Example 5. Figure 3.4 presents a part of a possible execution plan for the UML2RDBMS transformation. For mapping m_{p2s} the only all-loop action in a path from the root is Package, thus the map action ascription sets for $I(m_{p2s})$ are L_{Package}. For mapping m_{c2t} there are four all-loop actions, thus the map action ascription sets for $I(m_{c2t})$ are L_{Package}, $L_{\text{PackageToSchema}}$, L_{Class}, and L_{Schema}. For mapping m_{fa}, the sets include the output elements of the `c2t` mapping and the three all-loop actions. Thus, the map action ascription sets for $I(m_{fa})$ are $R_{\text{ClassToTable}}$, R_{Table}, $L_{\text{Attribute}}$, $L_{\text{PrimitiveDataType}}$, and $L_{\text{PrimitiveToName}}$. ◫

Finally, it is crucial that all mappings in the transformation are included in the execution plan. For that, the concept of execution plan completeness is introduced.

Definition 11 (Execution Plan Completeness). An execution plan is *complete* if for each mapping in the transformation there exists an invocation action and the invocation action has an incoming call edge. ⊛

This section defined a model to represent the control component and presented how element ascription is realized through paths in the model. The next chapter describes how this model and the ascription semantics can be used to explore how the proposed optimizations can be achieved by changing the structure of the model.

3.5 Summary

This chapter presented an overview of the syntax and semantics of the QVTc and defined the control component model. This model will facilitate the analysis of control components and enable systematic construction of execution plans. The chapter also provided the definition of transformation execution correctness that is fundamental in validating the synthesized control components. The next chapter shows examples of sub-optimal plans and shows how the knowledge embedded in the logic can be used to construct plans that are more optimal.

4

Execution Plan Synthesis Problem

The last chapter introduced the QVT Core [3] (QVTc) language, described the relation between the language semantics and the logic component and used the purpose of the logic component to define the execution of QVTc transformations at the mapping level. The chapter then defined *transformation correctness* as a formalization of the purpose of the logic component, based on the proposed execution definitions. With these definitions in place, the chapter presented the execution plan as an abstract representation (model) of the control component and defined *execution plan completeness*. The definitions and the execution plan are the first steps towards systematic synthesis of control components for QVTc transformations.

This chapter defines the control component synthesis problem as a scheduling problem and looks at the knowledge embedded in the logic component in order to identify the information required to find a solution. First, it discusses that part of that knowledge is in the form of the data dependence information and then it shows how data dependence analysis can be used to find a partial order of the mappings in a transformation. A partial order is required for solving scheduling problems and therefore to solve the control com-

ponent synthesis problem. Then, it defines execution plan validity and explores how the knowledge embedded in the logic component can be used to achieve the goals of the scheduling problem. The definitions and restrictions introduced in this chapter will be used in the following chapter to guide the systematic construction of solutions to the scheduling problem.

Section 4.1 presents the problem definition and Sect. 4.2 defines the partial ordering of mappings. Next, Sect. 4.3 discusses the limitations of the partial order to provide a correct transformation execution and from these define execution plan validity. The objective of Sect. 4.4 is to present how context reuse can be used to minimize the number of loops over elements of a type needed in the execution plan, and Sect. 4.5 shows how data dependency analysis can be used to minimize the number of mapping invocations. Finally, Sect. 4.6 summarizes and concludes the discussion.

4.1 Problem Definition

Given that the execution plan is a model of the control component, synthesis of the control component is equivalent to synthesis of the execution plan. Section 3.4 defined execution plans and characterized them as executable via depth first traversal. We will refer to the invocation order derived from the depth-first traversal as the execution plan's schedule. From this, a mapping m_i is said to be *scheduled* after mapping m_j if invocation of m_i follows m_j in the traversal (it does not have to be immediate). Consequently, synthesis of the execution plan can be viewed as a scheduling problem where the goal is to construct a tree–like structure that represents the desired execution order.

> Given the set of mappings of a QVTc transformation, the mapping scheduling problem involves scheduling the mappings so that the execution time of the transformation is minimized, while ensuring transformation correctness (from Definition 7).

In compiler optimization, the instruction scheduling problem is related to finding the optimal order in which instructions in a program should be executed. The execution plan synthesis problem is related to the scheduling problem because the solution is based on finding an optimal order in which mappings in a QVTc transformation should be invoked. Since the instruction scheduling problem is $\mathcal{NP}$-hard [13], metaheuristics can be successfully used to solve this kind of problems [109, 108, 7, 52, 92]. Given the aforementioned relation, this research approaches the execution plan synthesis problem as a scheduling problem and uses a metaheuristic to solve the synthesis problem.

Further, the solution space for the execution plan synthesis problem can quickly become intractable, as discussed next. When the solution space becomes intractable, metaheuristic algorithms provide the means to find good enough solutions in a reduced amount of time, by enabling efficient and effective exploration of the solution space [20]. The execution plan can be considered as a labelled tree (each mapping name is a label) in which the number of labelled trees on n nodes, i.e. the solution space, is n^{n-2} [24]. However, given the vertex set $\{1, 2, 3\}$, the number of trees calculated in Cayley [24], considers tree $1 \backsim 2 \backsim 3$ to be equal to tree $3 \backsim 2 \backsim 1$. That is, trees with the same forward and backward ordering are considered equal. Since in the execution plan invoking mappings in $1 \backsim 2 \backsim 3$ is different from invoking them in $3 \backsim 2 \backsim 1$, the solution space for the execution plan synthesis problem is larger than n^{n-2}. Otter [79] proved an asymptotic estimate to calculate the number of labelled trees which takes into consideration the ordering. Deriving such an estimate for the case of execution plans is out of the scope of this research, but it suffices to state that both approaches suggest that the solution space grows exponentially with the number of mappings.

Finally, for a given tuple of parameters σ of candidate and trace model elements, the *enforce* invocation $m^{\dagger}(\sigma)$ can be considered constant across different execution plans. The reason for this is

that the execution plan does not modify the transformation mappings and hence, mapping execution (consider atomically) does not depend on the structure of the execution plan. Thus, time differences between execution plans only depend on the number of times each mapping is invoked and the calculation of Σ (all possible combinations of σ) for each mapping invocation. Additionally, invocation of mappings is non–pre–emptive (based on the used implementation of the QVTc execution engine), that is, once a mapping is invoked it must finish without being interrupted.

Based on all the above, the mapping scheduling problem is defined as

1. Minimize the number of times each mapping is scheduled.
2. Minimize the number of times each mapping is invoked.
3. Minimize the number of invocations with a NA result.
4. Minimize the number of invocations with a False result.

4.2 Mapping Order using Inter-Mapping Data Dependency Analysis

A fundamental aspect of solving the execution plan scheduling problem is to identify a partially ordered relationship among the mappings [7, 109]. This section shows that a partial ordering can be found using data dependence analysis. Data dependence analysis has been used in compiler optimization of imperative languages [116, 47], declarative languages [45, 25] and functional languages [67]. Data dependence analysis is mainly used to optimize instruction scheduling (imperative languages) and to define instruction scheduling (declarative/functional languages), and can be done statically or dynamically. Given that this research focuses on synthesis of the control component as part of compilation, the

data dependence analysis only uses information from static analysis of the QVTc transformation program and the properties of the metamodels of the candidate models and the trace model. This section presents the use of data dependency analysis to define a set of dependency relations between mappings that allow identification of a partial ordering of mappings.

In the optimization of functional languages (a form of declarative languages) the data dependence analysis is used to construct a dataflow graph [12]. The dataflow graph representation only describes the flow of data between instructions. Each node represents an instruction and the edges of the graph represent FIFO channels. An instruction consumes data from its incoming FIFO channels and places produced data in its outgoing FIFO channels. Based on the functional languages case, this research proposes the use of an alternative representation in the form of an *element flow graph*, in which each node represents a mapping and each edge represents a data dependency between two mappings. The edges represent the flow of elements between mappings. The FIFO channels are not required because data is managed directly by the QVTi virtual machine.

Data analysis for a mapping can be done by looking at the variables, predicates and assignments of the mapping. To define a partial ordering of mappings it is only necessary to consider the mappings as atomic blocks and to identify the dependencies that exist due to the flow of elements between them. Given that variables define the types of elements of interest of a mapping and the types of new elements created by the mapping, partial ordering is defined only based on data dependence at the variable level. The type information that is embedded in the mapping's variables is captured in the IN and OUT sets (Definition 2).

To represent the flow of elements in the QVTc transformation, this section discusses the data relations between mappings (inter-mapping relations) and uses these relations to define and give examples of inter-mapping data dependence relations in a

QVTc transformation. These definitions are based on the definitions by Wolfe et al. [116] for data dependence analysis for imperative languages.

4.2.1 Inter-Mapping Data Dependence Relations

This section introduces the concepts of data dependence relations and data dependence graph (the element flow graph) for a QVTc transformation.

Definition 12 (Data Dependence Relations). Given two mappings m_v and m_w, the following data relations may hold true or the mappings may be data independent.

1. If some type $X \in OUT(m_v)$ and $X \in IN(m_w)$, then m_w is *data-type dependent* on m_v denoted by $m_v \delta^{\tau} m_w$. For more detailed specifications the type that defines the dependency can be supplied (or a set of types): $r_v \delta^{\tau}_X r_w$, where X is the type that defines the relation.

2. If some types $X \in OUT(m_v)$ and $Y \in IN(m_w)$, and $Y \lhd X$ (Y is a subtype of X, see Sect. 2.1), then m_w is *data-kind dependent* on m_v denoted by $m_v \delta^{\gamma} m_w$. For more detailed specifications the type that define the dependency can be supplied (or a set of types): $m_v \delta^{\gamma}_{Y \lhd X} m_w$, where $Y \lhd X$ is the type inheritance that defines the relation. ⊛

Definition 13 (Indirect Data Dependence). Mapping m_w is *data dependent* on m_v, denoted $m_v \delta m_w$, if $m_v \delta^{\tau} m_w$ or $m_v \delta^{\gamma} m_w$. For more detailed specifications the type that define the dependency can also be supplied (or a set of types): $m_v \delta_X m_w \implies m_v \delta^{\tau}_X m_w \vee m_v \delta^{\gamma}_{? \lhd X} m_w$. The question mark denotes that the data-kind dependent relation can be to any subtype of X. Mapping m_w is *indirectly data dependent* on m_v,

denoted $m_v \Delta m_w$, if there are mappings $m_{v_1}, \ldots, m_{v_n}, n > 0$, such that $m_v \delta m_{v_1}, m_{v_1} \delta m_{v_2} \ldots m_{v_{n-1}} \delta m_{v_n}, m_{v_n} \delta m_w$. ⊛

Definition 14 (Data Dependence Graph). All data dependencies in a QVTc transformation of n mappings can be represented by a *data dependence graph* **G** of n nodes, one for each $m_i (1 \leq i \leq n)$. For each δ^τ and δ^γ relation between m_v and m_w, there is a corresponding edge in **G** from the node representing m_v to the node representing m_w. ⊛

Example 6. The data dependence graph for the UML2RDBMS example is shown in Fig. 4.1. Note that the figure only shows one of the primitive data type mappings: INTEGERTONUMBER (m_{i2n}), as the other mappings for primitive types have the same dependencies. The data dependencies are as follows:

$m_{p2s} \delta^\tau_{PackageToSchema} m_{i2n}$ $\quad$ $m_{p2s} \delta^\tau_{PackageToSchema} m_{c2t}$

$m_{p2s} \delta^\tau_{Schema} m_{c2t}$ $\quad$ $m_{i2n} \delta^\tau_{PrimitiveToName} m_{fa}$

$m_{i2n} \delta^\tau_{PrimitiveToName} m_{a2c}$ $\quad$ $m_{c2t} \delta^\tau_{ClassToTable} m_{a2c}$

$m_{c2t} \delta^\tau_{Table} m_{a2c}$

$m_{c2t} \delta^\gamma_{AttributeOwner \triangleleft ClassToTable} m_{fa}$

$m_{fa} \delta^\tau_{AttributeToColumns} m_{a2c}$

$m_{p2s} \Delta m_{fa}$ $\quad$ $m_{p2s} \Delta m_{a2c}$

$m_{c2t} \Delta m_{c2t}$ $\quad$ $m_{i2n} \Delta m_{a2c}$

◫

4.2.2 Loops in the Data Dependence Graph

In the general case the data dependence graph (DDG) is not guaranteed to be a Directed Acyclic Graph (DAG). This section presents the two types of loops that can exist in the DDG: entry–point loops and closed loops. The section discusses how the former does not constrain the domain of DDG supported by the proposed solution and the latter does.

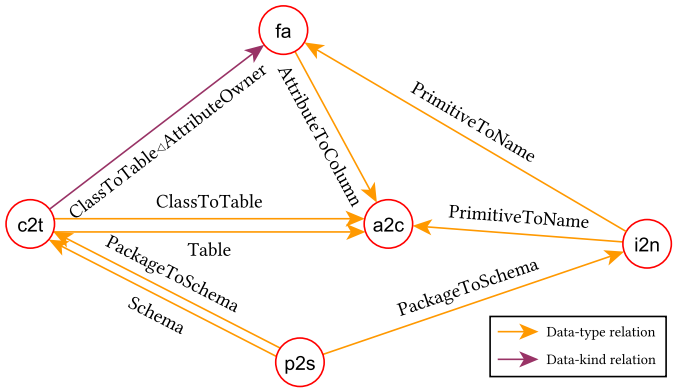


Figure 4.1: Data Dependence Graph for the UML2RDBMS example.

Entry–point Loops Consider that the metamodels in the UML2RDBMS are extended in order to support nested packages and nested schemas (e.g. to represent multi-level federated databases) to the metamodels in the UML2RDBMS example. A package can then have multiple nested packages and a parent. The same applies to schemas. Listing 4.1 presents a new mapping that will support these changes.

Listing 4.1: Mapping to support nested packages and schemas in UML2RDBMS.

```
97 map np in umlRdbms {
98     uml(p1:Package, p2:Package |
99         p2.parent = p1;) { }
100    enforce rdbms(s1:Schema) {
101        realize s2:Schema | s2.parent := s1;}
102    where(p2s1:PackageToSchema |
103        p2s1.umlPackage = p; p2s.schema1 = s; )          {
104        realize p2s2:PackageToSchema |
105        p2s2.umlPackage := p2; p2s2.schema := s2; }
106    map {
107        where() { p2s2.name := p2.name; s2.name := p2s2.name; } }
108 }
```

This mapping will result in a loop in the DDG given that $IN(m_{np}) \cap OUT(m_{np}) = \{PackageToSchema, Schema\}$, as presented in Fig. 4.2 (the mappings that are not involved in the loop are not showed). Although m_{np} is in a loop, given that $OUT(m_{p2s}) = \{PackageToSchema, Schema\}$, we say that this

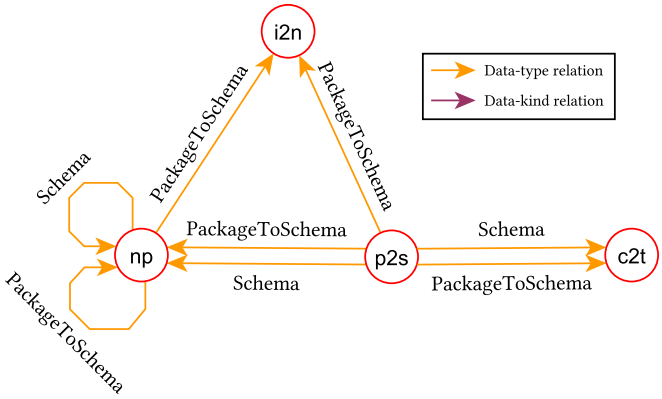


Figure 4.2: Data Dependence Graph (partial) for the UML2RDBMS example showing a dependence loop.

loop provides an *entry point*. The entry point allows m_{np} to execute at least once (by using the output of m_{p2s}) and hence it is possible to construct a plan in which m_{np} is invoked and satisfy Definition 11. Note that entry–point loops also apply for the case in which the loop for m_{np} is larger than a self-loop.

Closed Loops Now consider the case where an UML Class can have one super class, and a class can be the superclass of only one class, and that the RDBMS tables support a similar structure[1]. Mapping m_{c2t} can be modified and add an additional mapping added to consider the super class when creating the table, as presented in Listing 4.2. In this case $IN(m_{c2t}) = \{Package, Class, Schema, Table, PackageToSchema, ClassToTable\}$, $IN(m_9) = \{Class, Table, ClassToTable\}$, $OUT(m_{c2t}) = \{Table, ClassToTable\}$ and $OUT(m_9) = \{Table, ClassToTable\}$. However, there are no other mappings that produce $\{Table, ClassToTable\}$, and hence $m^{\dagger}_{c2t}(\sigma) = \mathsf{NA}$ and $m^{\dagger}_{9}(\sigma) = \mathsf{NA}$ always. As a result, no tables would be generated, and hence the transformation would be incorrect.

[1]This is not a realistic case, but helps to demonstrate the problem with closes loops without introducing an additional example.

Listing 4.2: Mappings in with a closed looped in the DDG

```
1  /** classToTable */
2  map m5 in umlRdbms {
3      uml(p:Package, c:Class, sc:Class |
4          c.superClass = sc; ... ) { }
5      check enforce rdbms(s:Schema, rt:Table |) {
6          realize t:Table |
7          t.superTable := rt; ...}
8      where(p2s:PackageToSchema, sc2t:ClassToTable |
9          p2s.umlPackage = p; p2s.schema = s;
10         sc2t.attOwner = sc; sc2t.table = rt;) {
11         realize c2t:ClassToTable |
12         c2t.owner := p2s; c2t.attOwner := c; c2t.table := t; }
13     ...
14 }
15 /** supperClassToTable */
16 map m9 in umlRdbms {
17     uml(c:Class, sc:Class |
18         sc.subclass = c;) { }
19     check enforce rdbms(t:Table) {
20         realize rt:Table; |
21         rt.kind := 'super'; rt.part = rt;}
22     where(c2t:ClassToTable |
23         c2t.attOwner = c; c2t.table = t;) {
24         ...
25     }
26 }
```

A different strategy is needed to resolve closed loops, i.e. loops without an entry point. One alternative is the one adopted by the Epsilon Transformation Language (ETL) and the Atlas Transformation Language (ATL) in which a first iteration over the mappings creates all required new elements and a second iteration executes the statements in the mapping. This solution is provided at the execution engine level. Making the changes to the existing QVTc execution engine to add this functionality is out of the scope of this project (see Chap. 6 and Chap. 7).

For transformations with loops in the DDG the algorithm will fail graciously and issue an error message to the user indicating that a loop was found and that as a result a plan cannot be synthesized. The message includes information about the mappings and the types involved in the loop. Note that transformations with closed loops cannot be correctly executed with the *naïve plan* either. In fact, this is a limitation of the implementation of the

QVTc execution engine used in this research. The reason being that creation of elements and assignment of their properties is an atomic operation, i.e. execution of the mapping. For closed loops a two stage create–update approach is needed (as implemented in the ETL and ATL execution engines). The create–update approach can also be manually enforced by splitting the mappings involved in the loop so that creation and assignment of properties happen in separate mappings. Hence, the developer can use the error information to modify the transformation and remove the loop.

4.2.3 Precedence-Based Mapping Ordering

Precedence mapping ordering avoids reaching a state in which elements of a type are not available for ascription for a mapping invocation, i.e. $m^{\dagger}(\sigma) = NA$. Therefore, precedence mapping ordering allows minimization of the number of invocations with a NA result.

Example 7. Consider the execution plan for the UML2RDBMS example presented in Fig. 4.3. For static analysis it is assumed that all-loop actions over types of the source model will always generate elements. The path to the invocation of m_{a2c} from *root* has an `AttributeToColumn` all-loop action. Since m_{fa} has not executed yet, there are no `AttributeToColum` elements for ascription[2] (see Sect. 3.4), and hence $m^{\dagger}_{a2c}(\sigma) =$ NA. On the contrary, for m_{fa} there is a `PrimitiveToName` kind action but in in this case, m_{i2n} has executed so there are some `PrimitiveToName` elements available for ascription. ◫

The term *starvation* is used in this research to refer the situation where a mapping invocation $m^{\dagger}(\sigma) =$ NA due to a lack of elements of a type in $IN(m)$. From this it follows that to avoid starvation, if for a given type X mapping m_w has only one $m_v \delta_X m_w$,

[2] An all-loop with no elements of the type will generate one dummy element which effectively assigns an 'undefined' value to the respective variable.

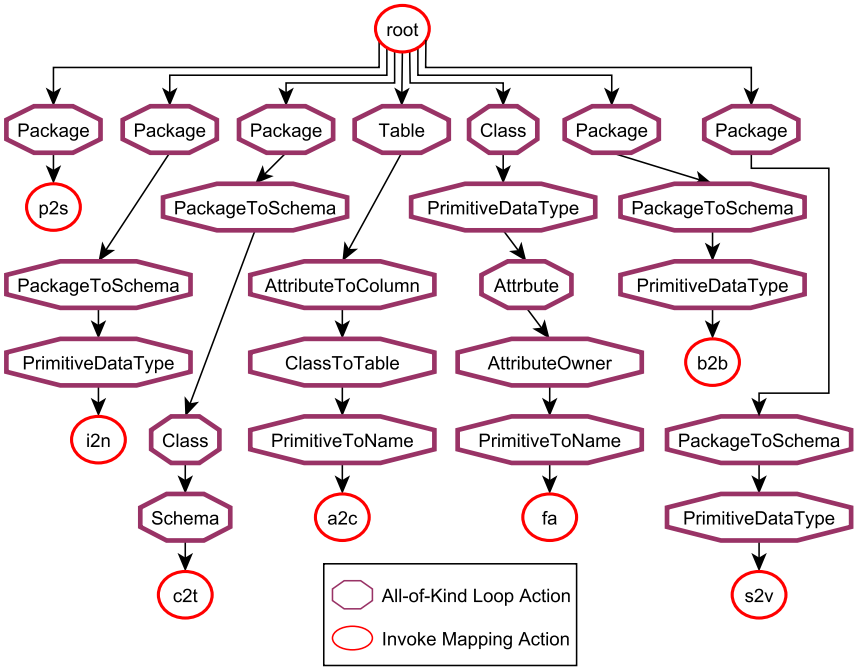


Figure 4.3: One possible Execution Plan for the UML2RDBMS example.

then m_v *must* be invoked before m_w. On the other hand, if for a given type X mapping m_w has multiple $m_{v_1}\delta_X m_w, m_{v_2}\delta_X m_w, \ldots, m_{v_n}\delta_X m_w$, then *at least one* of the m_v mappings must be invoked before m_w. Thus, data dependence relations can be used to identify precedence relations that allow us to prevent starvation. Preventing starvation is equivalent to minimizing invocations with a NA result. Partial ordering also contributes to minimizing the number of times a mapping is invoked because it is possible to find, for each mapping, an order which maximises the number of successors that are scheduled after the mapping. By maximizing the number of invoked successors, the number of invocations needed for a mapping is minimized.

Next, this section defines and gives examples of inter-mapping precedence dependence relations in a QVTc transformation.

Definition 15 (Consumer and Supplier sets). The set $CNS(m_i)$ denotes the *consumer* mappings of m_i (mappings for

which $m_i \delta m_j$), and the set $SPL(m_i)$ denotes the *supplier* mappings of m_i (mappings for which $m_j \delta m_i$). ⊛

Definition 16 (Precedence Relations). If $SPL_t(m_v) \subseteq SPL(m_v)$ is the subset of supplier mappings of m_v for a given type t, where $SPL_t(m_v) = \{m : m \in SPL(r_v) \wedge m \delta_t m_v\}$, the following relations are defined to represent the execution order of mappings.

1. If $|SPL_t(m_v)| = 1$ then m_v is *execution–compulsory dependent* on $SPL_t(m_v)$, i.e. execution of the mapping in $SPL_t(m_v)$ must precede execution of m_v. Given $m_w \in SPL_t(m_v)$, this relation is denote by $m_w \Phi_t m_v$.

2. If $|SPL_t(m_v)| > 1$, then m_v is *execution–alternative dependent* on $SPL_t(m_v)$, i.e. execution of at least one mapping in $SPL_t(m_v)$ must precede execution of mapping m_v. Given $m_w \in SPL_t(m_v)$ we denote this by $m_w \Psi_t m_v$. ⊛

Example 8. For the UML2RDBMS example the data precedence relations are as follows (m_{b2b} and m_{s2v} represent the BOOLEANTOBOOLEAN and STRINGTOVARCHAR mappings respectively):

$$m_{p2s} \Phi_{PackageToSchema} m_{i2n} \qquad m_{p2s} \Phi_{PackageToSchema} m_{b2b}$$
$$m_{p2s} \Phi_{PackageToSchema} m_{s2v} \qquad m_{p2s} \Phi_{PackageToSchema} m_{c2t}$$
$$m_{p2s} \Phi_{Schema} m_{c2t} \qquad m_{i2n} \Psi_{PrimitiveToName} m_{fa}$$
$$m_{b2b} \Psi_{PrimitiveToName} m_{fa} \qquad m_{s2v} \Psi_{PrimitiveToName} m_{fa}$$
$$m_{c2t} \Phi_{AttributeOwner} m_{fa} \qquad m_{c2t} \Phi_{Table} m_{a2c}$$
$$m_{c2t} \Phi_{ClassToTable} m_{a2c} \qquad m_{i2n} \Psi_{PrimitiveToName} m_{a2c}$$
$$m_{b2b} \Psi_{PrimitiveToName} m_{a2c} \qquad m_{s2v} \Psi_{PrimitiveToName} m_{a2c}$$
$$m_{fa} \Phi_{AttributeToColumn} m_{a2c}$$

◫

Definition 17 (Mapping order). We use the relationship $m_w \preccurlyeq m_v$ to denote the *order* between mappings m_v and m_w, such that $m_w \preccurlyeq m_v \implies m_w \Phi_t m_v \vee m_w \Psi_t m_v$ ⊛

Any synthesized execution plan must respect the *mapping order*.

Theorem 1: *Let D be the mapping order of a QVTc transformation, then D is a non-strict partial order.*

Proof: Reflexive *If $m_v \preccurlyeq m_v$ then $m_w \Phi_t m_v \vee m_v \Psi_t m_v$. $m_w \Phi_t m_v$ is not possible as it represents a closed loop. If $m_v \Psi_t m_v$, then $|SPL(m_v)| > 1$ and there must exist another mapping m_x, such that $m_x \delta m_v$, which represents an entry—loop, which are allowed.*

Antisymmetric *The mapping order is not antisymmetric. If there exists mappings m_v, m_w, m_x with $v \neq w \neq x$ and $m_x \delta_t m_v$, $m_x \delta_t m_w$, $m_v \delta_t m_w$, $m_w \delta_t m_v$, then $m_x, m_w \in SPL(m_v)$ and $m_x, m_v \in SPL(m_w)$ resulting in $m_v \preccurlyeq m_v$ and $m_w \preccurlyeq m_v$ but $m_v \neq m_w$.* ⊡

4.2.4 Completing the Data Dependency Graph

The definition of the DDG builds on the data dependence relations, and these build on the IN and OUT sets of mappings. However, it is usually the case that the source model(s) is(are) not modified in a transformation and as a result the types of the source metamodels do not appear in the OUT sets of any of the mappings. As a result, it would be impossible to construct a partial ordering that satisfies the data dependency relations that involve the source metamodels' types.

Going back to the producer-consumer analogy, the data dependence is effectively missing a mapping that produces the elements

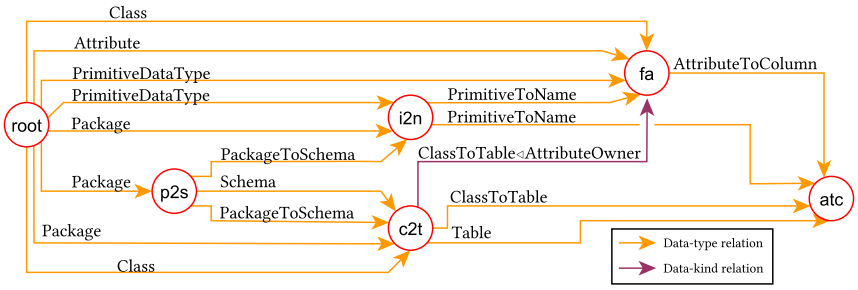


Figure 4.4: Completed Data Dependence Graph for the UML2RDBMS example.

that exist in the source model(s) at the beginning of the transformation execution. The definition of the control component model has a *root* node that represents the execution entry point. Since this action node does not represent invocation of an actual mapping, it can be safely assumed that it represents a dummy mapping m_α that in turn is responsible for producing elements of the types of the source metamodel(s). Since the root is the entry point of execution, all precedence relations to it are automatically satisfied and all partial orders with respect to the root can be respected too. Figure 4.4 presents the DDG for the UML2RDBMS with the source types added. The figure uses a hierarchical layout (left to right) to provide a notion of the partial order of mappings.

4.3 Ensuring Transformation Correctness

Although the precedence partial order works to prevent starvation, the lack of starvation in an execution plan is not enough to guarantee that the plan will result in a correct transformation execution.

Consider the execution plan for the UML2RDBMS example presented in Fig. 4.5. This plan respects the precedence partial order but results in an incorrect transformation execution. The reason is that mappings m_{b2b} and m_{s2v} are executed at the end

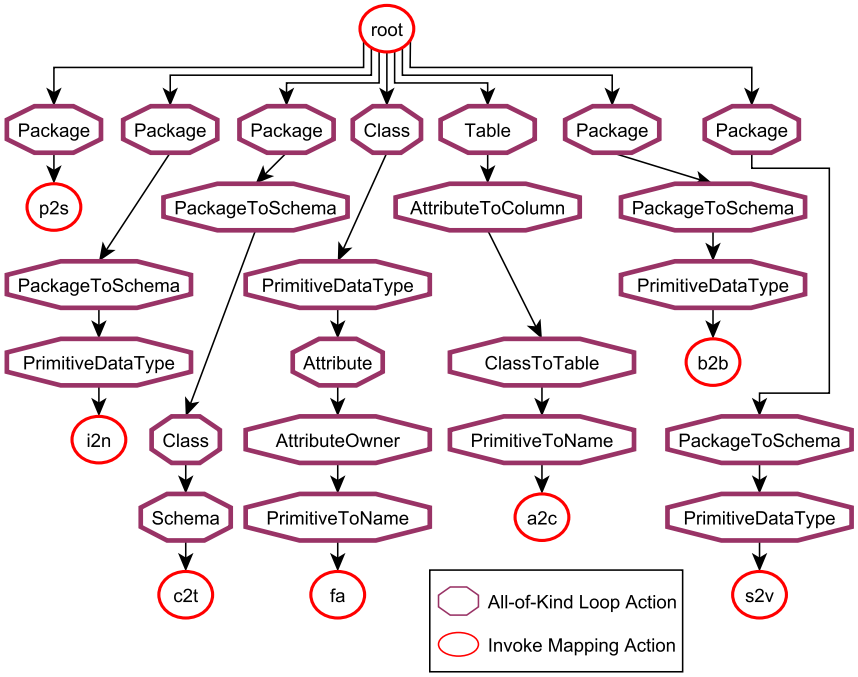


Figure 4.5: A non–thorough execution plan for the UML2RDBMS example.

of the transformation. Both mappings generate `PrimitiveToName` elements that are never consumed. Since $m_{b2b} \in SPL(m_{fa})$ and $m_{s2v} \in SPL(m_{fa})$

Lemma 1: *Mapping partial order is a necessary, but not sufficient condition for an execution plan to result in correct transformation execution.*

Proof: *Consider the case in which* $m_a, m_b \in SPL(m_v)$ *(as in the previous example), then* $m_a \preccurlyeq m_v$ *and* $m_b \preccurlyeq m_v$ *and any execution plan in which the mappings appear in on of these orders is valid* $[m_a \dots m_v \dots m_b], [m_b \dots m_v \dots m_a], [m_a \dots m_b \dots m_v]$, $[m_b \dots m_a \dots m_v]$*. Not that however, the first two cases will not result in correct transformations because elements produces by* m_b *and* m_a *respectively would not have been consumed by* m_v*.* □

The term *surplus producer* is used hereafter to refer to a mapping that produces elements that are not consumed by its con-

sumers. That is, a mapping m is a surplus producer if there $\exists m_w \in CNS(m)$, such that m_w is not scheduled after m.

Definition 18 (Execution Plan Thoroughness). An execution plan is *thorough* if, for every rule in the transformation, all the created elements of a rule are consumed by the rule's consumers, i.e. $\forall m_w \in CNS(m)$, m_w is scheduled after m. ⊛

For example, in Fig. 4.5 an additional invocation to m_{fa} could be added at the end in order to make the plan thorough.

Definition 19 (Execution plan validity). An execution plan is *valid* if it respects the mapping partial order and is thorough. ⊛

This definition of execution plan validity overlaps with the correctness and completeness definitions formulated in Lauder et al. [70] for triple graph grammar execution algorithms. This definition, however, allows validation of plans (algorithms) for declarative, rule-base, transformation languages.

Partial order and the notion of thoroughness are directly related to the correctness of the transformation, and allow minimization of the number of times a mapping is scheduled and the number of times a mapping is invoked with an NA result. The optimal conceivable execution plan is one in which each mapping is only scheduled once. Execution plan validity allows us to validate if a given execution plan that is considered optimal (each mapping is scheduled the optimal number of times) is valid or not. In the general case, execution plan validity can be applied to any plan.

4.4 Mapping Invocation Optimization via Context Reuse

Partial order is related to minimization of mapping scheduling (number of times a mapping appears in the execution plan) in the

execution plan scheduling problem. Another aspect of the execution plan scheduling problem is to minimize the number of times each mapping is invoked (a mapping can be invoked multiple times for each time it is scheduled in the execution plan). The number of times a mapping is invoked, from a static analysis point of view, depends on the number of all-loop actions that are present in the invocation path that triggers the mapping execution. (see Definition 10).

All-loop actions iterate over all elements of a given type and as such determine how many times a mapping is invoked. As the number of all-loop actions in an invocation path $ip(m_s, m_t)$ increases, the number of times the target mapping m_t is invoked increases exponentially. This section discusses an optimization mechanism to minimize the number of all-loop actions in the invocation paths of an execution plan. Section 4.5 shows how additional data dependency analysis can be performed to minimize the number of invocations with a False result.

Recall from Definition 10, that the elements for ascription available in an invocation path depend on the source mapping of the invocation path and the all-loop actions in the path. That is, the invocation path defines the context scope available for invocation of a mapping. Thus, for mapping $m_t = (T, V, S)$, and invocation path (from Definition 9) $ip(m_s, m_t) = \{I(m_s), c_1, A(t_1), c_2, \ldots, A(t_k), c_k, I(m_t)\}$, $k \leq |\{v \in V : type(v) \in IN(m)\}|$, that is, the number of all-loops can be lower than the number of input variables. Thus, it is potentially possible to further optimize an execution plans by using invocation paths that have a path source m_s that minimizes k, that is, most elements for ascription can be obtained by the elements generated by m_s.

One possible envisioned result of this optimization applied to the execution plan of Fig. 4.5, is presented in Fig. 4.6. In order to enable comparison, the figure preserves the execution order of mappings. The improved execution plan has gone from 23 to 18 all-loop actions.

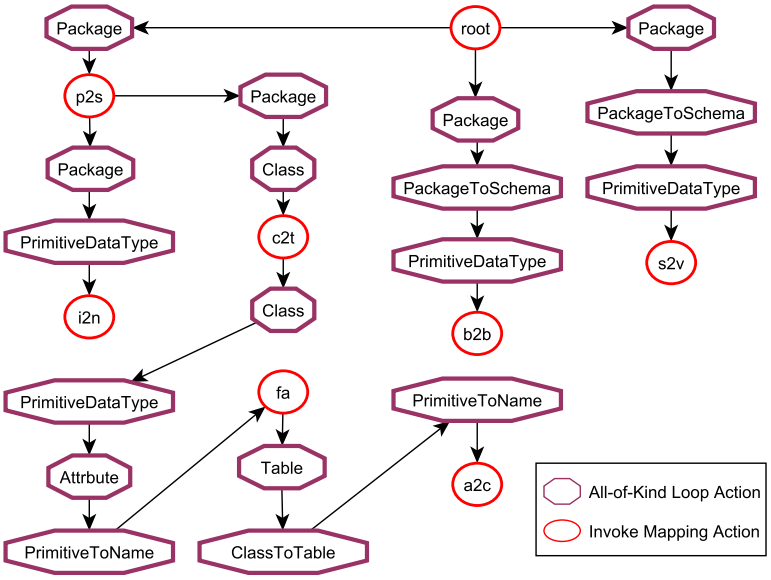


Figure 4.6: Context reuse in the execution plan.

4.4.1 Context Reuse Limitations

Context reuse is desirable as it allows construction plans that with invocation paths that use fewer all-loop actions. However, context reuse is not always feasible. Consider the case in which the UML metamodel from the running example is extended to support `Associations`. The additional classes and properties added to the original metamodels are presented in Fig. 4.7. The new mapping required to transform associations is presented in Listing 4.3. Mapping m_{c2t} is included for easy referencing. The mapping statements have been omitted as intra-mapping dependence analysis only uses the IN and OUT sets. Since the `Association` has a *source* and *destination* class, it is necessary that both classes have already been transformed and hence, the mapping has two `ClassToTable` and two `Table` input variables with their corresponding trace model classes.

Initially, since $m_{c2t}\delta^{\tau}_{ClassToTable}m_{a2fk}$, $m_{c2t}\delta^{\tau}_{ClassToTable}m_{a2fk}$ and $m_{c2t} \preccurlyeq m_{a2fk}$, m_{c2t} is a potential source for an invocation path to m_{a2fk}. However, notice that m_{c2t} only generates one element of each of these types (lines 114 and 118). Thus, the same element

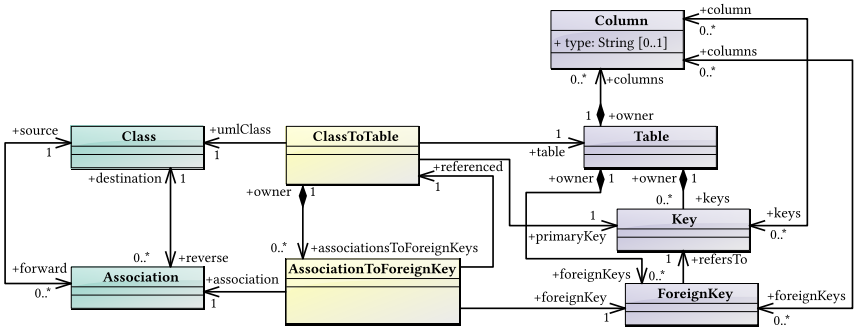


Figure 4.7: Additional classes to transform Associations.

would be used for ascription of the two variables in m_{a2fk}. That is, $c2t \rightarrow sc2t$ and $c2t \rightarrow dc2t$ (the same would happen with the table). Hence, such a reuse will only work to transform associations with the same source and destination class.

Listing 4.3: Mapping associationToForeignKey in UML2RDBMS.

```
110 map c2t in umlRdbms {
111     uml(p:Package, c:Class |
112         ...) { }
113     check enforce rdbms(s:Schema |) {
114         realize t:Table |
115         ...}
116     where(p2s:PackageToSchema |
117         p2s.umlPackage = p; p2s.schema = s;) {
118         realize c2t:ClassToTable |
119         ...}
120     ...
121 }
122 /** associationToForeignKey */
123 map a2fk in umlRdbms {
124     uml(p:Package, sc:Class, dc:Class, a:Association |
125       sc.kind = 'persistent'; dc.kind = 'persistent';
126       sc.namespace = p; a.source := sc; a.destination := dc;
127       a.namespace := p;) { }
128     check enforce rdbms(s:Schema, st:Table, dt:Table, rk:Key |
129       st.schema = s; rk.owner = dt; rk.kind = 'primary';) {
130         ...
131     }
132     where(p2s:PackageToSchema, sc2t:ClassToTable, dc2t:ClassToTable |
133       sc2t.owner = p2s; p2s.umlPackage = p; p2s.schema = s;
134       sc2t.table = st; dc2t.table = dt; sc2t.umlClass = sc;
135       dc2t.umlClass = dc;) {
136         ...
137     }
138     ...
```

```
139 }
```

Using the same element for ascription of two variables can cause invocations to result in NA. Hence, when optimizing an invocation path $ip(m_s, m_t)$ via context reuse the optimization needs to guarantee that elements generated from m_s are only ascribed to one variable. As a result, all-loop actions that provide elements of the types in $OUT(m_t)$ with multiple input variables in m_s cannot be removed from the path.

Additionally, although the types from the source metamodel are assumed to be available at the root (see Sect. 4.2.4) this does not imply that the root generates elements of these types. As such, the context reuse optimizations can't be applied to invocation paths that start at the root. That is, invocations paths $ip(m_{root}, m_i)$ will use all-loop actions for each of the arguments σ in $m^{\dagger}(\sigma)$.

Execution context reuse is a key aspect when minimizing the number of all-loop actions required for an invocation action in an execution plan and as a result minimize the number of times a mapping is invoked.

4.5 Element Ascription Optimization using Intra-Mapping Data Dependency Analysis

So far, this chapter has illustrated how the information from the mappings' IN and OUT sets is used to determine the data dependence relationships between mappings. This section looks at the information available in the mappings' statements, to identify intra-mapping relations. Intra-mapping relations are relations that exist between the mapping's variables and that can be used to further minimize the number of mapping invocations. The derivation results in the need for less loops in order to ascribe the mapping's variables.

The basic principles for the ideas explored in this section where the result of joint discussions with Dr. Edward Willink. A parallel research on the concept of derivation was later introduced by Dr. Willink [113]. The similarities are mainly in the identification of one of more variables can be identified as *primary* variables and that other variables in the mapping can be derived from them.

Recall from Definition 5 that in enforce mode a mapping $m = (T, V, S)$, can be applied to Σ_m, where Σ_m isthe Cartesian product of the elements that are consumed by the mapping. This section shows that intra-mapping relations can be used to reduce the number of sets and hence further reduce the number of required all-loop actions and minimize the number of times each mapping is invoked. This optimization reduces the time required to compute Σ_m and the number of iterations for a mapping invocation. Further, this section shows that the analysis used for this optimization minimizes the invocations with a False result.

As presented in Sect. 2.2.2, a valid binding of a pattern is a binding where all the variables of the pattern are bound to a value other than undefined, and where all the predicates of the pattern evaluate to true. Thus, the most efficient execution plan is one in which mappings are only invoked on element tuples that would result in valid bindings. Let's consider mapping m_{fa} in the UML2RDBMS example, presented again in Listing 4.4 (partial). The predicates in line 76 restrict the `Attribute` to be related to the `Class` via the *owner* property, and the `Attribute` to be related to the `PrimitiveDataType` via the *type* property. That is, a valid binding is one in which the elements assigned to c, t, a, fao and $p2n$ fulfil these conditions. Thus, for the example models in Fig. 3.3, tuples $\{C4 : \text{Class}, A3 : \text{Attribute}, Pr1 : \text{PrimitiveDataType}\}$ and $\{C4 : \text{Class}, A1 : \text{Attribute}, Pr3 : \text{PrimitiveDataType}\}$ do not fulfil the predicates of the UML domain and thus are not suitable for a valid binding. As a result, the mapping invocation for these two tuples would be False.

Listing 4.4: Mapping fromAttribute in UML2RDBMS.

```
73 /** fromAttribute */
74 map m6 in umlRdbms {
75     uml(c:Class, t:PrimitiveDataType, a:Attribute |
76         a.owner = c; a.type = t; ) { }
77     where(fao:AttributeOwner, p2n:PrimitiveToName |
78         fao.attOwner = c; p2n.primitive = t;) {
79         ...
```

This section argues that by considering predicates as *selection conditions* rather than filtering conditions, generation of tuples that result in False invocations can be pre–empted altogether. This concept stems from the fact that predicate `a.owner = c` can be read backwards: `c = a.owner`. Thus, instead of filtering the tuples to find the ones in which `c` matches the condition, the tuples can be constructed by picking an `a` and evaluating `a.owner` to select a `c` that will result in a tuple with a True invocation result. Hereafter, `a` acts as the *primary* variable and `c` as the *derived* variable. In this research, the process of identifying these relations is called variable *derivation*.

It is important to note that QVTc predicates are Object Constraint Language (OCL) expressions and as such, it is possible to construct expressions of much higher complexity than the form `<var>.<property>=<var>` used in the previous example. As the complexity of the expression increases (number of operators, variables, operation invocations, etc.) it is harder to identify relations between variables that can be used as selection conditions. This research limits the analysis to predicates of the form

```
<var>.<property>=<var>
```

given that it is the most common form of predicate observed in the development transformations (see Sect. 6.1 for a discussion on this limitation).

Example 9. For the mapping m_{fa} in Listing 4.4 and the candidate models in Fig. 3.3, $\Sigma_{m_{fa}}$ will consist of $3(Classes) \times 3(PrimitiveDataTypes) \times 4(Attributes) \times 3(AttributeOwner) \times$

$4(PrimitiveToName) = 432$ tuples. By using selection conditions this can be reduced to four, one per each attribute of each class. That means that $m^{\dagger}_{fa}(\sigma) = False$ in 428 of the cases. ◫

By using predicates as selection conditions, the ascription process can be optimized by reducing the number of tuples in σ and effectively minimizing the number of invocations with False results. Elimination of all invocations with False results is impossible as the static analysis cannot identify all possible relations between variables. Given that this analysis can be done statically, it does not affect the runtime performance.

Additionally, this research assumes that the candidate models and trace model are Meta Object Facility (MOF) [4] models (or models that exhibit the same characteristics, e.g. EMF [95] models). Thus, an element property may have an *opposite property*. An opposite property allows properties to be evaluated in the opposite direction of the property. That is, if Class A has a property *child* of type B, and Class B has a property *parent* that is the opposite of *A.child*, then `a.child.parent = a`.

Opposite properties can be used to reverse the QVTc predicate as part of the analysis. For example, the *owner* property of an attribute has an opposite property called *attributes* then the predicate `a.owner = c` could be rewritten as `c.attributes->includes(a)`. For opposite properties with multiplicity greater than 1 this rewrite is not useful. However, if the multiplicity was one, then the predicate can be rewritten as `c.attributes = a`. In this form, the predicate can be used as a selection condition too. This rewrite broadens the possibilities for variables that can be used as primary variables. Selecting which variables to use as primary variables is discussed next.

4.5.1 Intra-Mapping Data Dependence Relations

This section introduces the concepts of property dependence relations and property dependence graph for a Model Transformation Program (MTP) written in a QVTc. The definitions take into consideration that the QVTc execution engine provides a functionality in which missing opposite properties are added to the model automatically. Thus, it is assumed that an opposite relation will always exist. Further, although only single–valued properties are of interest, multi–valued dependencies are also shown (mainly in the opposite case) as they are useful for further optimizations (see Sect. 4.5.2). These concepts facilitate the analysis of derivations.

Definition 20 (Property Dependence Relations). Given mapping $m = (T, V, S)$ and two variables $v_s, v_t \in V$, the following property relations may hold or the variables may be independent.

1. If some predicate $s \in S$ and s is of the form $v_s.p = v_t$, then v_t is *direct-property dependent* on v_s and denote this by $v_s \vec{\pi}_p v_t$.

2. If some predicate $s \in S$ and s is of the form $v_s.p = v_t$, then v_s is *opposite-property dependent* on v_t and denote this by $v_s \overleftarrow{\pi}_p v_t$. ⊛

Definition 21 (Indirect Property Dependence). Variable v_s is *property dependent* on v_t, denoted $v_s \pi_p v_t$, if $v_s \vec{\pi}_p v_t$ or $v_s \overleftarrow{\pi}_p v_t$. Variable v_s is *indirectly property dependent* on v_t, denoted $v_s \Pi v_t$, if there are variables $v_{k_1}, \ldots, v_{k_n}, n \geq 0$, such that $v_s \pi_{p_i} v_{k_1} \ldots v_{k_n} \pi_{p_j} v_t$ and all properties $p_i \ldots p_j$ are single–valued. ⊛

Definition 22 (Property Dependence Graph). All property dependences in a mapping with n variables can be represented by a *property dependence graph* **PDG** of n nodes one for

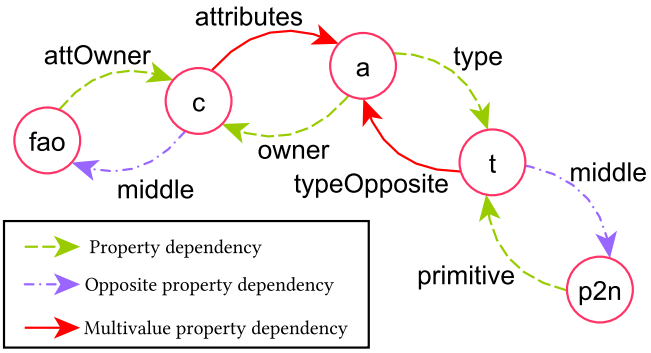


Figure 4.8: Property Dependence Graph for m_{fa} in UML2RDBMS example.

each v_i $(1 \leq i \leq n)$. For each $\overrightarrow{\pi}_p$ and $\overleftarrow{\pi}_p$ between v_l and v_k there is a corresponding edge in **PDG** from the node representing v_l to the node representing v_k. ⊛

Example 10. The property dependence graph for mapping m_{fa} in the UML2RDBMS example is shown in Fig. 4.8. The automatically created properties from the trace to the candidate models use the default naming scheme (*middle*, this name is configurable through annotations in the metamodels) The property dependencies are as follows:

$$
\begin{array}{ll}
a\overrightarrow{\pi}_{owner}c & c\overleftarrow{\pi}_{attributes}a \\
fao\overrightarrow{\pi}_{attOwner}c & c\overleftarrow{\pi}_{middle}fao \\
t\overrightarrow{\pi}_{type}a & a\overleftarrow{\pi}_{typeOpposite}t \\
p2n\overrightarrow{\pi}_{primitive}t & t\overleftarrow{\pi}_{primitive}p2n \\
a\Pi fao & fao\Pi a \\
t\Pi fao & p2n\Pi fao \\
a\Pi p2n & p2n\Pi fao
\end{array}
$$

◫

The property dependence graph (PDG) can be used to find all derivations by selecting each of the input variables as the primary

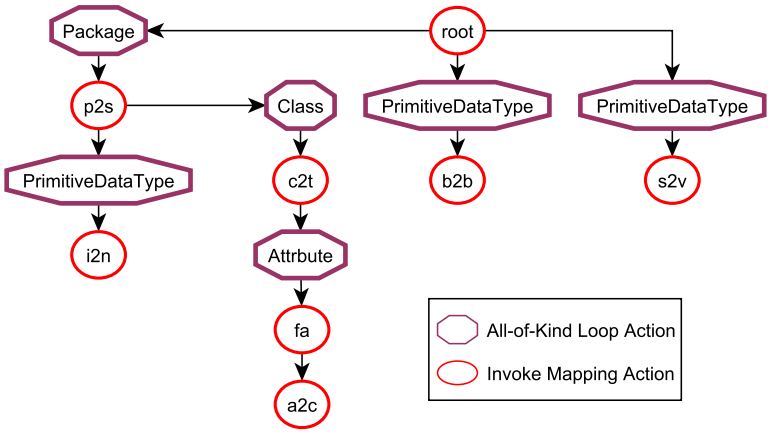


Figure 4.9: Context reuse in the execution plan.

variable. Derivations will allow us to further reduce the number of kind actions needed in an execution plan.

Example 11. Figure 4.9 presents the result of applying derivation optimizations to the plan presented in Fig. 4.6. For example, for the execution path $ip(m_{c2t}, m_{fa})$, the derivation can be done by selecting variable *fao* as the primary variable. An all-loop action for `Attribute` elements is still needed because the *attributes* property of `Class` is multi-valued. Depending on the invocation path there can be more alternatives to use as the primary variable. Selection of the alternative that provides the optimal performance is explained in detail in the synthesis chapter, Sect. 5.2.1. In the figure the number of all-loop actions has been reduced from 18 to 6. ◫

4.5.2 All-Of-Kind Action Refinement

For derivation, multi-valued properties (either direct or opposite) are not useful. However, these properties can be used to provide additional optimization. Consider mapping m_{fa} in the UML2RDBMS transformation, and the predicate `a.owner = c` rewritten as `c.attributes->includes(a)`. Although variable `a` cannot be derived from variable `c`, notice that the mapping is

only interested in attributes that match this condition. Hence, if `c` is selected as the primary variable, the `Attribute` all-loop action does not need to loop over all existing attributes, just the ones contained by `c`. In other words, multivalued properties can be used to provide efficient (less iterations) all-loop actions. A new type of action is added to the control component model to represent loops over attributes of a primary variable: *multi-valued-loop* action (multivalue for short).

Intra-mapping data dependence relations are key to minimize the number of all-loop actions because all-loop actions are no longer always needed for secondary variables that cannot be directly derived from the primary variable. Additionally, it is also key to minimize the number of invocations with a *False* result given that derivation will ensure that elements in a tuple are more likely to satisfy the mapping guards.

4.6 Summary

This chapter has described the synthesis problem as a scheduling problem and defined the four minimization targets that define the problem. It then showed how the knowledge embedded in the logic component can be used to provide the definitions and conditions that will enable these targets to be achieved during construction of solutions to the problem, the execution plans. The chapter identified key restrictions on the structure of the execution plan with regard to placement of invocation actions. The chapter showed how data dependency analysis between variables in a mapping can be used to minimize the number of mapping invocations. The chapter also discussed the issue of loops in the dependency graph and concluded that the scheduling problem can't be solved for all QVTc transformations. The next chapter uses the definitions and conditions introduced in this chapter to construct the algorithm that can construct an execution plan that is a valid solution to the scheduling problem.

5

Systematic Execution Plan Synthesis

The last chapter argued that the execution plan synthesis problem is a scheduling problem, and showed how data dependence analysis can be used to provide definitions and constraints to assess the validity of synthesized execution plans. These definitions and constraints are important as they can be used to guide the synthesis process and/or to validate synthesized execution plans. That is, the enable the systematic synthesis of execution plans.

Given the size of the solution space and the similarity of the execution plan synthesis problem to the instruction scheduling problem, the last chapter also argued against the use of an exhaustive $\mathcal{NP}$ scheduling algorithm to solve the execution plan synthesis problem. As an alternative the use of a meta-heuristic algorithm was proposed. A meta-heuristic algorithm can produce potentially sub-optimal solutions, but the solutions are found within a realistic timeframe. This chapter presents how the $\mathcal{MAX} - \mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) [98] metaheuristic algorithm can be used to solve the execution plan synthesis problem. The chapter shows that the execution plan synthesis problem can be solved as a combinatorial optimization problem and then presents the observed

results of the proposed algorithm in a set of development transformation examples.

Section 5.1 describes a *base execution plan* that can be used as an oracle to validate execution plans constructed using the systematic approach. Following, Sect. 5.2 argues for the use of the Ant Colony Optimization (ACO) meta-heuristic to solve the execution plan synthesis problem and formally defines the execution plan synthesis problem as a combinatorial optimization problem. Section 5.3 presents how the ACO meta-heuristic can be used in this particular scenario and Sect. 5.4 shows how the definitions from Chap. 4 are used to construct feasible neighbourhoods, a key aspect of the ACO meta-heuristic. Section 5.6 reports on the results of the synthesis algorithm on the set of examples used for development and tuning of the algorithm. Finally, Sect. 5.7 summarizes and concludes the discussion.

5.1 A Base Control Component

No other implementations of the QVT Core [3] (QVTc) language were found during the literature review of this research. Hence, this section presents the algorithm that will act as a base control component to validate the correctness of synthesized control components and as the benchmark for performance evaluation. Target models generated by executing the synthesized execution plans can be compared against target models generated using the base control component. Performance of the synthesized execution plans is expected to be better than the performance of the base control component. This algorithm is labelled as *naïve*, in the sense that it does not use data dependency analysis to define the execution order of mappings. The lack of ordering is key to assess if the schedule of synthesized execution plans results in better performance, which in this research is gauged by measuring the execution time of the transformation. Since this research is only interested in evaluating the scheduling of mappings, the base

control component uses intra-mapping data dependence analysis to reduce the number of all-instances loops. Hereafter, the term *naïve plan* is used to refer to the execution plan constructed with this algorithm.

Definition 8 discussed that all possible permutations of elements must be considered for ascription. These elements include any elements that exist in the candidate models at the beginning of the execution and any elements that are created during execution. The implication of having to consider elements that are created during execution is that given a mapping m, Σ_m cannot be calculated definitely before execution. A workaround is to calculate Σ_m just before invoking the mapping. However, assuming mappings are executed sequentially, $m_1 \rightarrow m_2 \rightarrow \cdots \rightarrow m_n$, this still poses the issue that if execution of m_i creates some elements that must be included in Σ_{m_j}, with $j < i$, then they would not have been considered. To solve this, all mappings have to be invoked multiple times, which guarantees that new elements will be considered for ascription in future invocations of a mapping. In this case, a terminating condition must be defined in order to guarantee that the algorithm eventually finishes.

The implementation of the QVTc virtual machine used in this research only supports enforcement at the mapping level. When a mapping is invoked all the statements are executed in enforcement mode. That means, all realized variables will create new elements in the target/trace model and assignments will modify the attributes of these new elements. As a result, if a rule is invoked twice with the same arguments, duplicate elements will be created in the target/trace model. For example if mapping m_p2s (see Listing 3.2) was invoked for `P1:Package` in the example model (see Fig. 3.3) then two `S1:Schema` instances would be created in the target model. To avoid this behaviour the algorithm uses a cache mechanism to remember all invocations with a True or False result. Invocations with NA results will be reattempted in later iterations. This implementation detail limits the applicability of

the engine as discussed in Chap. 7.

This implementation detail also implies that element instantiation is the only change we can track as a flag that one or more mappings have executed with a True result. Element instantiation is important because it signals that mappings that previously resulted in NA invocations might evaluate to True or False if invoked again. Thus, the terminating condition can be defined as the lack of change (element creation) in the target and trace models. If no new elements are created, there is no point in re-invoking mappings that have previously resulted in NA invocations. The terminating condition should only be tested after all mappings have been invoked at least once.

Algorithm 1 Naïve declarative transformation execution

```
1:  procedure EXECUTE(M)
2:      repeat
3:          forEach m in M do
4:              forEach σ in Σ_m do
5:                  if NOTEXECUTED(m, σ) then
6:                      if m†(σ)≠ NA then
7:                          HASEXECUTED(m, σ)
8:                      end if
9:                  end if
10:             end for
11:         end for
12:     until no changes in candidate models
13: end procedure
```

Algorithm 1 presents an algorithm that results in correct transformation execution as per Definition 7 and takes into consideration the requirements discussed previously. Procedure `notExecuted` (line 5) tests if the mapping has been previously invoked with a given σ. If $m^{\dagger}(\sigma) \neq \text{NA}$, procedure `hasExecuted` caches the information to avoid repeated invocations (`notExecuted` uses the information cached by `hasExecuted`). Note that procedures `notExecuted` and `hasExecuted` can be implemented by the transformation engine as part of the mapping invocation functionality. Hence, no additional execution actions are needed in order to represent the naive control algorithm as an execution plan. They are included in the algorithm for completeness[1].

[1]This additional functionality was introduced to the implementation used

The outer loop (line 2) results in all mappings being invoked as many times as necessary. Note also that at each iteration mappings can be invoked in any order. We assume that the terminating condition is implemented in the execution engine and hence it is not part of the execution plan. The reasons for this algorithm to correctly execute a Model Transformation Program (MTP) written in a QVTc are as follows:

1. All possible invocation orders are considered: execution can start at any mapping and all mappings are visited (line 3).

2. All mappings are invoked as many times as necessary (line 2).

3. All possible combinations of candidate model elements are used for ascription. New elements created by a mapping execution will be available for ascription in the next iteration (line 4).

4. Successful invocations, $m(\sigma) \neq \text{NA}$, are cached to avoid duplicate element creation.

For the construction of the *naïve plan* the mappings will be always visited in the same order for each iteration of the outer loop, however the order of the rules is defined randomly during construction.

5.2 The Execution Plan Synthesis Problem as a Combinatorial Optimization

The Ant System is a cooperative heuristics searching algorithm in which the agents replicate the behaviour of ants, originally introduced by Dorigo et al. [34]. Agents work individually to construct

in this research

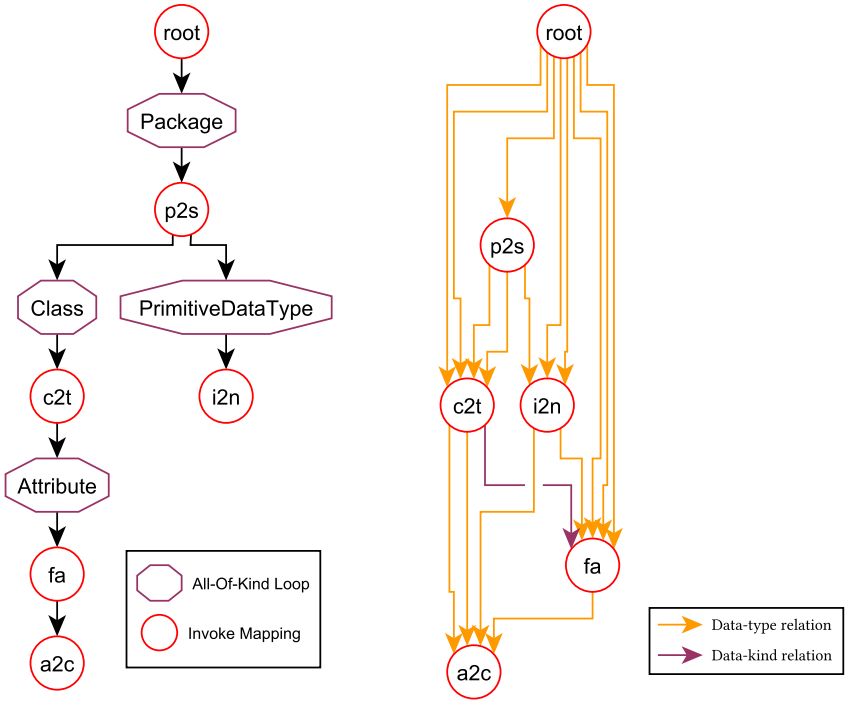


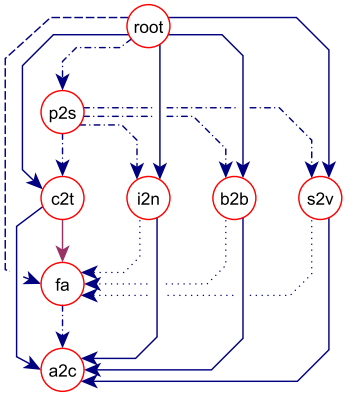
Figure 5.1: An Execution Plan and a Data Dependence Graph side by side.

solutions to the problem by exploring the solution space randomly. However, they share information about the solutions they find (in the form of pheromones) allowing a collective knowledge of the best solutions to emerge.

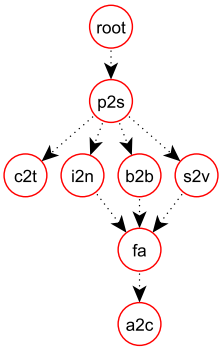
The ACO meta-heuristic is a metaheuristic for solving hard combinatorial optimization problems [35] and was chosen because it has been used with promising results in instruction scheduling optimizations [109, 108].

Figure 5.1 presents the execution plan and the data dependence graph (DDG) for the UML2RDBMS example side-by-side (taken from Fig. 4.9 and Fig. 4.4 respectively). The information for mappings m_b2b and m_s2v and the type information in the DDG have been removed in order to aid readability. The figure indicates a strong relation between the structure of the execution plan and the structure of the DDG.

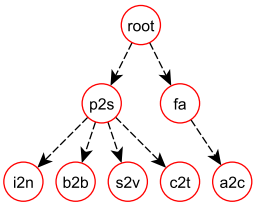
To further understand this relation, consider the DDG of Fig.



(a) DDG



(b) Execution Plan A



(c) Execution Plan B

Figure 5.2: Execution Plans and their relation to the Data Dependency Graph.

5.2a and the execution plans of Fig. 5.2b and Fig. 5.2c. The all-loop nodes in the execution plan and the type information in the DDG have been removed to aid readability. Since all–loop actions can only have one incoming and one outgoing edge this representation of the execution plan respects the structure and consequently the depth first search traversal order, and hence the execution order, of the original execution plan. Notice that both execution plans can be constructed by using a subset of the edges of the DDG. Execution plan A uses the dotted edges and execution plan B the dashed. Common edges are represented in Fig. 5.2a with a dot–dash line.

In fact, an execution plan (sans the loop actions) will have the same structure as a subgraph of the DDG. The reason for this is that invocations paths in the execution plan are always defined based on the dependencies (edges) of the DDG. A result of this fact is that construction of the execution plan can be understood as the construction of spanning trees in the DDG. Further, since execution plan synthesis problem is a minimization problem, it is in fact a minimum spanning tree (arborescence) problem on the DDG. This type of problem is a combinatorial optimization problem and has been solved for other type of problems using the ACO metaheuristic [71, 19].

In order to solve a combinatorial optimization problem with the ACO algorithm, it must be modelled in the form of $P = (S, \Omega, h)$ [35] where S is the search space defined by a finite ordered set of decision variables(hereafter called S-variables to distinguish them from QVTc variables), each with a finite domain (see Sect. 5.2.1), Ω is a set of constraints among the S-variables (defined next), and $h : S \longmapsto \mathbb{R}_0^+$ is the objective function to be minimized (see Sect. 5.2.3). "A feasible solution $s \in S$ is an assignment to each S-variable of a value in its domain such that all the problem constraints in Ω are satisfied" [35]. A feasible solution $s^* \in S$ is called a global minimum of P if and only if we have that $h(s^*) \leq h(s) \forall s \in S$, and $S^* \subseteq S$ denotes the set of all global minima.

Section 4.1 discussed that the execution plan synthesis problem is a scheduling problem in which a mapping execution order has to be found. Without loss of generality the execution plan synthesis problem can be redefined as a scheduling problem in which a mapping invocation order has to be found. That is, a schedule for the invocation paths has to be found. For a transformation $Q = (O, M)$, there are $|M^2| + |M|$ domain variables, one for each possible path between all pair of mappings and one for each path between the root mapping and all the other mappings. Since a execution plan does not require that a mapping invoke

all other mappings, then a domain variable can either be assigned an $ip \in IP$ or null. By considering that the variable index represents the schedule order, then an instantiation of all variables is an execution plan. Ω can be defined based on Definition 19, that is, a solution $s \in S$ is feasible if the execution plan formed by the decision variables with a value different than null is a *valid* plan.

5.2.1 The Invocation Graph

This section introduces the concept of *Invocation Graph* as an alternative representation of the execution plan to be used for the ACO algorithm, in order to provide a reduced domain for the decision variables and to provide weight information that can be used by the objective function.. The *Invocation Graph* is a weighted rooted graph. A *rooted graph* is a graph in which one node is labelled as the root of the graph. Traversal operations on a rooted graph (e.g. depth-first search) will always start at the root node. A *weighted graph* refers to an edge-weighted graph, that is, a graph where edges have weights or values.

Definition 23 (Invocation Graph). Given the set of all possible invocation paths IP, the *invocation graph* $IG = (M, IP', ic)$, is a weighted rooted graph that consists of finite sets M of mappings and $IP' \subseteq IP$ of directed edges, and a function $ic : ip \rightarrow \mathbb{R}$ that assigns each edge (invocation path) a real–valued weight. Mapping *root* is the designated root node. The domain of the S-variables in S is now IP'. ⊛

Section 4.5.1 showed that for a given invocation $I(m_j)$ the property dependence graph (PDG) provides different alternative ways to derive the invocation arguments. Hence, for a pair of mappings m_i, m_j there exists $ip_1(m_i, m_j), ip_2(m_i, m_j), \ldots, ip_k(m_i, m_j)$ different possible invocation paths between the mappings, where each ip_l uses a different variable derivation alternative.

Section 4.5 showed that by using derivation the number of all–loop actions in a path can be reduced by eliminating them or by

replacing them by multivalue actions that, arguably, will perform fewer iterations. Hence, a particular derivation may determine the optimal number of all-loop and multivalue actions for a given invocation path. This research argues that it is possible to statically compute an optimal derivation and therefore an optimal invocation path for a given pair of mappings. Hence, the IG is restricted to optimal invocation paths, represented by $IP' \subseteq IP$.

Since variable derivation does not affect the mapping ordering, nor is part of the additional validation conditions Ω, optimizing variable derivation can be done as a local optimization and prior to execution plan synthesis. Next sections show how the PDG can be used to find the optimal derivation and hence optimal invocation paths.

5.2.2 Property Dependence Graph Weights and Spanning Arborescence Cost

In order to find an optimal derivation, we must be able to quantitatively compare derivations. For this, we define a weighted PDG:

Definition 24 (Weighted Property Dependence Graph). Given a mapping $m = (T, V, S)$, a *weighted property dependence graph* $\mathbf{WPDG} = (V, P, ds, dt, w)$ is a PDG (from Definition 22) with additional functions $ds, dt : P \to V$ that assign each edge source and target nodes, and a function $w : P \to \mathbb{R}$ that assigns each edge a real-valued weight.

The weight function is defined as follows:

$$w(p) = \begin{cases} 1, & \text{if } p \text{ is explicit} \\ 5, & \text{if } p \text{ is implicit} \end{cases} \tag{5.1}$$

These weights are arbitrary but express a difference in the computation effort of the property derivation. Explicit properties are defined in the metamodels and can be directly computed. On the other hand, implicit properties are maintained by the execution engine and thus additional processing is required to store and retrieve their values. ⊛

The minimum derivation (MD) problem can be considered as an extension to the minimum–weight rooted spanning arborescence (MWRSA) [86, 71] problem. An arborescence is a directed graph in which, given a vertex v (labelled the root) and any other vertex u, there is exactly one directed path from the root to u. The main differences are that the weighted property dependence graph (WPDG) can be disconnected and that the node that provides the minimum arborescence has to be found (or nodes for the disconnected case). In order to solve the MD problem, we need to define the objective function to be minimized.

For this, this section introduces the concept of *Derivation Path*.

Definition 25 (Derivation Path). Given a **WPDG** and an arborescence R_v with $v \in V$, a *derivation path* $d = (p_1, p_2, \ldots, p_n)$ represents the property navigations required to derive a secondary variable $v_s = dt(p_n)$ from the primary variable $v_p = ds(p_1)$. The set of derivations paths of an arborescence is $D(R_v)$. A derivation path can have at most one multi-valued property.⊛

Section 4.5.2 showed that derivation from a multi-valued property involves creating an additional multi-valued-loop action in the invocation path. As a result, the variables after a multi-valued property must be derived from the iterator of the multivalue action, not from the primary variable.

The cost function value $c(d)$ of a derivation path d is defined as follows:

$$c(d) = 1 + \begin{cases} \sum_{p \in d} w(p) & \text{if no multi-valued property} \\ \sum_{p \in d'} w(p) + \sum_{p \in d''} w(p) & \text{if one multi-valued property} \end{cases} \tag{5.2}$$

where, given p_k is the multi-valued property, $d' = (p_1, p_2, \ldots, p_k), k < n$ and $d'' = (p_{k+1}, p_{k+2}, \ldots, p_n)$ The initial cost of one is to account for the assignment of the primary variable (from the context or an all-loop action iterator).

The MD problem can then be technically described as follows. Let $\mathcal{A}_v$ be the set of all arborescences in $WPDG$ rooted in variable v made up from derivation paths. In this context, an arborescence is a directed rooted tree in which all edges points away from the root variable and each variable can appear only once, and where there is at most one multi-valued property dependency edge. By verifying that variables can appear only once, loops in the $WPDG$ can be broken. The objective function value (that is, the cost) of an arborescence $R_v \in \mathcal{A}_v$ is defined as follows:

$$f(R_v) := \sum_{d \in D(R_v)} c(d) \tag{5.3}$$

The goal of the MD problem is to find a variable v' and an arborescence $R^*_{v'} \in \mathcal{A}_{v'}$ such that the cost of $R^*_{v'}$ is smaller than or equal to the cost of all other arborescences in $\mathcal{A}_{v'}$ and of all other arborescences in $\mathcal{A}_{v_i} \, \forall \, v_i \in V$. If the the WPDG is disconnected, we can find the optimal arborescence for each sub-graph given that derivations in sub-graphs are independent of each other.

Example 12. Figure 5.3 presents the weighted PDG for $m_f a$ in the UML2RDBMS example, which complements the PDG in Fig. 4.8. If variable $p2n$ is selected as the root, then t can be derived via *primitive* with a cost of 1. Next, a can be derived via *primitive.typeOpposite* with a cost of $1 + 1 = 2$. For c, the *typeOpposite* property is multi-valued, which results in a representing the iterator of multivalue action `for a in t.typeOpposite`. Hence, c is derived from a via *owner* with a cost of 1. And in a similar way, fao will be derived via *owner.middle* with a cost of $1 + 5 = 6$ (middle is implicit). The total derivation cost will be 11 (1+2+1+6 +1). Table 5.1 presents the MD for mapping $m_f a$. Each arborescence is represented by its edges $(source, target)$ and an $^+$ is used to indicate that an implicit iterator is used. It is important to note that as seen in Table 5.1 it is possible for two or more MD to have the same derivation cost. If multiple MD with the same cost are

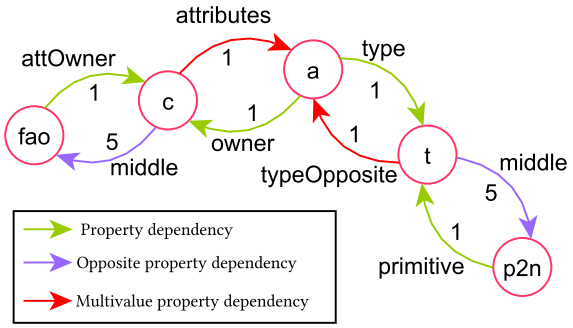


Figure 5.3: Weighted Property Dependence Graph for m_{fa} in the UML2RDBMS example.

Table 5.1: Cost of the arborescences for mapping m_{fa} in the UML2RDBMS example.

Root variable(s)	Arborescence	Cost
fao	$\{(fao, c), (c, a), (a^+, t), (t, p2n)\}$	11
c	$\{(c, fao), (c, a), (a^+, t), (t, p2n)\}$	14
a	$\{(a, c), (c, fao), (a, t), (t, p2n)\}$	15
t	$\{(t, p2n), (t, a), (a^+, c), (c, fao)\}$	14
p2n	$\{(p2n, t), (t, a), (a^+, c), (c, fao)\}$	11

available for a given invocation path, then one can be picked randomly. ◫

However, the derivation information alone is not enough to determine the cost of an invocation path. Consider the case of invocations from the root. In this case, an additional all-loop action over all instances of the primary variable's type is needed. In general, an all-loop action over all instances of the primary variable's type is needed for each of the minimum derivations of the WPDG, unless the primary variable can be assigned a value from the context.

Additionally, the derivation process must be repeated for each iteration of the primary variable and for each iteration of the implicit iterators. Intuitively, the cost for an invocation path $ip(m_s, m_t)$ (Definition 9)is then of the form $ic(ip) = f(R^*_v) \times (L + N)$, where $v \in V_{m_t}$, where R^* is the minimum derivation

of IP, L is a factor associated to the number of all-loop actions over all instances and N is a factor associated to the number of multivalue actions. Next, in order to understand the performance relations of the all-loop actions and the derivations, the following section shows how the ic function was derived empirically based on the minimum derivations and the observed performance of the different alternatives.

5.2.3 Deriving the Invocation Cost Formula

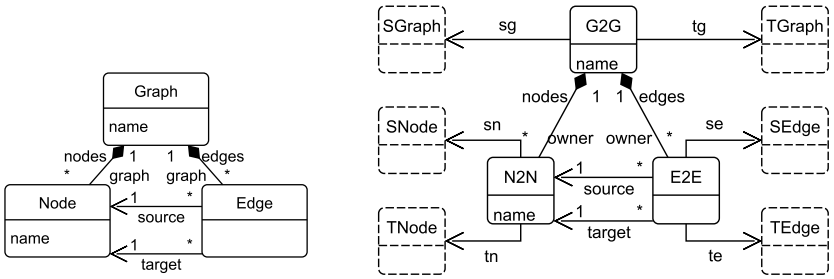
This section describes how the invocation cost formula was derived.

> The experience with relational system has shown that the main purpose of a cost model is to differentiate between good and bad executions, in fact, it is known, from the [experience with relational system], that even an inexact cost model can achieve this goal reasonably well [66].

From this, the cost function used in this research has been derived intuitively from observation. The derivation is done with the objective of differentiating good and bad executions.

For the description of the empirical analysis this section introduces a simpler transformation which creates a copy of a graph. There are two main reasons for the use of a simpler transformation. First, in order to argue for the validity of the cost function all possible execution plans for the transformation should be constructed and evaluated. Although it is a time–consuming task and takes a considerable effort, with a simpler transformation the complete solution space could be constructed in a reasonable amount of time. Second, to analyse the DDG and relate its structure to the observed performance results it is desirable to have a DDG of a manageable size.

The metamodels for this example are presented in Fig. 5.4 and the PDG in Fig. 5.5. Mapping *g2g* transforms a graph into



(a) Source/Target Metamodel

(b) Trace Metamodel

Figure 5.4: Source/Target and Trace metamodels for the Graph to Graph example.

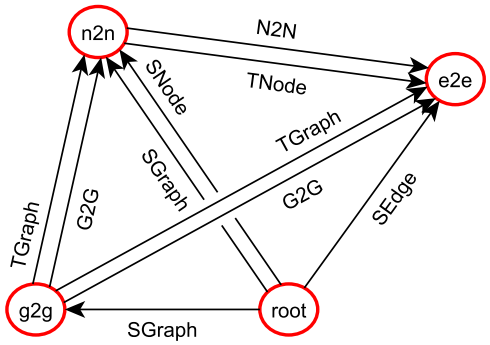


Figure 5.5: *PDG* for the Graph to Graph example.

a graph, and *n2n* and *e2e* do the same respectively for nodes and edges. For this transformation the mapping partial order is given by $g2g \preccurlyeq n2n$, $g2g \preccurlyeq e2e$ and $n2n \preccurlyeq e2e$. A brute force approach was used to construct the complete solution space for the graph transformation. From this, the valid execution plans were selected manually by inspection of their structure. The valid execution plans include solutions with three distinct invocation structures and with a wide range of derivation options. In total, 240 valid execution plans were found.

To compare the performance of the different plans a model with approximately 100k elements was randomly generated. The size of the model was chosen so the effect of loops and the invoca-

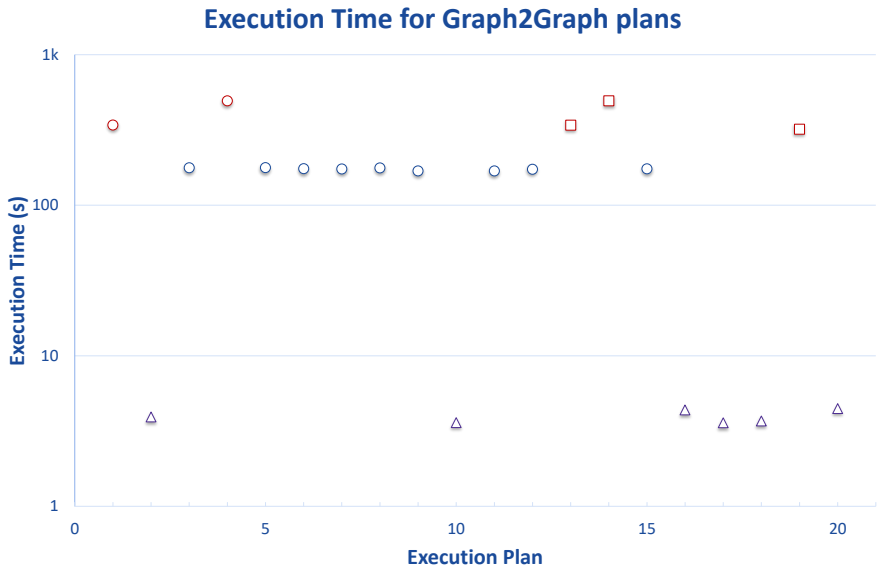


Figure 5.6: Execution times for the 20 execution plans sample.(Fixed Y-axis)

tion path size (number of loops) can be observed. The model was generated using a semi-automated model generation strategy developed using the EMG tool [83]. If all-loop actions have a number of iterations much smaller than multivalue actions or vice versa, then it is possible that performance of plans with more of one type of action than the other exhibit better performance related to the number of iterations and not to the derivations. To avoid this, the generation script is designed to have a balanced number of elements of each type in the source metamodel. As a result, the source model has multiple graphs ($|G| = 233$) and each graph in turn has $|N| + |E| = 466$ elements. Although an all-loop over `Graph` elements will have fewer iterations than an all-loop over `Node` elements, it will have similar iterations as a multivalue loop over `Graph.nodes`. Initially a sample of 20 plans was taken from the total 240. Each plan was executed 10 times using the 100k-element model as the source model. The results are presented in Fig. 5.6.

The plot shows that three distinct subgroups of plans that have similar execution times can be identified. These groups are

highlighted in the figure by using different marker shapes. There is a group that exhibits execution times around 5 seconds (triangle), another around 170 seconds (circle) and the third one around 500 seconds (square). Each of the subgroups was analysed, taking into consideration the structure of the plans, number of kind actions and derivations used. It was found that the performance results exhibit a proportional relation between the binding cost and the number of all-loop actions, i.e. $f(R^*) \times L$. Further, it was also observed that for multivalued actions the effect was low for one or two actions but increased exponentially for three or more. Finally, the cost of implicit property access was adjusted as the results indicate that the difference is not as big as initially assumed (5 fold). The adjusted value used for implicit properties is 1.2. The final empirical formula for the invocation cost that better fitted the observed results was(replaced T for R):

$$ic(ip) = f(R_v^*) \times L + f(R_v^*)^N \tag{5.4}$$

Since the number of all-loop actions (L) and multivalue actions (N) can be known from the derivation information, it is possible to calculate the cost of any of the 240 valid execution plans of the graph transformation.

To evaluate the cost formula, a new sample of 20 execution plans was used. The cost for each of the execution plans was calculated and each of them was executed with the 100k-element model. Each execution plan was again executed 10 times. The results are shown in Fig. 5.7, were the execution plans in the x-axis are sorted according to cost. The cost of each execution plan (5.4) is shown above the box plot.

The graph reveals that performance can be used to group the execution plans in three distinct groups. One group with an execution time of around 4 seconds ($M = 3884, SD = 255$), another group with an execution time of around 220 seconds ($M = 226794, SD = 22240$) and the final group (with only one plan) with an execution time of around 600 seconds. The average median of each group is lower than the third quantile of the next

group, so it can be concluded that in general, the performance of each of the groups is better than the next group.

It is also important to note that all the execution plans in each group have similar cost(5.4). For the 4 second group: $(M = 44.8, SD = 10.9)$, and for the 220 second group: $(M = 225, SD = 27.7)$. These results indicate that the cost function is good to differentiate between good and bad executions. That is, picking any of the execution plans within the lower cost ones would results in a good execution (shorter time). Note that the differences in cost between the best and the worst execution plan make it straight forward to differentiate between good and bad plans. Also, even if the execution plan with the best cost is not the best overall performing execution plan, its performance is comparable to the best performance.

Since same transformation was used for deriving the cost formula the observed results were expected. Still, it is important to validate that the relationship between cost and execution time is caused by something other than random chance. For this, an analysis of variance (ANOVA) test[11] was conducted. The null hypothesis is that there is no significant difference between the execution times of the execution plans.

The results of the ANOVA test are shown in Table 5.2. Since $p < .001$, we can discard the null hypothesis. Thus, the execution times of the plans are statistically different. To find if there is a correlation between the cost and the observed execution time, a Tukey Honest Significant Differences (Tukey-HSD) post-hoc test was performed on the data. The Tukey-HSD results are presented in Table 5.3. The results support the previous discussion on the capacity of the cost function to differentiate between good and bad executions.

With the cost function in place it is possible to generate the IG for any QVTc transformation.

Example 13. The IG for the UML2RDBMS example is presented in Fig. 5.8. Note that as discussed, only the least-cost

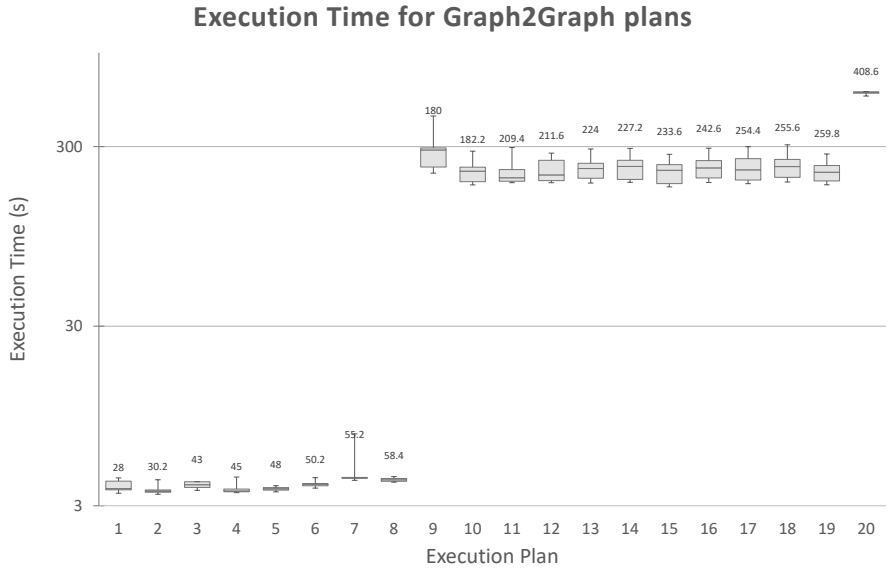


Figure 5.7: Cost evaluation on a 20 execution plans sample of the graph transformation.

Table 5.2: ANOVA test results for the cost function.

	df	SumSq	Mean Sq	F value	Pr(>F)
Cost	1	3.01E+12	3.01E+12	372	$< .001$
Residuals	198	1.60E+12	8.07E+09		

derivation is used and as such the invocations for m_{fa} have a cost of 11 from other mappings except the root. The root cost is 14 as the root only produces elements from the source model and hence can only produce `Class` or `Attribute` elements. From these, using the `Class` has the best cost. ◫

There exists a risk in deriving the cost function based on only one transformation. Mainly, it is possible that the derived cost function is biased to transformations with similar characteristics as the Graph2Graph and will not work in the general case. The behaviour of the cost function was revisited while developing the execution plan synthesis algorithm for all the development examples (see Sect. 5.6). Although an exhaustive evaluation was not carried out (due to the effort to build all possible solutions and analyse the DDG), the results observed during development provided enough

Table 5.3: HSD test results for the cost function.

Group	Plan	Cost	Execution time (ms)
a	20	408.6	5.98E+05
b	9	180	2.89E+05
b	18	255.6	2.35E+05
b	14	227.2	2.30E+05
b	17	254.4	2.30E+05
b	16	242.6	2.29E+05
b	13	224	2.25E+05
b	12	211.6	2.24E+05
b	19	259.8	2.20E+05
b	15	233.6	2.19E+05
b	10	182.2	2.17E+05
b	11	209.4	2.14E+05
c	7	55.2	4.62E+03
c	8	58.4	4.21E+03
c	6	50.2	3.95E+03
c	3	43	3.92E+03
c	1	28	3.86E+03
c	5	48	3.73E+03
c	4	45	3.72E+03
c	2	30.2	3.66E+03

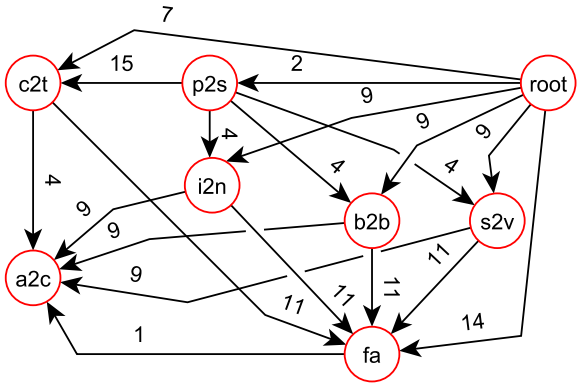


Figure 5.8: Invocation graph IG for the UML2RDBMS example.

confidence in the usefulness of the derived cost function. However, an assessment of the effectiveness of the cost function was carried out as part of the evaluation of the approach proposed in this research (see Chap. 6).

5.2.4 The Objective Function

Given the problem model $P = (S, \Omega, h)$, so far the edges in the IG have been identified as the domain for the values of the variables in S and Ω has been related to Definition 19. To complete the combinatorial optimization model problem this section defines the objective function. Note that finding the minimum-cost execution plan from the IG is also similar to the MWRSA [86, 71] problem.

Let $\mathcal{EP}$ be the set of all spanning subgraphs in IG that are rooted in the *root* mapping. In this context, note that the spanning subgraph is a directed rooted graph in which all arcs point away from the root vertex and where loops are allowed. Moreover, note that $\mathcal{EP}$ contains all subgraphs, not only the ones with maximal size. The objective function value (that is, the weight) $h(Q)$ of a subgraph $Q \in \mathcal{EP}$ is defined as follows:

$$h(Q) := \sum_{ip \in edges(Q)} ic(ip) \tag{5.5}$$

This objective function will allow us to find a spanning subgraph $Q^* \in \mathcal{EP}$ such that the weight of Q^* is smaller than or equal to all other spanning subgraphs in $\mathcal{EP}$ found by the ACO algorithm. Q^* will be the best good-enough execution plan that can be found. The main difference with respect to the MWRSA problem is that IG is not a Directed Acyclic Graph (DAG). Concerning existing work, the literature only offers solutions for the case in which the graph is a DAG [19, 71]. Therefore, the proposed algorithm must consider loops while evaluating $h(Q)$ and Ω. This can be easily implemented by keeping a cache of visited nodes which allows identification of loops.

Given that the derivation information can be computed from the PDG and hence the minimum derivations found, the IG can

be computed before the execution plan synthesis algorithm is executed.

5.3 Ant Colony System for Execution Plan Synthesis

The previous section showed how the execution plan synthesis problem can be modelled as an optimization problem amenable to be solved using the ACO metaheuristic. This section presents the details of the implementation of the ACO algorithm. The section starts by defining the foraging area and then presents the algorithm details, including the pheromone update conditions, the chosen settings for the algorithm parameters and the terminating condition. Details on the feasible neighbourhood selection are presented in the next section.

5.3.1 The Foraging Area

Imagine that a grid is placed on the foraging area and the ants are only allowed to travel from/to the nodes in the grid. An ant is considered to have constructed a solution depending on the problem being solved by assessing, for example, the number of nodes the ant has visited. Additional conditions include reaching a specific target, number of visited nodes, visiting all nodes a given number of times, among others. Further conditions can be placed on the links between nodes to allow/disallow ants to travel between specific nodes. The construction graph represents this grid.

In the execution plan synthesis setting, a solution is constructed by assigning invocation paths to the variables in the solution space. Hence, the nodes the ant must visit are the edges from the invocation graph $IG = (M, IP', ic)$ (see Definition 23). We define the construction graph $\mathcal{C}(IG)$ for the Execution Plan Synthesis (EPS) synthesis problem a directed graph on the $l + 1$ nodes $\{ip_0, ip_1, \ldots, ip_l\}$, with a designated origin node $o := ip_0$ as

proposed by Neumann et al. [78]. The edge set $\mathcal{F}$ of cardinality l^2 is given by

$$\mathcal{F} := \{(ip_i, ip_j) | 0 \leq i \leq l, 1 \leq j \leq l, i \neq j\} \tag{5.6}$$

i.e., $\mathcal{C}$ is obtained from the complete directed graph by removing all self-loops and the edges pointing to o.

When an ant visits a node in $\mathcal{C}$ it is choosing the next invocation that will be scheduled. In this representation, the cost measure $\varsigma(ip_i, ip_j)$ of each edge $f \in \mathcal{F}$ is the derivation cost associated to the target node of the edge, that is $\varsigma((ip_i, ip_j)) = ic(ip_j)$. Additionally, a desirability measure $\tau(ip_i, ip_j) : D \rightarrow \mathbb{R}$ is added to each edge. The desirability measure assigns each edge a real–valued *pheromone* value.

5.3.2 The Synthesis Algorithm

Algorithm 2 ACS algorithm skeleton.

```
1: procedure ACS
2:     initColony
3:     initPheromoneTrails
4:     repeat
5:         resetColony
6:         constructPlans
7:         updateTrials
8:         Q^ib ← bestIterationSolution
9:         if h(Q^ib) < h(Q^gb) then
10:            Q^gb ← Q^ib
11:        end if
12:    until termination condition
13: end procedure
```

The Ant Colony System (ACS) algorithm is presented in Algorithm 2. Procedure initColony() initializes the colony and sets the algorithm configuration parameters. Procedure initPheromoneTrails() initializes the pheromone values in $\mathcal{C}(IG)$. The main loop is repeated until a termination condition is met (usually number of iterations or time limit). On each iteration of the main loop, the ants construct valid execution plans and the pheromone values are updated. After each iteration, the iteration–best solution (Q^{ib})

cost is compared to the global–best solution (Q^{gb}), and if lower the global–best solution is updated.

5.3.3 Plan Construction

Algorithm 3 Plan construction.

```
1: procedure constructPlans(colony)
2:     forEach ant ∈ colony do
3:         while ¬isSolutionReady(ant) do
4:             IP* ← getNeighbours(ant)
5:             ip ← stateTransition(IP*)
6:             updateSolution(ant,ip)
7:         end while
8:     end for
9: end procedure
```

The plan construction algorithm is presented in Algorithm 3. Each ant in the colony constructs a candidate solution by starting with an empty solution and iteratively adding invocations to the execution plan until it is complete, which is verified by procedure isSolutionReady. Procedure getNeighbours calculates the valid set of invocation paths the ant can visit next and procedure stateTransition evaluates the *state transition rule* for each ant on the neighbourhood. Finally, procedure updateSolution updates the ant's memory by adding the last selected invocation path to the plan under construction, Q_a. Next, this section describes the *state transition rule*. The details of the neighbourhood selection are presented in Sect. 5.4.

The candidate solution can be represented by a list of edges $Q = [ip_1, ip_2, \ldots, ip_n]$. The order of the invocations edges in Q represents the structure of the execution plan as explained in Sect. 5.2.1. The *state transition rule* is defined as follows: an ant positioned at node ip_r chooses the next invocation path to schedule ip_s by applying the rule given by (from (2.4)):

$$ip_s = \begin{cases} \arg \max\limits_{ip_u \in \mathcal{N}^k_{ip_r}} \{[\tau(ip_r, ip_u)] \times [\eta(ip_r, ip_u)]^\beta\}, & \text{if} \quad q \leq q_0 \\ S, \quad \text{otherwise} \end{cases} \tag{5.7}$$

where $\arg\max_x f(x) = \{x \mid f(x) = \max_{x'} f(x')\}$, τ is the pheromone, $\eta = 1/\varsigma$ is the inverse of the cost, q is a random number in $U(0,1)$ (standard uniform distribution), q_0 is a parameter ($0 \leq q_0 \leq 1$), $\mathcal{N}^k_{ip_r}$ is the set of invocations that an ant can visit according to Ω (feasible neighbourhood), β is a parameter which determines the relative importance cost vs. pheromone ($\beta > 0$), and S is a random invocation selected according to the probability distribution given by (from (2.2)):

$$p_k(ip_r, ip_s) = \begin{cases} \dfrac{[\tau(ip_r, ip_s)]^\alpha \times [\eta(ip_r, ip_s)]^\beta}{\sum\limits_{ip_u \in \mathcal{N}^k_{ip_r}} [\tau(ip_r, ip_u)]^\alpha \times [\eta(ip_r, ip_u)]^\beta}, & \text{if } s \in \mathcal{N}^k_{ip_r} \\ 0, \text{ otherwise} \end{cases} \tag{5.8}$$

By multiplying the pheromone on edge (ip_r, ip_s) by the heuristic value (in this case the invocation cost), the probability distribution favours the selection of invocations that are cheaper and that have a higher level of pheromone.

The parameter q_0 in (5.7) determines whether the ant favours *exploitation* or *biased exploration*. Biased exploration, defined by (5.8), encourages the ants to move to invocations with lower cost and larger amount of pheromones. The problem with biased exploration is that the colony search can concentrate "early around suboptimal solutions leading to a premature stagnation of the search." [98]. This effect is balanced with exploitation, in which case ants can sometimes, if $q \leq q_0$ (the ant samples a random number $U(0,1)$ when choosing the next invocation), move to invocations with the highest cost. This choice enables the colony to escape from local suboptimal search areas.

5.3.4 Pheromone Trail Updating

One of the key aspects of the $\mathcal{MM}$AS, is that only one ant is allowed to deposit pheromones on the foraging area. This ant "may be the one which found the best solution in the current iteration (iteration–best ant) or the one which found the best solution from the beginning of the trial (global–best ant)." [98]. Consequently, the *global pheromone updating rule* is given by(from (2.3)):

$$\tau(ip_r, ip_s) \leftarrow \rho \times \tau(ip_r, ip_s) + \Theta\tau^{best}(ip_r, ip_s) \tag{5.9}$$

where ρ is the pheromone decay parameter and $\Theta\tau^{best}(ip_r, ip_s) = 1/h(Q^{best})$, $h(Q)$ is defined by (5.5) and Q^{best} denotes the iteration–best (Q^{ib}) or the global–best solution (Q^{gb}).

To select between Q^{ib} or Q^{gb}, this research will follow the dynamical mixed best ant feedback strategy proposed by Stützle et al. [98], where Q^{ib} is used to update the pheromone information but at given intervals use Q^{gb}. The intervals are given by frequency f^{gb}. The value of f^{gb} is gradually increased to veer the search from an early exploratory phase to a late exploitation of the overall best solution phase. Section 5.3.5 shows how f^{gb} is defined.

The second key aspect of the $\mathcal{MM}$AS is that the range of possible pheromone values is limited to the interval $[\tau_{\min}, \tau_{\max}]$. This research adjusts the pheromone intervals as presented in Sect. 2.6.2. The pheromone value obtained from (5.9) is truncated such that $\tau_{\min}(t) \leq \tau(ip_r, ip_s)(t) \leq \tau_{\max}(t)$.

5.3.5 Parameter Settings

Stützle et al. [99] presents a comprehensive study of the effects of the different parameters in the behaviour of ACO algorithms. The authors conclude that existing work on dynamic parameter adaptation is inconclusive and lacks an in–depth understanding of the effect of individual parameter's settings. As a result, this research uses static settings for the ACS algorithm. The parameter settings are selected based on the observations in [98] and the settings of the ACS algorithm [32].

Table 5.4: Dynamic *best ant feedback strategy* frequency values.

Iteration (t) range	f^{gb}
$0 < t \leq 25$	0
$25 < t \leq 75$	5
$75 < t \leq 125$	3
$125 < t \leq 250$	2
$t > 250$	1

The parameter settings are as follows. For the random pheromone selection $\alpha = 1$ [32] and $\beta = 2$ [98]. The number of ants is set to 25, since it is the lower suggested value for $\mathcal{MM}$AS and ACS works better for smaller populations [32, 98]. Since local search is not used, $\rho = 0.75$ was set as a middle point between 0.6 [98] and a high value (e.g. 0.9) [32]. For q_0 a value of 0.75 presents the best trade–off between results with short and long runtimes [98]. For $\tau_{\min}$ and $\tau_{\max}$ the default settings proposed by Stützle et al. [98] are used. Since both values depend on $f(Q^{best})$, for $\tau(0)$ the pheromone is set to some arbitrary high value. After the first iteration, the pheromone value will be between the imposed limits. Finally, the dynamic best ant feedback strategy used is depicted in Table 5.4, taken from Stützle et al. [98].

All the configuration parameters are set to their initial values in procedure INITCOLONY in Algorithm 2.

5.3.6 Termination Condition

Existing literature does not provide additional information on the criteria to establish the termination condition for the ACO algorithm. From Stützle et al. [98], a terminating condition of $2500 \times (l + 1)$ iterations is set (recall that $l + 1$ is the number of nodes in $\mathcal{C}$), "which is sufficient to achieve convergence of $\mathcal{MM}$AS" [98]. Since the EPS problem is not comparable to any of the existing, well documented graph path problems this research proposes a different approach.

In the ACO, the agents will eventually start building plans

that share a lot of common invocations with Q^{gb} and hence it is expected the cost of all these plans to be similar. Thus, $f(Q^{ib})$ is expected to oscillate around $f(Q^{gb})$. With this type of behaviour; the terminating condition was defined based on the confidence interval for the mean of the samples of $f(Q^{ib})$. Since the standard deviation of the population of $f(Q^{ib})$s is unknown, the $t-$Confidence Interval is used to estimate if the ACS is oscillating around a cost interval that will likely include the mean of the complete population of plans. The confidence interval can be used in this case because the ants are randomly sampling the execution plans. The algorithm terminates when the confidence interval of the solutions' cost is below 0.05%.

5.4 Feasible Neighbourhood

In an ACS implementation, the constraints in Ω can either be used to model the foraging area (the foraging area can be an alternate representation of the problem) or to place further restrictions on the paths the ants can follow during exploration. For example, when using the ACS for solving problems like the TSP, the constraints in Ω are use to restrict the ants' paths. In the TSP problem the feasible neighbourhood of an ant is all cities that the ant has not visited yet. That is, the constraint that all cities must be visited once is used to limit the possible cities to which the ant can travel next. This research proposes the use of Ω to restrict the feasible neighbourhood of the ants. By restricting the feasible neighbourhood, it is guaranteed that all constructed solutions satisfy Ω. Next, we describe how these constraints can be used to determine the feasible neighbourhood $\mathcal{N}^k_{ip_r}$ when the ant needs to transition to another node.

When Ω was introduced in Sect. 5.2, it was stated that these constraints where defined by Definition 19, which places constraints on the structure of the solution being constructed in two respects: partial order and thoroughness.

5.4.1 Partial Ordering Validation

Given the construction graph $\mathcal{C}(IG)$, for an ant that is at node $ip_r(m_{s_r}, m_{t_r})$, the algorithm to validate if $ip_f(m_{s_f}, m_{t_f})$ should be added to $\mathcal{N}^k_{ip_r}$ according to the partial ordering is presented in Algorithm 4, where Q' denotes the solution under construction. The loop at line 3 validates the *execution–compulsory dependent* and the loop at line 10 the *execution–alternative dependent* relations from Definition 16. The first loop checks that all compulsory mappings have been invoked and the second loop checks that for each type at least one of the alternative mappings has been invoked.

Algorithm 4 Partial order validation.

```
1:  procedure partialOrderValidation(ip_f, M)
2:      valid ← true
3:      for all m_p ∈ M do
4:          if m_p Φ m_{t_f} then
5:              valid ← valid ∧ ∃ip_j ∈ Q' | m_{t_j} = m_u
6:          end if
7:      end for
8:      if valid then
9:          valid ← false
10:         for all z ∈ {v ∈ IN(m_{t_f}) : type(v)} do
11:             if |SPL_z(m_{t_f})| > 1 then
12:                 valid ← valid ∨ ∃ip_j ∈ Q' | m_{t_j} Ψ_v m_{t_f}
13:                 if valid then
14:                     break
15:                 end if
16:             end if
17:         end for
18:     end if
19:     return valid
20: end procedure
```

Example 14. Consider the construction step in Fig. 5.9. Note that the presented IG has the adjusted cost values. At iteration $t-1$ the ant has moved to the invocation edge $ip_r(m_{p2s}, m_{c2t})$ (Fig. 5.9a), with the resulting Q' in Fig. 5.9b.

Recall that the construction graph $C(IG)$ connects each invocation to all other invocations, so the ant could select any of the remaining black invocations. However, by applying Algorithm 4 invocations that will produce invalid plans can be filtered out. Some of the available invocations (diamond boxed

numbers) have been labelled to analyse their validation at iteration t, based on the proposed algorithm.

1. $m_{fa} \Phi m_{a2c} \wedge m_{fa} \notin Q'$: this invocation is not added to the feasible neighbourhood.

2. $m_{c2t} \Phi m_{fa} \wedge m_{c2t} \in Q'$, and $m_{12n} \Psi m_{fa} \wedge m_{12n} \in Q'$: this invocation is added to the feasible neighbourhood.

3. $m_{p2s} \Phi m_{b2b} \wedge m_{p2s} \in Q'$: this invocation is added to the feasible neighbourhood.

4. $m_{fa} \Phi m_{a2c} \wedge m_{fa} \notin Q'$: this invocation is not added to the feasible neighbourhood.

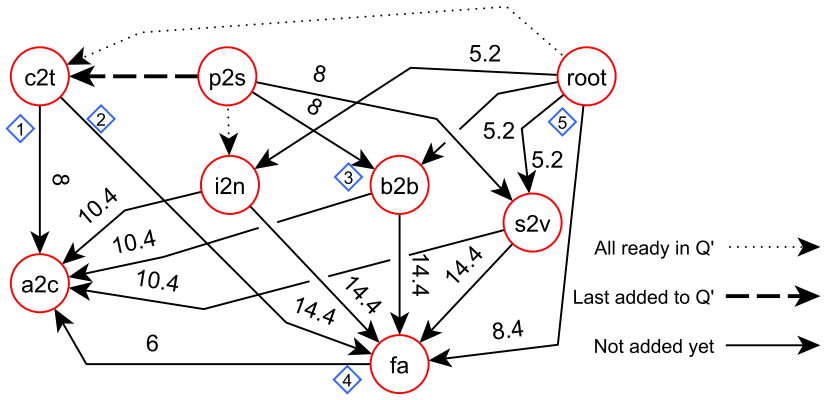
5. $m_{p2s} \Phi m_{s2v} \wedge m_{p2s} \in Q'$: this invocation is added to the feasible neighbourhood. ◫

From this example there is one characteristic of the plan construction that is worth mentioning: the plans are not constructed following a possible execution order. In other words, the ant is not restricted to continue building the plan just with possible invocations paths from the last added invoked mapping. In the above example, this means that the ant does not only look at invocations paths that start at m_{c2t}. In this case, since the invocations from root to m_{b2b} and m_{s2v} have a lower cost, it is more probable that the ant will pick one of them instead of invocations that start a m_{c2t}. This means that the ant constructs the solution randomly and not following a plausible scheduling order.

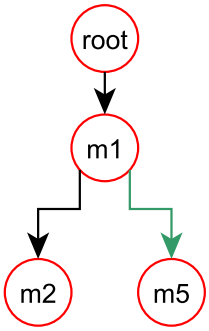
5.4.2 Thoroughness Validation

Thoroughness is directly related to the creation and consumption of types. That is, all *producer–consumer* relations must be satisfied. Consequently, thoroughness validation can only be performed when all mappings have at least been scheduled once.

Consider the plan under construction presented in Fig. 5.10a. This plan presents the behaviour discussed in Sect. 4.3 regarding



(a) Foraging Options in the IG

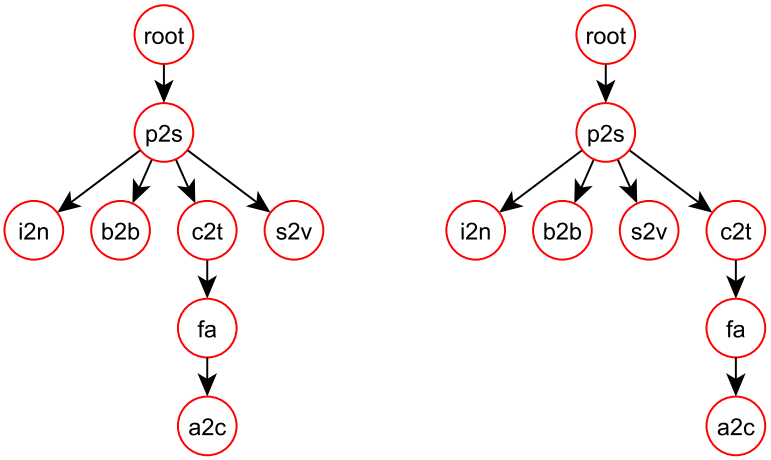


(b) Q'

Figure 5.9: Feasible neighbourhood partial order validation.

thoroughness, in particular the `PrimitiveToName` elements produced by m_{s2v} are never consumed. Since $m_{fa} \in CNS(m_{s2v})$, no columns will be generated for attributes with a string primitive type. In this case, $\mathcal{N}^k_{ip_r}$ should include all invocations that have m_{fa} as a target to allow the ant to (possibly) pick an invocation that will make the plan thorough next.

Additionally, consider the plan under construction presented in Fig. 5.10b. At a first look, this plan seems to be thorough. Compared to the previous case, m_{fa} is invoked after m_{i2n}, m_{b2b} and m_{s2v} and hence all attributes of all primitive types are considered. Further, m_{fa} is invoked from m_{c2t} consuming `ClassToTable` elements produced, and finally m_{a2c} is called from m_6 consuming



(a) Missing Consumers (b) Shadowed Consumers

Figure 5.10: Thoroughness validation of invocations in the feasible neighbourhood.

all `AttributeToColumn` elements produced. However, note that m_{c2t} also produces `Table` elements. Arguably, since m_{a2c} consumes `Table` elements it should consume the elements produced by m_{c2t}. However, the `Table` element consumed by m_{a2c} is derived from `a2c:AttributeToColum` (via `a2c.owner.table`) and since a complete static property access/write analysis is not performed, it is impossible to prove that this derived element is the same one that m_{c2t} produced. As a result, m_{a2c} is *shadowed* from consuming elements from m_{c2t} and therefore, $\mathcal{N}^k_{ip_r}$ should include all invocations that have m_{c2t} as a source. Shadowing refers to the fact that a mapping cannot see beyond the context of the source of its invocation path and hence, cannot be assumed to consume types produced by previous invocations if the variables of these types are not primary variables and it an all-loop is not used for ascription.

5.4.3 Containment Relations

Finally, one interesting aspect of execution plan validation comes from the semantics of the modelling technology, rather than from the semantics of the QVTc language. In particular, this research

considers the composition relations from the UML Specification [5], which are supported by Meta Object Facility (MOF) models[2].

Formally, composition is defined as follows. In the UML an `Association` has a reference *memberEnd:Property*, that determines the (two or more) `Properties` that are involved in the association. Each *Property* has an attribute *aggregation: AggregationKind* that specifies the kind of aggregation that applies to the Property. When the aggregation is *composite*, it indicates that the composite object has responsibility for the existence and storage of the composed objects (parts). For example, in the UML metamodel of the Minimal UML2RDBMS Example (Fig. 3.2), a `Package` contains `PackageElements`, and a `Class` contains `Attributes`. If a package is removed from a model all its classes will be removed too.

Now, consider mapping m_{fa} in the Minimal UML2RDBMS presented in Listing 3.5. Note that {`Class`, `PrimitiveDataType`} $\subset IN(m_{fa})$ and `Package` $\notin IN(m_{fa})$. As a result the mapping does not place any restriction on the container of the classes and primitive data type (which complies to the UML semantics, i.e. the type of an `Attribute` can be defined in any package, not just the package in which the attribute's owner is defined). If mapping m_{fa} was invoked from m_{c2t} and m_{c2t} from m_{p2s} then, it is possible that for some invocations m_{fa} would *starve* because the package that owns the `PrimitiveDataType` might not have been transformed yet and thus the required `PrimitiveToName` would not exist (assuming m_{p2s} invokes m_{i2n},m_{b2b} and m_{s2v}).

To solve this issue, the types in $IN(m)$ are examined and the respective metamodel is queried to identify the containment relations between them. If two variables $v_1 : t_1, v_2 : t_2$, with $t_1, t_2 \in IN(m)$ each related by a containment association to t_3, then there must exist a third variable $v_3 : t_3$, and predicates $v_1.p_1 = v_3$ and $v_2.p_2 = v_3$ (where p is the containment property).

[2]Note that we are referring to UML as the modelling capabilities supported by MOF[4] models, not the simplified UML domain in the running example.

That is, v_1 and v_2 must have the same container. If not, the invocation cannot be added to $\mathcal{N}^k_{ip_r}$. Further, the mapping must be invoked from the root (respecting the mapping ordering).

5.5 QVTi Generation

Once the best execution plan has been found the resulting QVTi MTP must be constructed. For this the loop and invocation actions in the execution plan are transformed into QVTi *for loops* and *mapping calls* respectively. Next, the derivation information from the intra-mapping dependency analysis is used to create Object Constraint Language (OCL) expressions that result in the correct variable derivations from the invocation context and loops. Finally, the mappings (QVTm) are inserted (without modification). The result of the merge process is a QVTi Abstract Syntax Tree (AST) which (using existing functionality) is then saved using the QVTi concrete syntax in a text file.

5.6 Development Transformations

This section presents and overviews the transformations used for development and testing of the ACS algorithm, referred to hereafter as the execution plan synthesis algorithm (EPSA). Systematic evaluation of the EPSA is presented in Chap. 6. The intention of each transformation is discussed, and the solution found by the EPSA presented. Particular characteristics of each example and how they related to the generated execution plan are also discussed.

In general, most of the development transformations exhibit one–to–one relations between the concepts of the source domain and the concepts of the target domain. In the representation of the synthesized execution plans the derivation information has been added to the invocation edges for completeness.

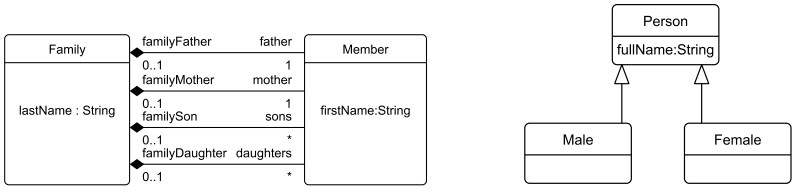
All resulting plans were executed using the QVTc virtual machine and the target models where validated via visual inspection and by comparison to the models generated by the *naïve plan*. In all cases, all models were correct by inspection and both sets of target models were identical.

The transformations were selected to explore different aspects of the EPSA but to be of manageable size. The development transformations have a low number of mappings so the different artefacts (DDG, PDG, WPDG, minimum derivations and $\mathcal{C}$) constructed as part of the synthesis could be verified manually and that a step by step execution of the EPSA was easy to follow. The examples were selected to have DDG with loops and PDG with derivation paths involving multi-valued properties. In the figures that show execution plans additional lables have been added to the invocation edges in order to show the variable derivations used for that invocation.

5.6.1 Families to Persons

The purpose of this transformation is to transform a model from a domain where *Families* are represented by their member structure (father, mother, sons/daughters), to a domain where *Persons* are identified by their gender and full name. The respective metamodels are presented in Fig. 5.11. The complete transformation script is presented in Annex 8.1.

The transformation consists of two mappings: `Member2Male` and `Member2Female`. As their names imply each mapping is responsible for transforming a male and female member of the family respectively, to their persons equivalents. For example, a *father* becomes a *male* and a *daughter* becomes a female. This transformation is a very basic and its purpose was to test the minimal behaviour of the EPSA. With no dependencies between rules, they can be invoked in any order. Figure 5.12 presents the plan generated by the EPSA.



(a) Families metamodel (b) Persons metamodel

Figure 5.11: Families2Persons Metamodels

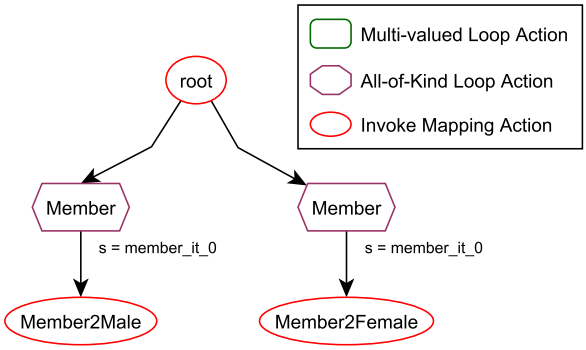


Figure 5.12: Families2Persons Synthesized execution plan.

5.6.2 Upper to Lower

The Upper to Lower transformation is an extension to the graph to graph transformation introduced in Sect. 5.2.3. The extension consists in changing the case of the names of the input graph to all lower case letters. The graph metamodel is presented in Fig. 5.13 The complete transformation script is presented in Annex 8.2. The transformation consists of three rules, one for each type of the graph metamodel.

Figure 5.14 presents the plan generated by the EPSA. Note that an invocation of *edge2edge* from *node2node* would be invalid because *edge2edge* consumes two variables of type `Node`. Mapping *edge2edge* has been correctly invoked after all nodes have been transformed.

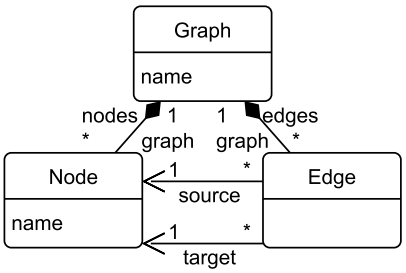


Figure 5.13: Graph Metamodel

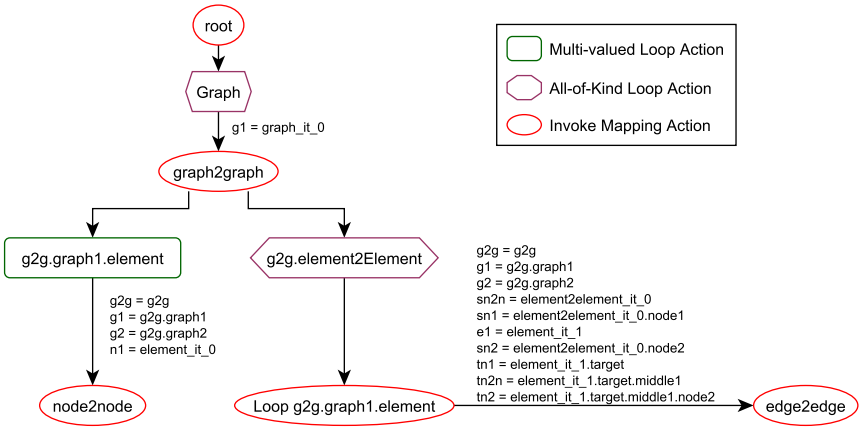


Figure 5.14: Upper2Lower Synthesized execution plan.

5.6.3 HSV to HSL

The HSV (Hue, Saturation, Value) to HSL (Hue, Saturation, Luminosity) transformation is similar to the Upper2Lower, in the sense that both domains have a metamodel with the same structure. In this case the structure is a tree, in which each node can have multiple children and a single parent. The metamodel is presented in Fig. 5.15. The complete transformation script is presented in Annex 8.3.

The transformation consists of two rules. One to transform the root/top node (i.e. no parent) and a second rule that recursively transforms the children. In particular this transformation exhibits a loop due to recursion. The purpose of this example was to understand the implications of recursive transformations in the execu-

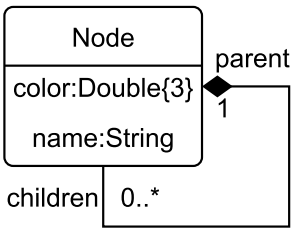


Figure 5.15: HSV and HSL Metamodel

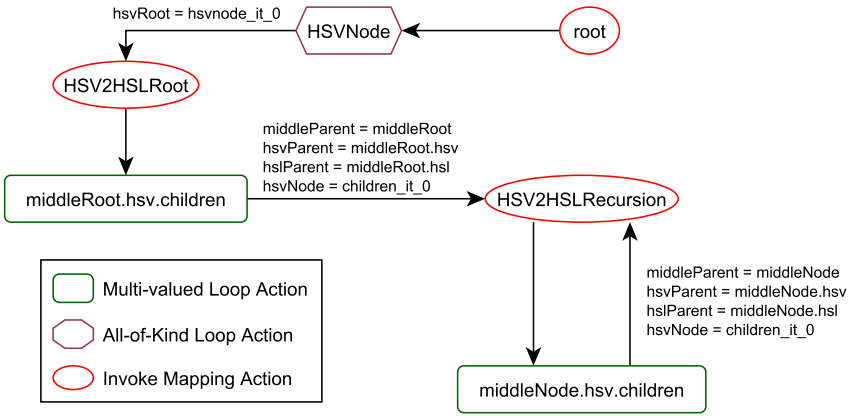


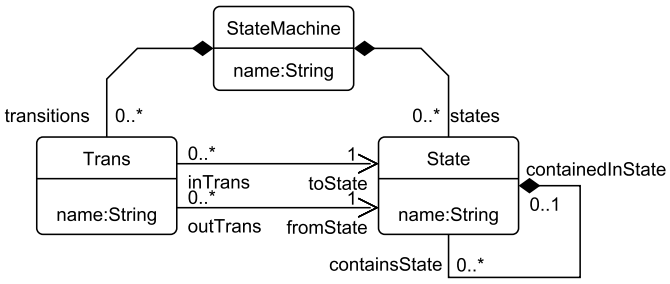
Figure 5.16: HSV2HSL Synthesized execution plan.

tion plan construction. Figure 5.16 presents the plan generated by the EPSA. Mapping *HSV2HSLRecursion* correctly invokes itself in order to provide the recursive behaviour.

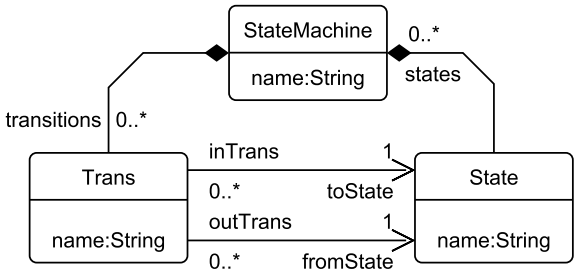
5.6.4 Hstm to Stm

The Hierarchical State Machine to (flat) State Machine transformation is based on the *Hstm2Stm* QVT Relations [3] (QVTr) transformation available in the Eclipse QVTd examples. As the name implies, the transformation takes a hierarchical state machine at the input and produces a flattened state machine. The metamodels are presented in Fig. 5.17. The complete transformation script is presented in Annex 8.4.

The transformation consists of three rules. Two rules to transform leaf and container states (*LStateToState* and *CStateToState*)



(a) Hstm metamodel



(b) Stm metamodel

Figure 5.17: Hstm2Stm Metamodels

respectively, and one additional rule to transform transitions. Figure 5.18 presents the plan generated by the EPSA. As with the Upper2Lower example, the *HTransToTrans* mapping invocation is restricted by *Path Validity* (4.3) and hence it cannot be invoked from the mappings that produce `Stm::State` elements as it consumes two elements of this type.

5.6.5 UML to RDBMS Minimal

Figure 5.19 presents the plan generated by the EPSA for the running example. This plan is consistent with the different scenarios that have been discussed in this and previous chapters. The complete transformation script is presented in Annex 8.5.

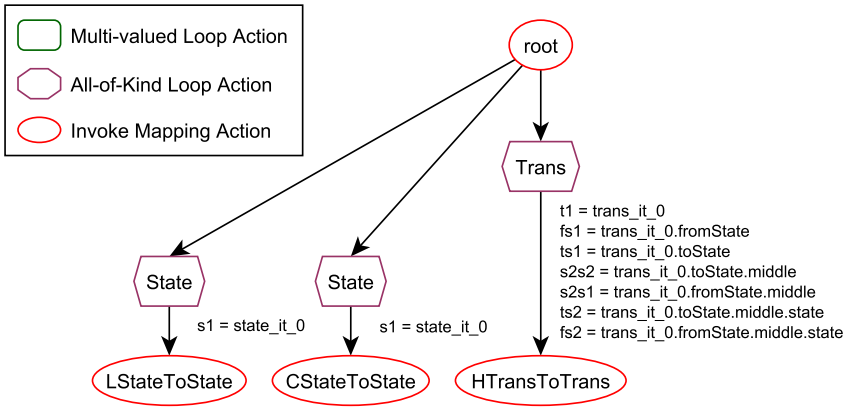


Figure 5.18: Hstm2Stm Synthesized execution plan.

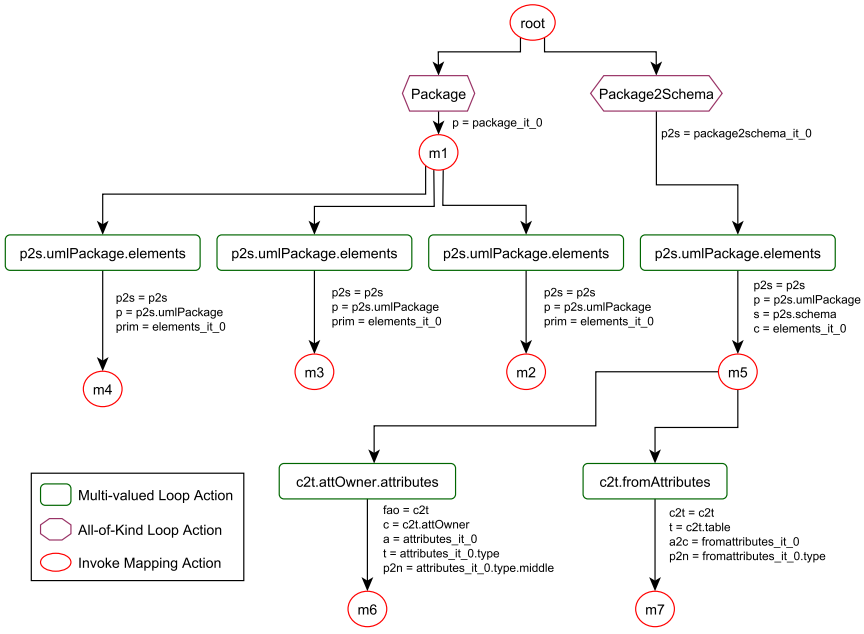


Figure 5.19: Minimal UML2DBMS Synthesized execution plan.

5.6.6 UML to RDBMS Complete

The UML to RDBMS transformation is based on the Meta Object Facility (MOF) Query/View/Transformation (QVT) Specification example. The complete transformation, as opposed to the running example, has additional rules to handle associations and attributes with complex types (i.e. an attribute that has a Class as a type). Further, for attribute and association transformation, the hierarchy of classes is observed (i.e. inherited attributes and associations are correctly transformed).

The transformation consists of 11 rules. Most of the rules are one-to-one, except for the attribute rules which takes into consideration primtive and complex types. Figure 5.20 presents the plan generated by the EPSA.

5.6.7 Experimental Results for the Development Transformations

In this section we report on experimental results obtained with the EPSA algorithm on the development examples. The experiments where performed on an Intel Core i5-4460 3.2GHz processor with 8 GB RAM, running Windows 10 and using a 64–bit Java VM (version 1.8.0_111).

The results are given in Table 5.5, gathered over 25 trials for each transformation. The table shows, for each transformation, the cost of the best execution plan found, the average iteration (numbered from 0) at which the best execution plan was found, the average time to find the best execution plan and the average maximum total number of iterations (i.e. after how many iterations the algorithm stopped). Finally, the maximum total execution time of the algorithm (i.e. the total execution time of the longest execution) is given. Only the best cost is reported as the objective of the algorithm is to find the best plan. Later in this section a discussion on the different costs is presented to analyse the space exploration aspect of the algorithm.

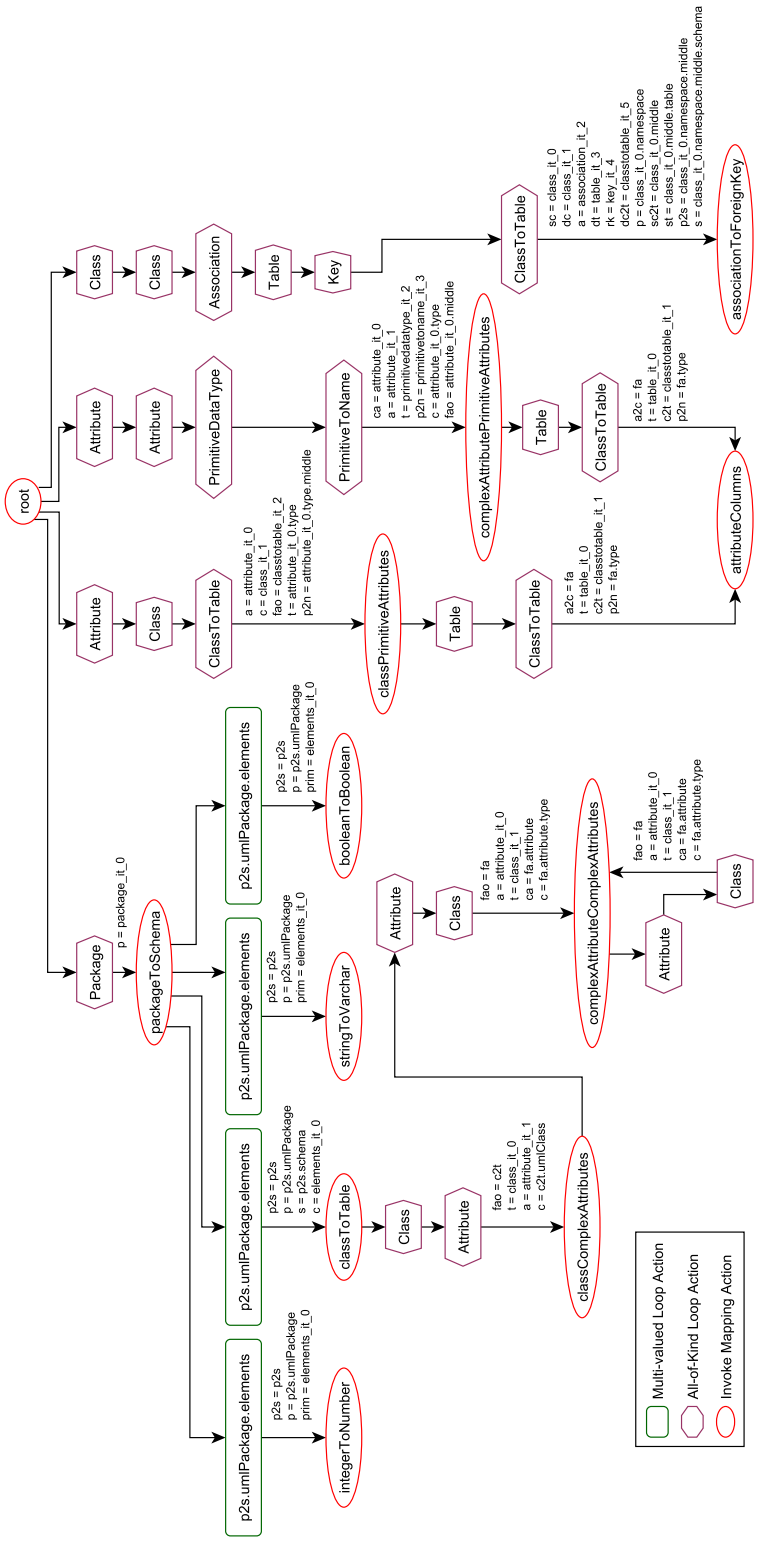


Figure 5.20: UML2DBMS Synthesized Plan

Table 5.5: Comparison of the EPSA on the development transformations.

Example	Best	i_{avg}	t_{avg}(s)	$i_{\max}$	$t_{\max}$(s)
Families2Persons	5	0	0.014	205	2.23
Upper2Lower	28	0	0.013	219	3.03
HSV2HSL	13	0	0.011	205	2.68
Hstm2Stm	19	0	0.016	219	3.17
UML2RDBMS Minimal	29	0	0.085	246	19.0
UML2RDBMS	164	0	0.39	257	99.3

Given are the example name, the best solution cost, the average number of iterations i_{avg} (numbered from 0), the average time t_{avg} to find the best solution in an iteration, the maximum number of iterations $i_{\max}$ and the maximum computation time $t_{\max}$. Averages are taken over 25 trials.

The results show that our algorithm is able to find solutions for all the development transformations. It is interesting to note that in all cases, the algorithm finds the best plan in the first iteration. This behaviour can be attributed to three aspects:

1. The $\mathcal{MM}$AS algorithm with pheromone limits and *pseudo–random–proportional* transition rule is designed to find an optimal solution fast.

2. The low number of mappings translates to an invocation graph with low fan–out/fan–in (number of edges leaving and incoming a node respectively) measures. As a result, the number of alternative invocations from each mapping is limited.

3. For the larger examples (running example and the complete UML2RDBMS), the metamodels have a very rigid containment structure which results in the feasible neighbourhood being highly constrained by the containment relations (Sect. 5.4.3).

The descriptive statistics for the best solution cost, the average number of iterations and the average time to find the best solution

Table 5.6: Descriptive statistics for the Best, iteration found and time taken.

	Best		i_{avg}		$t_{avg}(s)$	
	M	SD	M	SD	M	SD
Families2Persons	5.00	0	0	0	0.014	0.046
Upper2Lower	28.0	0	0	0	0.013	0.004
HSV2HSL	13.0	0	0	0	0.011	0.006
Hstm2Stm	19.0	0	0	0	0.016	0.033
UML2RDBMS Minimal	29.0	0	0	0	0.085	0.034
UML2RDBMS	164	0	0	0	0.39	0.028

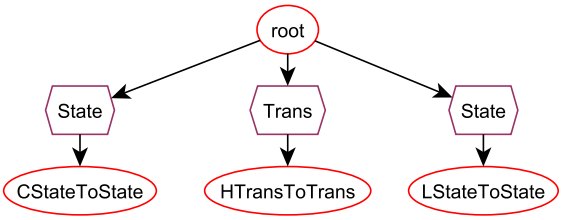


Figure 5.21: Invalid Hstm2Stm execution plan under construction.

in an iteration presented in Table 5.6 suggest that it is only needed to run the algorithm once to find the best solution. Further, they also suggest that only one iteration is required to find the best solution. The reason for this might be the size (number of rules) of the transformations. Chapter 6 presents these same results in a different set of transformations to determine if this behaviour can be generalized.

It was observed that for the Hstm2Stm not all the ants are able to find a solution in all the iterations. This is a result of how the plans are constructed and the validation process. In particular, thoroughness (Definition 18) cannot be validated until the plan is complete. Thus, thoroughness is not evaluated until all mappings have been invoked at least once. If at this point a plan is not thorough, the algorithm will try to add additional invocations that will guarantee thoroughness (by adding them to the ant's feasible neighbourhood, see Sect. 5.4). However, some of the required additional invocations might not be available and hence the plan is

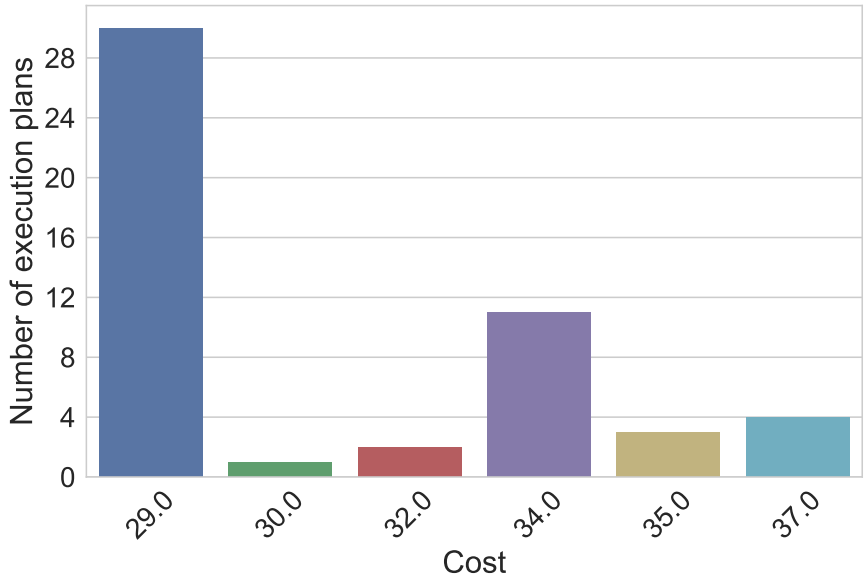
left in an invalid state. For example, consider the partial plan in Fig. 5.21. In order to make the plan thorough, and additional invocation from mapping *root* to mapping *HTransToTrans* is needed. However, since the execution plan is not a multigraph, it is impossible to add this additional invocation and as a result the plan cannot be completed.

As mentioned previously, only the best cost is reported, but the algorithm can find plans of others costs during the search. Further, it may be possible that solutions with different structures have the same cost (for example in the running example the mappings m_{i2n}, m_{b2b} and m_{s2v} invoked from m_{p2s} will have the same cost regardless of the order in which they are invoked). Given that two plans with different structure can have the same cost, the cost information is not sufficient to determine if the ants are constructing different solutions. To further analyse the behaviour of the algorithm additional data is needed.

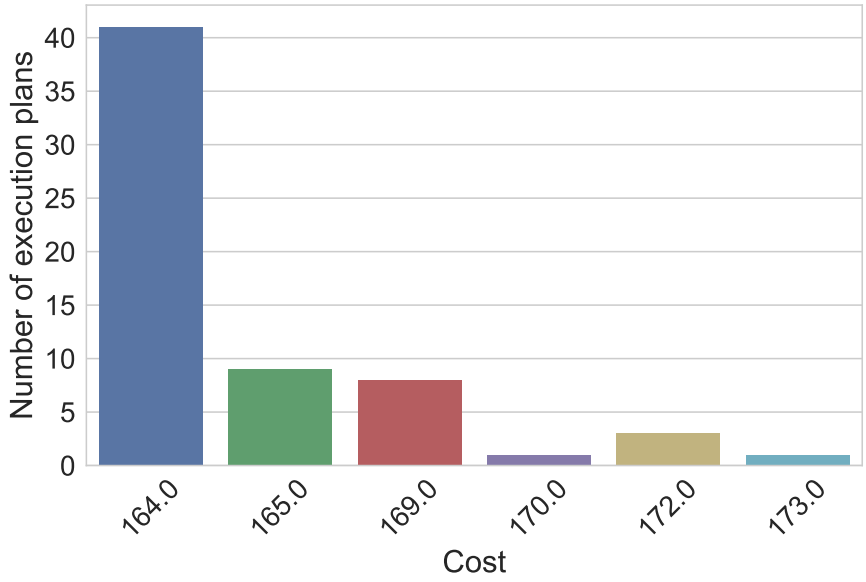
For this, during execution plan construction, the number of different plans of a given cost found during an iteration of the algorithm is logged. The results are presented in Fig. 5.22, which summarizes the information over the 25 trials. The figure only shows the results for the Minimal UML2RDBMS and complete UML2RDBMS. For the Families2Persons and Hstm2Stm examples only two alternatives of the same cost exist; for the Upper2Lower and HSV2HSL examples only 1 alternative for three different costs exits.

The results indicate that the best plan is found more frequently than the others, which is accordant with the characteristics of the EPSA, i.e. the *pseudo–random–proportional* transition rule results in ants constructing minimum cost execution plans (exploitation preference) and as a result ants are more likely to find the best solution.

Chapter 6 will also present results on the execution of the generated plans to compare them to each other and to the *naïve plan*.



(a) Minimal UML2RDBMS



(b) UML2RDBMS

Figure 5.22: Number of execution plans constructed with the same cost.

5.7 Summary

This chapter have presented the execution plan synthesis problem as a combinatorial optimization problem and described an $\mathcal{MM}$AS algorithm to solve the problem. This chapter provided an alternative representation of the problem that is more amenable to the $\mathcal{MM}$AS algorithm and provided detailed information on the key aspects of the implementation of the algorithm. Finally, we have presented the results of running the implementation of the proposed algorithm to synthesize execution plans for a set of QVTc development transformations. The results are consistent with the characteristics of the $\mathcal{MM}$AS algorithm. The next chapter presents the evaluation of the proposed algorithm by generating execution plans on a different set of QVTc transformations and executing the synthesized execution plans using a set of oracle candidate models.

Part III

Evaluation and Conclusions

6

Evaluation

The last chapter introduced the execution plan synthesis algorithm (EPSA), the algorithm proposed in this research to find a solution for the Execution Plan Synthesis (EPS) problem, and showed that the results on a set of initial examples are encouraging. Since these examples were used to fine tune and debug the algorithm, a different set of transformation examples were used to perform the evaluation of the proposed algorithm.

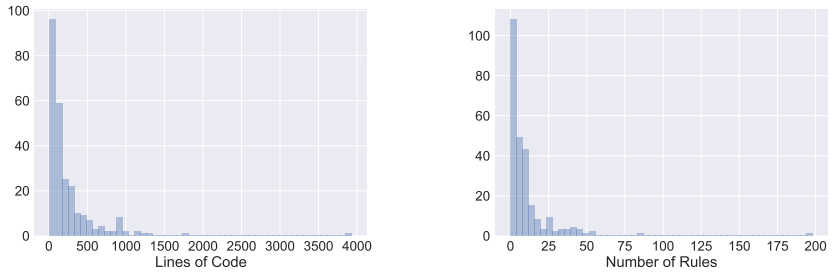
This chapter presents the results of the experiments carried out to evaluate the EPSA with respect to the three research hypotheses. The experiments were designed to evaluate the correctness of the synthesized execution plans, the performance of the best synthesized execution plan versus the *naïve plan* (see Sect. 5.1) and whether the cost function allows the algorithm to differentiate good and bad execution plans (from a performance point of view).

Correctness is the corner stone of the research question. There is no point in constructing control components for QVT Core [3] (QVTc) transformations if they do not result in correct transformations. Correctness on its own, without looking into performance, is paramount as it shows that the use of data dependence analysis is a feasible approach to synthesizing execution plans. Performance, measured against the *naïve plan*, will provide assur-

ance that the cost of the additional time and processing needed for synthesizing execution plans provides significant execution performance gains. Evaluation of the cost function is critical to argue that the cost function of the EPSA is a good differentiator of performance. If it is not, it is not possible claim that the best solution found by the EPSA is a good–enough solution. This chapter also briefly discusses the computational results of the EPSA to show that the algorithm behaves as expected from an Ant Colony Optimization (ACO) algorithm.

The evaluation is done in a set of unseen transformations selected from a group of existing transformations available freely. The set of unseen transformations are required to reduce the threats to internal validity and external validity [91]. Mainly, using the development examples could be seen as using the same test for pre– and post–testing. Hence the test would not tell us anything about the EPSA performance. For external validity, a set of transformations with different characteristics will help us generalize to the theoretical population of all transformations. In the results presentation the *best synthesized execution plan* is a reference to the best solution found by the EPSA, that is, the execution plan with the lowest cost.

The chapter begins in Sect. 6.1 by describing the selection criteria for the evaluation transformations and then Sect. 6.2 to Sect. 6.10 give a small description of each of the selected transformations. Next, Sect. 6.11 discusses the behaviour of the EPSA and presents details of the space exploration for each of the examples. Section 6.12 presents the two experiment designs carried out during evaluation and gives the results for each of the evaluation transformations. Finally, Sect. 6.15 summarizes and concludes the discussion.



(a) Lines of Code (b) Number of rules

Figure 6.1: Distribution of LOC and size for the transformations in the ATL Zoo.

6.1 Evaluation Transformations

From the literature review conducted for this research, it was found that the ATL Zoo[1] seems to be the only publicly available repository of model transformations. Figure 6.1 presents the distribution of the Lines of Code (LOC) and size (measured in number of rules) for all the available examples in the ATL Zoo. The LOC and size information was used to select a representative sample from the Atlas Transformation Language (ATL) Zoo to use as evaluation transformations. Note that each example can be constituted by one or more ATL transformation scripts. The data is taken over the individual scripts, not per example.

The LOC distribution (Fig. 6.1a) shows that most of the scripts have under 500 LOC and that from those, the majority has under 200 LOC. The size (Fig. 6.1b) distribution shows that most of the scripts have under 50 rules and that from those, the majority have under 15 rules. The selected transformation scripts have rules in the range {min=3, max=31} and LOC in the range {min=80,max=411}. These transformations are a representative sample of the most common size and LOC values. Increasing the number of transformations and including transformations with larger sizes and LOC was considered but discarded due to the required effort to translate the transformations from ATL to QVTc

[1]https://www.eclipse.org/atl/atlTransformations/

Table 6.1: Overview of the original validation examples.

Name	Origin	Language	Mappings	Lines of Code
Abstract2Concrete	Eclipse QVTd	QVTr	5	154
Mi2Si	Eclipse QVTd	QVTr	6	93
DNF	Eclipse QVTd	QVTr	9	400
PathExp2PetriNet	ATL Zoo	ATL	3	80
PetriNet2XML	ATL Zoo	ATL	5	210
TextualPathExp2PathExp	ATL Zoo	ATL	12	230
BibTeXML2DocBook	ATL Zoo	ATL	26	466
XSLT2XQuery	ATL Zoo	ATL	31	411
Railway2Control	In House	QVTc	7	168

(which includes rule translation, helper functions translation, definition of the trace model, and debugging) and the similarity of the larger transformations with those already selected.

The evaluation transformations group was augmented with three transformations from the ModelMorf[2] set of examples. In this case, the examples were selected based on the number of rules and perceived complexity from a visual inspection of the code. The complexity is related to the number of input variables of each mapping and the purpose of the transformations. Finally, an *in house* transformation that uses the train benchmark metamodel [102] for the source model was added to the set of evaluation transformations. Given that this metamodel is intended for benchmarking different model management languages we considered it appropriate to include it in the group of evaluation transformations. The selected transformations are presented in Table 6.1, details of size and LOC is given.

Table 6.2 shows the size and LOC information for the evaluation transformations after translated to QVTc. In general, the size remains the same and the LOC decreases. The biggest change is observed for the DNF example, in which the size increased from 9 to 18.

One of the identified limitations of this research was the possible existence of closed loops in the data dependence graph (DDG)

[2]ModelMorf (`http://www.tcs-trddc.com/trddc_website/ModelMorf/ModelMorf.htm`, last accessed 23/03/2017)is a proprietary tool from Tata, which has authorized the use of their test cases in the Eclipse Meta Object Facility (MOF) Query/View/Transformation (QVT)d Project.

Table 6.2: Overview of translated validation examples (sorted by LOC).

Name	Mappings	Lines of Code
Mi2Si	3	64
PathExp2PetriNet	3	80
Abstract2Concrete	5	105
BibTeXML2DocBook	6	150
Railway2Control	7	168
PetriNet2XML	5	200
TextualPathExp2PathExp	12	230
DNF	18	284
XSLT2XQuery	31	350

(see Sect. 4.2.2). This situation was not observed in any of the development or evaluation transformations. Although the sample size is small, it appears that closed loops are not that common. Further, since closed looped can be broken by splitting up the involved mappings their existence does not pose a significant threat to the applicability of the presented approach.

The other important limitation was the static analysis of Object Constraint Language (OCL) expressions in predicates was limited to the pattern: `<var>.<propertry>=<var>`. As with the development transformations this is the most common form of predicate observed in the evaluation transformations. It was also observed that this pattern was often used to verify multiplicity associations via the opposite property. That is, if class A has a reference b to elements of type B with multiplicity *0 to many*, and the opposite property a in type B exits, then elements `a1:A` and `b1:B` these two predicates are equivalent:

```
a1.b->includes(b1);
b1.a = a1;
```

The latter was usually preferred. However, it was also observed that some of the metamodels do not use opposite properties and hence the first type of predicate has to be used. As a result, variable derivation in transformations that reference metamodels that

lack opposite result in the use of more multi-valued-loops. As discussed in Sect. ?? this will have an effect on the performance of the transformations.

The following sections provide the details for each of the examples. The results are presented in Sect. 6.11 and Sect. 6.12. When discussing the metamodels qualified type names will be used in the form `domain::type` (domain: domain name, type: type name) when types in the domains have the same name.

6.2 Abstract to Concrete

The *Abstract to Concrete* transformation is part of the Eclipse QVTd project example transformations and was originally developed by Tata Consultancy as part of the ModelMorf project (an implementation of the QVTr language). The example describes a transformation of a simplified UML model to another simplified UML model. The aim of this transformation is to generate, from a source UML model, a target UML model that flattens the inherited operations of a class. That is, a `Class` in the target model will collect all the operations inherited from the closure of it's super classes that are abstract. Given that the transformation is endogenous, the source and target models have the same structure.

6.2.1 Metamodels

The source and target domains are defined by the simplified UML metamodel presented Fig. 6.2. A UML package is represented by the `Package` element. This element is composed of zero or more `Type` elements, which have a *name* (String) attribute. There are two possible types: `Class` and `PrimitiveDataType`. `Class` elements have an *isAbstract* (boolean) attribute and an *inheritsFrom* reference to another `Class`. The *inheritsFrom* relations defines the super class of a `Class`. A `Class` element is composed of `Operation` elements, which have a *name* (String) attribute. An `Operation`

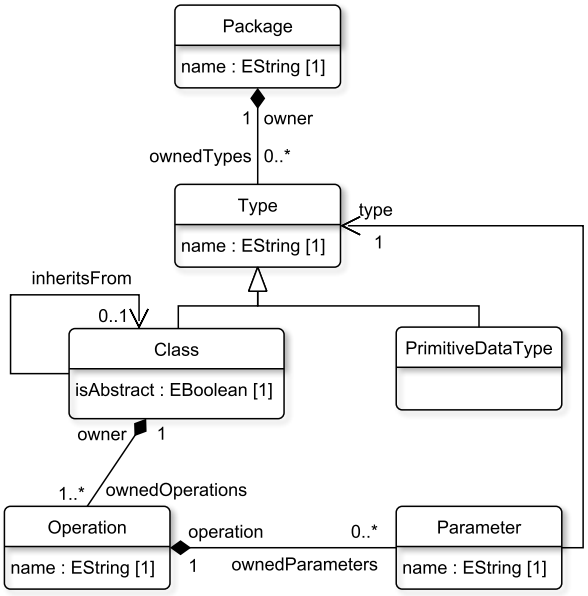


Figure 6.2: Abstract to Concrete class metamodel.

element is composed of `Parameter` elements, which have a *name* (String) attribute.

6.2.2 Mappings Specification

These are the high–level description of the rules to transform the simplified UML model (umlIn) to a flattened simplified UML model (umlOut).

- For each `umlIn::Package` element a `umlOut::Package` element is created. The *name* of the created Package is the same as the one from the source Package.

- For each `umlIn::Class` that `inheritsFrom` an abstract class (i.e. *isAbstract* = True), a `umlOut::Class` is created. The created class has a copy of each of the operations in the abstract class.

- For each `umlIn::PrimitiveDataType` element an equivalent `umlOut::PrimitiveDataType` element is created. The *name*

of the created PrimitiveDataType is the same as the one from the source PrimitiveDataType.

6.2.3 QVTc Code

The QVTc code for the Abstract to Concrete transformation consists of four (4) mappings and one (1) query. The query *subClasses* provides all the subclasses of a `Class` element, given that the metamodel does not provide this information. We only discuss the details of two of the mappings, the complete code can be found in Annex 8.7.

AbstractClassToConcreteClass. This mapping, presented in Listing 6.1, identifies classes from the source model, `umlIn` domain, that inherit from (line 47) an abstract class (line 49). For each of this classes a class is generated in the output model (line 51), *umlOut* domain. Additionally, a trace element is created (line 57) which keeps a reference to the input class (line 59), the abstract class (line 58) and the created class (line 60). The middle domain is used to find the generated package (`pOut:umlOut::Package`) for the input class' owner: lines 45, 55 and 56. This mapping has five input and two output variables. In the input variables two have the same type: `umlIn::Class`.

Listing 6.1: AbstractClassToConcreteClass mapping in Abstract to Concrete example.

```
43 map AbstractClassToConcreteClass in AbstractToConcrete {
44     umlIn (pIn:Package, cc1:Class, ac:Class |
45             cc1.owner = pIn;
46             not cc1.inheritsFrom.oclIsUndefined();
47             cc1.inheritsFrom = ac;
48             ac.owner = pIn;
49             ac.isAbstract; ) { }
50     enforce umlOut (pOut:Package) {
51         realize cc2 : Class |
52         cc2.owner := pOut;
53     }
54     where (p2p:PackageToPackage |
55             p2p.pIn = pIn;
56             p2p.pOut = pOut; ) {
57         realize p2c : ParentToChild |
```

```
58          p2c.parent := ac;
59          p2c.class := cc1;
60          p2c.concreteClass := cc2;
61          p2c.owner := p2p;
62      }
63      map {
64          where() {
65              p2c.name := cc1.name;
66              cc2.name := p2c.name;
67          }
68      }
69  }
```

OperationToOperation. This mapping, presented in Listing 6.2, identifies operations from the source model, `umlIn` domain, that are contained (line 77) in an abstract class (line 73) that is the super class of another class (line 76). For each of this operations an operation is generated in the output model (line 79). Additionally, a trace element is created (line 86) which keeps a reference to the input and output operations, lines 87 and 88. The middle domain is used to find the generated class (`cc2:umlOut::Class`) for the class that inherits from the operation's owner class. The generated operation is added as a child of the output class, line 80. This mapping has five input and two output variables. In the input variables two have the same type: `umlIn::Class`.

The remaining mappings generate the required `Package`, `Parameter` and `PrimitiveDataType` elements.

Listing 6.2: OperationToOperation mapping in Abstract to Concrete example.

```
71  map OperationToOperation in AbstractToConcrete {
72      umlIn (pc : Class, sc:Class, aco:Operation |
73              pc.isAbstract;
74              subClasses(pc)->notEmpty();
75              not sc.inheritsFrom.oclIsUndefined();
76              sc.inheritsFrom = pc;
77              aco.owner = pc; ) { }
78      enforce umlOut (cc2 : Class | ) {
79          realize cco:Operation |
80          cco.owner := cc2;
81      }
82      where (p2c : ParentToChild |
83              p2c.class = sc;
```

```
84              p2c.parent = pc;
85              p2c.concreteClass = cc2; ) {
86          realize o2o : OperationToOperation |
87          o2o.abstract := aco;
88          o2o.concrete := cco;
89          o2o.class := p2c;
90      }
91      map {
92          where() {
93              o2o.name := aco.name;
94              o2o.name := cco.name;
95              aco.name := o2o.name;
96              cco.name := o2o.name;
97          }
98      }
99  }
```

6.3 BibteXML to DocBook

The *BibTeXML to DocBook* example is part of the ATL Zoo. The BibTeXML to DocBook example describes a transformation of a BibTeXML model to a DocBook model. BibTeXML [84] is an XML-based format for the BibTeX bibliographic tool. DocBook [107] is an XML-based format for document composition. The aim of this transformation is to generate, from a BibTeXML model, a DocBook model that presents the bibliographic information in four different sections. The first section provides the full list of bibliographic entries. The second section provides the sorted list of the different authors referenced in the bibliography. The third section presents the titles of the bibliography titled entries. Finally, the last section provides the list of referenced journals. The transformation is exogenous, and the source and target metamodels have very distinct structures. The source metamodel has a very flat structure and most of its classes have multiple inheritance. The target metamodel, on the contrary, has a hierarchical structure with four levels of containment and only three of its classes use inheritance (single).

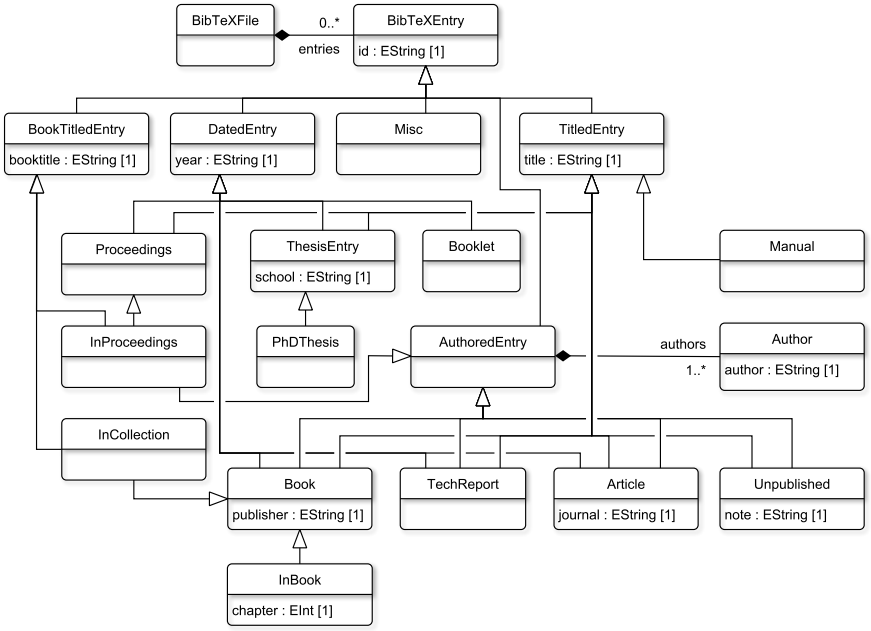


Figure 6.3: BibTeXML metamodel.

6.3.1 Metamodels

The source metamodel is a simplified version of the type structure of the BibTeXML [84] specification. The metamodel only considers the fields defined as mandatory in BibTeX, as presented in Fig. 6.3. A BibTeXML bibliography is modelized by the `BibTeX` element. This element is composed of one or more `BibTexEntry` elements. BibTexEntry has five subclasses to group the major type of bibtex entries. These five classes define the mandatory attributes, such as *year* (`DatedEntry`), *title* (`Titled Entry`), etc. These five subclasses are in turn subclassed to provide the actual bibtex entry types: `Article`, `Proceedings`, `Book`, etc. There are in total 13 possible entry types.

The target metamodel represents a subset if the DocBook [107] specification, as presented in Fig. 6.4. A DocBook is modelized as a `DocBook` element. This element is composed of one or more `Book` elements, which in turn are composed of one or more `Article` elements. An `Article` can have multiple sections and finally each section is composed of `Para` (paragraph) elements that hold the

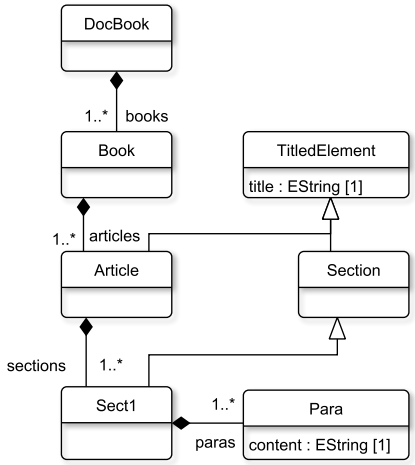


Figure 6.4: DocBook metamodel.

content.

6.3.2 Mappings Specification

These are the high–level description of the rules to transform the BibTexML model to a DocBook model. The transformation can be considered in its whole as a query that collects the information of all BibTex entries (title, authors, year, journal, etc) in a section, and also creates a list of authors, a list of titles, and a list of journals.

- For each `BibTexFile` element a `DocBook` and an `Article` element are created. Additionally, four `Sect1` sections are added to the article, with names *References List*, *Author List*, *Title List* and *Journal List*.

- For each distinct `Author` element a `Paragraph` element is created, with it contents assigned to the author's name.

- For each `BibTexEntry` one, two or three `Paragraph` elements are created. This paragraph contains the complete entry information, the entry title (when applicable) and the entry journal (when applicable).

6.3.3 QVTc Code

The QVTc code for the BibTexML to DocBook transformation consists of six mappings and four queries. The queries *authorSet*, *titleSet* and *journalSet* keep track of the transformed elements to avoid duplicates. The query *buildEntryPara* concatenates the BibTeX entry information into a single string. We only discuss the details of two of the mappings, the complete code can be found in Annex 8.8.

Main. This mapping, presented in Listing 6.3 performs the first part of the transformation, creating the required DocBook sections from a single `BibTeXFile` element. This mapping has one input and eight output variables. Four of the eight output variables have the same type: `Sect1`.

Listing 6.3: Main mapping in BibTeXML to DocBook example.

```
20 map Main in bibtex2docbook {
21     check bibtex(bib:BibTeXFile){}
22     enforce docbook() {
23         realize doc:DocBook, realize book:Book, realize art:Article,
24         realize se1:Sect1, realize se2:Sect1, realize se3:Sect1,
25         realize se4:Sect1 |
26         doc.books := OrderedSet{book};
27         book.articles := OrderedSet{art};
28         art.sections_1 := OrderedSet{se1, se2, se3, se4};
29         se1.title := 'References List';
30         se2.title := 'Authors List';
31         se3.title := 'Titles List';
32         se4.title := 'Journals List';
33     }
34     where () {
35         realize bib2doc:Bib2Doc |
36         bib2doc.file := bib;
37         bib2doc.doc := doc;
38     }
39 }
```

EntryToPara. This mapping, presented in Listing 6.4 performs the transformation of a `BibTeXEntry` to a `Para` that contains all the information of the entry. Note that this mapping extends the *InfoToPara* mapping, which is also presented in the listing. The

corresponding section is found by name (line 60), and the *buildEntryPara* query is invoked to create the paragraph content (line 65). This mapping has one two input and three output variables.

Listing 6.4: Main mapping in BibTeXML to DocBook example.

```
41 map InfoToPara in bibtex2docbook  {
42     enforce docbook(se:Sect1 ) {
43         realize p:Para |
44         se.paras := se.paras->including(p);
45     }
46     where (){        —fb:FromBibtex) {
47         realize e2p:InfoToPara,
48         realize fb:FromBibtex |
49         e2p.para := p;
50     }
51     map {
52         where() {
53             p.content := fb.info;
54         }
55     }
56 }

58 map EntryToPara in bibtex2docbook refines InfoToPara {
59     check bibtex(entry:BibTeXEntry) {}
60     enforce docbook(se.title = 'References List';) {}
61     where () {
62         realize fb:FromEntry |
63         e2p.fromEntry := fb;
64         fb.entry := entry;
65         fb.info := buildEntryPara(entry);
66     }
67 }
```

6.4 Disjunctive Normal Form

The *Disjunctive Normal Form* (DNF) transformation is part of the Eclipse QVTd project example transformations and was originally developed by Tata Consultancy as part of the ModelMorf project (an implementation of the QVTr language). The example describes a transformation to perform simplification/reduction of an arbitrary boolean expression into disjunctive normal form by using DeMorgan's law and the distributive law. The aim of this transformation is to perform the transformations listed in Table 6.3, where . is the conjunction logic operator (And), + is the dis-

Law	Input	Output
Distributive	$a.(b+c)$	$(a.b)+(a.c)$
Distributive	$(b+c).a$	$(b.a)+(c.a)$
DeMorgan	$\neg(a+b)$	$\neg a.\neg b$
DeMorgan	$\neg(a.b)$	$\neg a+\neg b$

Table 6.3: Disjunctive Normal Form transformations.

junction logic operator (Or) and $\neg$ is the negation logic operator (NOT). Note that for conversion to DNF only the OR Distributive law is used. The original transformation is an in-place transformation. Since the version of the QVTc engine we are use does not support this mode of execution, our QVTc translation is not an in-place transformation. This particular transformation highlights the semantic differences between QVTc and QVTr languages that results in QVTc transformations being more verbose. As a result, the QVTc transformation has 26 rules vs. only 9 in QVTr.

6.4.1 Metamodels

The transformation is endogenous, with the source and target domains defined by the boolean expression metamodel in Fig. 6.5. A boolean expression is represented by the `BooleanExpr` element. This element is composed of one (1) or more `Expr` elements, which in turn reference one (1) or more `Expr` elements. The metamodel describes a boolean expression tree similar to what would be constructed by a binary expression tree, with the difference that the metamodel does not restrict `Expr` elements to only have two (2) nested expressions. Expressions can be of type `And`, `Or`, `NOT` and `Literal`.

6.4.2 Mappings Specification

The transformation rules obey the descriptions presented in Table 6.3, and can be expressed in a high level as follows:

- For each `bexp::And` expression that has two nested expres-

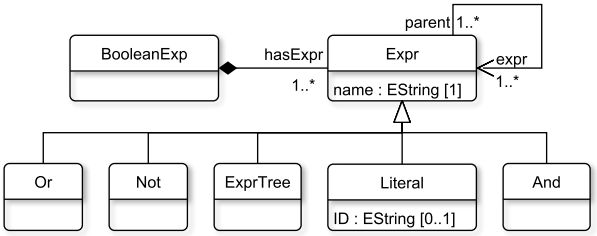


Figure 6.5: Boolean expression metamodel.

sions, one `bexp::Expr` and one `bexp::Or`, a `dnf::Or` expression is generated that has two nested `dnf::And` expressions. Additionally, one `dnf::Expr` expression is generated as the input `bexp::Expr` has to be duplicated.

- For each `bexp::Not` expression that has a nested `bexp::And` or `bexp::Or` expression, an `bexp::Exp` is generated that has two nested `bexp::Not` expressions. The generated expression is an `dnf::Or` if the input nested expression is an `bexp::And`, and a `dnf::And` if the input expression is an `bexp::Or`.
- All other expressions are copied to the output model.

6.4.3 QVTc Code

The QVTc code for the DNF transformation consists of 26 mappings and one (1) query. We only discuss the details of two of the mappings that deal with distributive transformations, the complete code can be found in Annex 8.9. This example is very complex and the complete QVTc code should be consulted to understand how the transformation works.

OrDistribution. This mapping, presented in Listing 6.5, is responsible for the distribution equivalences: $a.(b+c) = (a.b)+(a.c)$ or $(b+c).a = (b.a)+(c.a)$. It identifies `bexp:And` expressions from the source model that are composed of a `bexp:Or` and another expression (lines 162-165). The rule assumes that the a term from the source expression is used to construct the $b.a$ `And`, and hence, a

copy of it must be created in order to construct the $c.a$ expression. Note that this rule is based on the logical and being commutative, that is $c.a = a.c$. The required information for the copy is placed in the trace model, via the `OrDistributionCopy` element. Given that a, b and c are expressions that can also be distributed or split via DeMorgan's law, the trace model (via the `Expr2MultExpr` element) is again used to keep the references of the source expressions (line 173) and it's parent target expression (line 174). The actual assignment of the final expressions is thus delayed until it can be decided if each of the expressions was further transformed. This mapping has three input and 6 output variables.

Listing 6.5: OrDistribution in DNF example.

```
157 map OrDistribution in dnf {
158     check bexp(e1:And, e2:Or, a:Expr |
159         e2 <> a;
160         e1.expr->includes(e2);
161         e1.expr->includes(a); ) {}
162     enforce dnf() {
163         realize e3:Or, realize e4:And, realize e5:And |
164         e3.expr := Sequence{e4, e5};
165     }
166     where( ) {
167         realize or2and: OrDistribution, realize
                a2a_c:OrDistributionCopy,
168         realize disor2ands:Expr2MultExpr  |
169         or2and.beExpr := e1;
170         or2and.dnfExpr := e3;
171         or2and.ID := e1.ID;
172         or2and.beParentExpr := e1.parent;
173         e3.ID := or2and.ID;
174         a2a_c.beExpr := a;
175         a2a_c.dnfAndDist := if e1.expr->indexOf(a) = 1 then e5
176                             else e4
177                             endif;
178         a2a_c.ID := a.ID + '_c';
179         disor2ands.beExpr := e2;
180         disor2ands.dnfExprs := Sequence(exprMM::Not){e4, e5};
181         --disor2ands.ID := e2.ID;
182         e4.ID := e2.ID + '_1';
183         e5.ID := e2.ID + '_2';
184     }
185 }
```

Expr2ExprCopy This mapping, presented in Listing 6.6, is responsible for copying the *a* expressions in an *Or ditribution*, as presented in the previous section. This mapping is intended to be refined by mappings for the specific type of expression, that is, in line 191 we would expect the correct type of expression to be constructed. Finally, since all but the `LiteralExp` can have nested expressions, this rule will result in a recursive invocation until the expression to be copied is a `LiteralExp`. This mapping has two input and two output variables. The recursion is done via the *Expr2ExprCopy_Rec* mapping presented in Listing 6.7.

Listing 6.6: Expr2ExprCopy in DNF example.

```
188 map Expr2ExprCopy in dnf {
189     check bexp(a:Expr | ) { }
190     enforce dnf() {
191         realize adnf:Expr |
192     } where(e2e_c:OrDistributionCopy |
193         e2e_c.beExpr = a;
194     ) {
195         realize e2e_d:Expr2ExprCopy |
196         e2e_d.beExpr := a;
197         e2e_d.dnfExpr := adnf;
198         e2e_d.ID := a.ID + '_c';
199         adnf.ID := e2e_d.ID;
200         adnf.parent := e2e_c.dnfAndDist;
201     }
202 }
```

Listing 6.7: Expr2ExprCopy_Rec in DNF example.

```
223 map Expr2ExprCopy_Rec in dnf  {
224     check bexp(a:Expr, aparent:Expr |
225         aparent.expr->includes(a);
226     ) { }
227     enforce dnf() {
228         realize adnf:Expr |
229     } where(e2ep:Expr2ExprCopy |
230         e2ep.beExpr = aparent;
231     ) {
232         realize e2e_r:Expr2ExprCopy |
233         e2e_r.beExpr := a;
234         e2e_r.dnfExpr := adnf;
235         e2e_r.ID := a.ID + '_rc';
236         adnf.ID := e2e_r.ID;
237         adnf.parent := e2ep.dnfExpr;
238     }
239 }
```

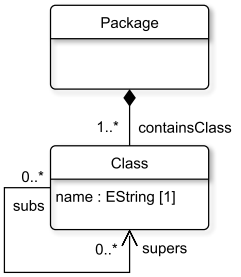
6.5 Multiple Inheritance (Mi) to Single Inheritance(Si)

The *Multiple Inheritance to Single Inheritance* (Mi2Si) transformation is part of the Eclipse QVTd project example transformations and was originally developed by Tata Consultancy as part of the ModelMorf project (an implementation of the QVTr language). The transformation addresses the transformation of an UML Class hierarchy with multiple inheritance to a Java Class hierarchy with single inheritance. The aim of this transformation is to generate, from an UML (simplified) class hierarchy, a Java (simplified) class hierarchy by defining a multiple-inheritance (MI) interface hierarchy corresponding to the MI UML class hierarchy and to establish implementation links between the class hierarchy and the implementation hierarchy. This transformation implements one of several approaches [29, 27] that have been proposed for this problem.

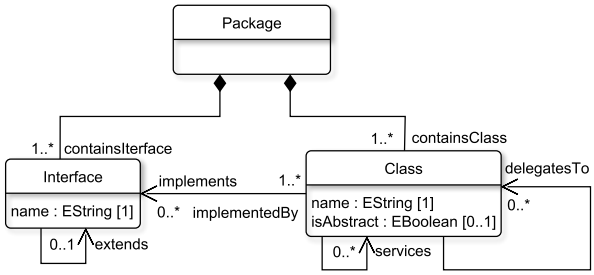
6.5.1 Metamodels

The transformation is exogenous, with the source and target domains defined by the UML and Java metamodels respectively, presented in Fig. 6.5. A UML package is represented by the `Package` element. This element is composed of `Class` elements, which have a *name* (String) attribute. A `Class` has zero or more super-classes (reference *supers*) and zero or more sub-classes (reference *subs*).

A Java package is represented by the `Package` element. This element is composed of `Class` and `Interface` elements. A `Class` has a *name* (String) and an *isAbstract* (boolean) attribute, and *services* and *delagatesTo* references to a `Class`. A `Class` can implement (reference *implements*) zero or more `Interface` elements. An `Interface` has a *name* (String) attribute and can *extend* zero or one interfaces, and can be *implementedBy* one or more classes.



(a) UML metamodel



(b) Java metamodel

Figure 6.6: Meatamodels for the Mi2Si transformation.

6.5.2 Mappings Specification

The transformation rules perform two tasks: transform each UML *Class* into a Java `Class` + `Interface` pair and generate an interface hierarchy (via *implements*) that represents the UML class hierarchy (from *supers*).

- For each `uml::Package` element a `java::Package` element is created.
- For each `uml::Class`, a `java::Class` and a `java::Interface` are created. The *name* of the created Class is the same as the one from the source Class. For the Interface *name* an 'I' is used as a prefix to the name of the source Class.
- For each relation of a `uml::Class` to a super `uml::Class`, the generated `java::Class` will implement the generated `java::Interface` for the super class.

6.5.3 QVTc Code

The QVTc code for the Mi2Si transformation consists of 3 mappings. We only discuss the details of two of the mappings, the complete code can be found in Annex 8.10.

ClassInPackage This mapping, presented in Listing 6.8, transforms all `uml::Class` elements contained in a `uml:Package` (line 37) to a `java::Class` and a `java::Interface`. A trace element is created for each of the generated elements (line 48) which keep references to the input `java::Class` and generated elements. Finally, the name of the `java::Class` is copied from the *name* of the the `name` of the `uml::Class` (line 56–57) and the *name* of the interface is the concatenation of the letter 'I'and the name of the `uml::Class` (line 58). The middle domain is used to place the generated elements in the generated `java::Package` for the `uml::Package` that contains the input class. This mapping has four input and four output variables.

Listing 6.8: ClassInPackage in Mi2Si example.

```
35 map ClassInPackage in Mi2Si {
36   check uml(p1:Package, c1:Class |
37         p1.containsClass->includes(c1);){
38     }
39     enforce java (p2:Package) {
40         realize c2:Class, realize i:Interface |
41         c2.implements := Sequence{i};
42         p2.containsClass := p2.containsClass->including(c2);
43         p2.containsIterface := p2.containsIterface->including(i);
44     }
45     where (p2p:Package2Package |
46     p2p.umlPackage = p1;
47     p2p.javaPackage = p2;) {
48         realize c2c:RClass2Class, realize c2i:Class2Interface |
49         c2i.umlClass := c1;
50         c2i.javaInterface := i;
51         c2c.umlClass := c1;
52         c2c.javaClass := c2;
53     }
54     map {
55         where () {
56             c2c.name := c1.name;
57             c2.name := c2c.name;
58             c2i.name := 'I' + c1.name;
```

```
59              i.name := c2i.name;
60          }
61      }
62  }
```

ClassSuperToImplements This mapping, presented in Listing 6.9, identifies class–superclass relation {`umlc1:uml::Class`, `umlc2:uml::Class`} (line 66) and transforms that hierarchy into an implementation relation, where the generated `java::Class` for *umlc1* implements the generated `java::Interface` for *umlc2* (line 69). The middle domain is used to correctly reference the generated elements. This mapping has 6 input and no output variables.

Listing 6.9: ClassSuperToImplements in Mi2Si example.

```
64 map ClassSuperToImplements in Mi2Si {
65     check uml(umlc1:Class, umlc2:Class |
66         umlc1.supers->includes(umlc2);
67     ) { }
68     enforce java (javac1:Class, javai2:Interface) {
69         javac1.implements := javac1.implements->including(javai2);
70     }
71     where (c2toi:Class2Interface, c12c:Class2Class |
72         c12c.umlClass = umlc1;
73         c12c.javaClass = javac1;
74         c2toi.umlClass = umlc2;
75         c2toi.javaInterface = javai2;
76     ) { }
77 }
```

6.6 Text Path Expression to Path Expression

The *Path Expression to Petri Net* example is part of the ATL Zoo. The original example is actually a three–step transformation sequence that produces an XML Petri net representation (in the PNML format) from a textual definition of a Path Expression. The example is available online from the ATL Zoo website[3], where the

[3] https://www.eclipse.org/atl/atlTransformations/#PathExp2PetriNet, last visited 08-02-2017.

path f;(g;h + k;m*;n);(p+q);s end

Figure 6.7: Textual path expression example.

complete documentation can be found. In this, and the following two Sections (6.7, 6.8) we will provide an overview of each of the three steps of the transformation, focusing on key aspects rather than a complete exposition of the example. The example Path Expression used in the discussion is taken from the ATL example description and as such some of the descriptions are taken directly from the example reference document and added here for completeness. The transformation assumes that a parser exists in order to generate the concrete syntax model from the actual test representation.

The first step of the transformation, *Text Path Expression to Path Expression*, is a concrete syntax to abstract syntax transformation for Path Expressions. This transformation addresses the problem of going from a text representation of a Path Expression to an abstract syntax representation that is amenable to graphical representation. Consider the Path Expression presented in textual form in Fig. 6.7. This Path Expression is composed of a simple transition ("f"), followed by a composed alternative transition ("g;h + k;m*;n"), followed by a simple alternative transition ("p+q"), and a final simple transition ("s"). The Path Expression representation of this expression is presented in Fig. 6.8. Given that the abstract syntax has explicit states (represented by the circles in Fig. 6.8), the aim of this transformation is thus to generate the explicit states from their implicit representation in the concrete syntax.

6.6.1 Metamodels

The transformation is exogenous, with the source and target domains defined by the TextualPathExp and PathExp metamodels respectively, presented in Fig. 6.9. A Textual Path Expression is represented by the `TextualPathExp` element, which is composed

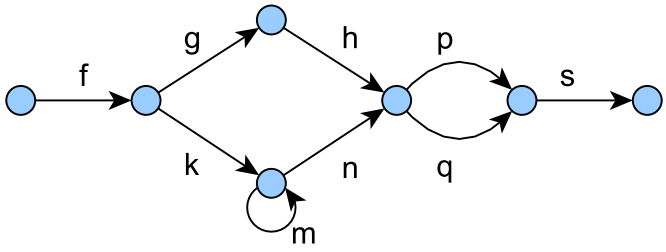


Figure 6.8: Graphical path expression example.

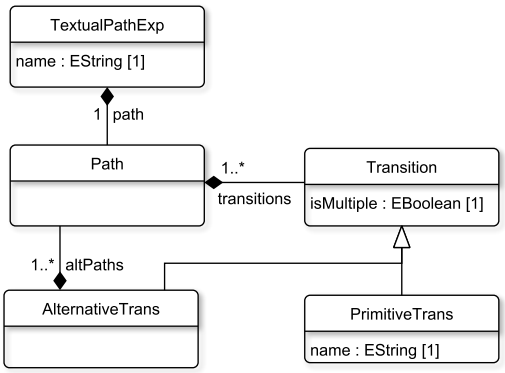
of one `Path` elements. A path contains one or more `Transition` elements. Transitions can be multiple or not, and there are two kind of transitions: `PrimitiveTrans` and `AlternativeTrans`. Alternative transitions are composed of one or more `Path` elements.

A Path Expression is represented by the `PathExp` element, which is composed of zero or more `Transition` elements, and one or more `State` elements. A `State` can have zero or more *outgoing* and *incoming* `Transition` elements. A `Transition` has a *source* and a `target State`. `PathExp` and `Transition` inherit from `Element`, which gives them a *name* (String) attribute.

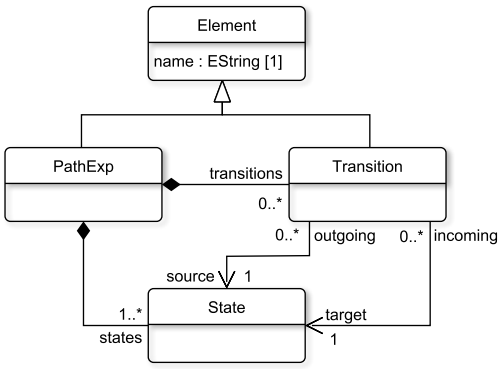
6.6.2 Mappings Specification

The transformation rules perform three tasks: generate the equivalent transitions in the PathExp model (*path* domain), generate the required states in the PathExp model and correctly assign the `State`'s *source* and *target* transitions. The transformation makes the following assumptions:

- `AlternativeTrans` should not be *multiple* (i.e. only simple loops can be defined).
- The first and the last transitions of a `Path`, including the root `Path` (the path in the `TextualPathExp` element), should not be *multiple*.
- The first transition of the input model must be a `PrimitiveTrans`.



(a) Textual Path Expression metamodel



(b) Path Expresion metamodel

Figure 6.9: Metamodels for the TextualPathExp2PathExp transformation.

The transformation can be described as follows:

- For each `PrimitiveTrans` element a `Transition` element is created.
- For each `PrimitiveTrans` that is not multiple and is not the last transition in a Path, a `State` element is created, which is the *target* of the crated transition.
- For each `AlternativeTrans` element a `State` element is created.
- For each `PrimitiveTrans` that is not the last transition in a

Path or the first transition in the root path, the source state is the `State` created for the previous `text::Transition` in the Path, where the previous transition is computed as follows:

  - If the `Transition` is not the first in the path, then the previous is the first non-multiple `Transition` that precedes the transition. E.g. in the example path, for transition h the previous is g and for n is k.
  - If the `Transition` is the first in the path, then the previous is the first non-multiple `Transition` that precedes the `AlternateTrans` that owns the path. E.g. in the example path, for transition k the previous is f, and for q is the `AlternateTrans` that forms the diamond structure.

- For each `PrimitiveTrans` that is the last transition in a Path, the `source State` is the one created for the previous transition (as explained above) and the *target* `State` is the state generated for the `AlternateTrans` that owns the path.
- For the first `PrimitiveTrans` in the root path, an additional `State` is created to be the source of the `Transition`.

6.6.3 QVTc Code

The QVTc code for the TextualPathExp to PathExp transformation consists of 9 mappings and 3 queries. We only discuss the details of two of the mappings, the complete code can be found in Annex 8.12.

SingleTransition. This mapping transforms a `PrimitiveTransition` whose *isMultiple* attribute is False, into a `Transition`, the code is presented in Listing 6.10. Note that this mapping refines *PrimitiveTransition* also presented in the listing. The name of the source transition is used (line 48) to assign the name of the target

transition. This rule also creates the target `State` of the transition. The source cannot be created because it depends on the location of the transition in the path. This mapping has four input and three output variables.

Listing 6.10: PrimitiveTransition in TextualPathExp to PathExp example.

```
33 map PrimitiveTransition in text2model {
34     check text(tpe_pt:PrimitiveTrans) { }
35     enforce path(pe:PathExp) {
36         realize pe_t:Transition |
37         pe_t.PathExp := pe;
38     }
39     where (tpe2pe:TextPath2Path |
40             tpe2pe.pathexp = pe; ) {
41         realize t2t:PrimitiveTrans2SubPath |
42         t2t.textTrans := tpe_pt;
43         t2t.pathTrans := pe_t;
44     }
45     map {
46         where() {
47             t2t.name := tpe_pt.name;
48             pe_t.name := t2t.name;
49         }
50     }
51 }

53 -- A PrimitiveTrans that is not multiple generates a state,
54 -- additionally to the transition. The state is the target of the
55 -- transition
56 map SingleTransition in text2model refines PrimitiveTransition {
57     check text(not tpe_pt.isMultiple;) { }
58     enforce path() {
59         realize pe_ts:State |
60         pe_ts.PathExp := pe;
61         pe_t.target := pe_ts;
62     }
63     where() {
64         t2t.state := pe_ts;
65     }
66 }
```

TransitionSource. This mapping, presented in Listing 6.11, is responsible for assigning the source state for a `Transformation` created from a `PrimitiveTransition`. The source `State` is resolved by using the trace. The `t2t_prev:Tansition2Transition` will point to the correct transition, which is created either from

an Alternative transition (if `tpe_pt:PrimitiveTrans` is the last transition in an alternative path) or a Primitive transition (if `tpe_pt:PrimitiveTrans` is not the last transition in a path.) This mapping has five input and no output variables.

Listing 6.11: TransitionSource in TextualPathExp to PathExp example.

```
104 map TransitionSource in text2model {
105     check text(tpe_pt:PrimitiveTrans) { }
106     enforce path(pe_t:Transition, pe_s:State) {
107         pe_t.source := pe_s;
108     }
109     where(t2t:PrimitiveTrans2SubPath, t2t_prev:Tansition2Transition |
110             t2t.textTrans = tpe_pt;
111             t2t.prevTrans = t2t_prev.textTrans;
112             t2t.pathTrans = pe_t;
113             t2t_prev.state = pe_s; ) { }
114 }
```

6.7 Path Expression to Petri Net

The second step of the *Path Expression to Petri Net* example, is a transformation from the PathExp domain to the PetriNet domain. The main difference between these two domains is that in the latter the arcs that connect places/states and transitions are modelled. This transformation addresses the problem of representing the arcs. The similarities and small changes can be observed in Fig. 6.10. This figure is almost the same as Fig. 6.8, except that in this case the transition is represented by a bar, as opposed to an arc.

6.7.1 Metamodels

The transformation is exogenous, with the source and target domains defined by the PathExp (pt) and PetriNet (pn) metamodels respectively, with the latter presented in Fig. 6.11. A PetriNet is represented by the `PetriNet` element, which is composed of one `Place`, `Transition` and `Arc` elements. A `Place` has a *target* `TransToPlaceArc` and a source `PlaceToTransArc`. A `Transition`

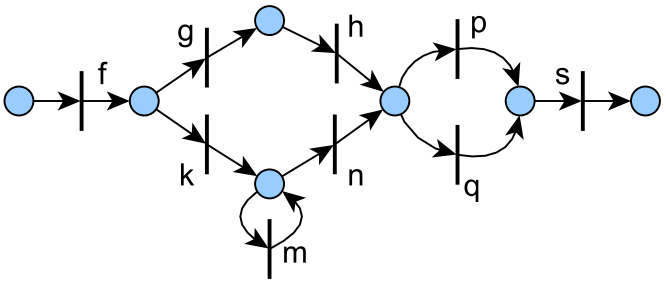


Figure 6.10: Petri Net example.

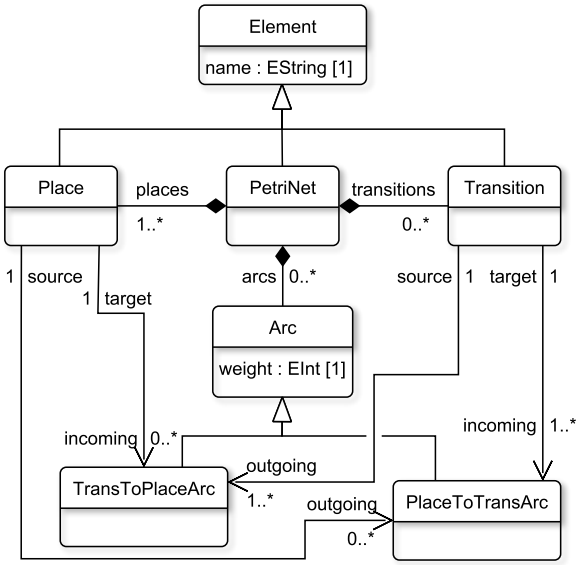


Figure 6.11: Petri Net metamodel.

has a source `TransToPlaceArc` and a target `PlaceToTransArc`. `Place`, `Transition` and `PetriNet` elements have a *name* attribute.

6.7.2 Mappings Specification

The transformation performs a major task: transform a transition in the PathExp to the required Transition and Arcs in the PetriNet model. For each `PathExp` element a `PetriNet` element is created. Similarly, for each `State` a `Place` element is created. Finally, for each `pt:Transition` a `pn:Transtion`, a `TransToPlaceArc`

and a `PlaceToTransArc` elements are created. The arc use the `pt:Transition` *source* and *target* information to reference the respective *incoming* and *outgoing* `Place` elements, respectively.

6.7.3 QVTc Code

The QVTc code for the Path Expression to Petri Net transformation consists of three mappings. We only discuss the details of one of the mappings, the complete code can be found in Annex 8.7.

Transition2Transition. This mapping, presented in Listing 6.12, is responsible for creating a `Transition`, a `TransToPlaceArc` and a `PlaceToTransArc`. Then, the correct references are set by using the trace model to resolve the required places (lines 58–64). This mapping has nine input variables and six output variables.

Listing 6.12: Transition2Transition in PathExp to Petri Net.

```
46 map Transition2Transition in path2petri {
47     check path(pt:Transition, pe:PathExp, ss:State, ts:State |
48         pe.transitions->includes(pt);
49         pt.source = ss;
50         pt.target = ts;
51     ) { }
52     enforce petri(pn:PetriNet, sp:Place, tp:Place) {
53         realize nt:Transition, realize pn_ia:PlaceToTransArc,
54         realize pn_oa:TransToPlaceArc |
55             pn.transitions := pn.transitions->including(nt);
56             pn.arcs := pn.arcs->including(pn_ia);
57             pn.arcs := pn.arcs->including(pn_oa);
58             pn_ia.weight := 1;
59             pn_ia.source := sp;
60             pn_ia.target := nt;
61             pn_oa.weight := 1;
62             pn_oa.source := nt;
63             pn_oa.target := tp;
64     }
65     where (pe2pn:PathExp2PetriNet, ss2sp:State2Place,
           ts2tp:State2Place |
66         pe2pn.pathexp = pe;
67         pe2pn.petrinet = pn;
68         ss2sp.state = ss;
69         ss2sp.place = sp;
70         ts2tp.state = ts;
71         ts2tp.place = tp;
72     ) {
```

```
73          realize t2t:Trans2Trans, realize t2tInc:Trans2InArc,
74          realize t2tOut:Trans2OutArc  |
75          t2t.pathTrans := pt;
76          t2t.netTrans := nt;
77          t2tInc.source := ss2sp;
78          t2tInc.target := t2t;
79          t2tOut.source := t2t;
80          t2tOut.target := ts2tp;
81          t2t.name := pt.name;
82          nt.name := t2t.name;
83      }
84  }
```

6.8 Petri Net to PNML(XML)

The third, and final, step of the *Path Expression to Petri Net* example, is a transformation from the PetriNet domain to the XML domain. The XML model will provide an XML representation of the PetriNet in PNML format[4]. The main responsibility of this transformation is to correctly generate identification numbers (id) for each of the petri net elements, as cross-references in PNML are based on id values. The transformation exploits the fact that containers in the PetriNet metamodel are ordered, and hence the position of an element in its container is used to provide this information. The transition is, as with the PathExp to PetriNet (Sect. 6.7), straight forward, with one mapping to transform elements of each of the PetriNet classes into it's XML representation. Note that the used XML metamodel is a simplified version of the XML specification.

6.8.1 Metamodels

The transformation is exogenous, with the source and target domains defined by the PetriNet and XML metamodels respectively, with the latter presented in Fig. 6.11. A PNML document is represented as a `Root` element, which is composed of 0 (zero) or more *Node* elements. A `Node` has a *value* and *name* attributes that are

[4] `http://www.pnml.org`. last accessed 10/05/2017

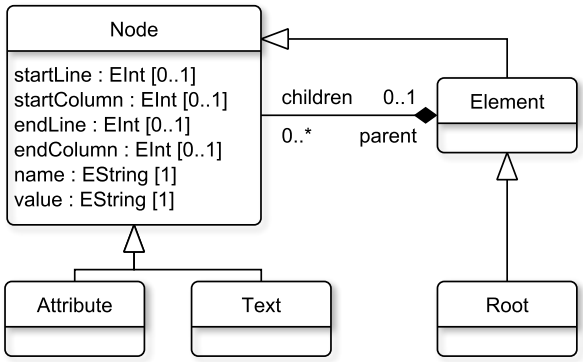


Figure 6.12: PNML (XML) metamodel.

used to store information about the node. A `Node` can be of three types: `Attribute`, `Text` or `Element`. `Element` elements are used to model parent-child relationships, with the *name* attribute use to encode the name of the modelled XML tag.

6.8.2 Mappings Specification

The transformation constructs an XML `Element` for each element in the Petri Net. This element is composed of one or more `Attribute` and `Element` elements. When creating representations for `Place`, `Transition` and *Arc* elements, the position in the collection that contains the element is used to provide an id. To avoid duplicates, the `Transition` id's are offset by the total number of places, and the *Arc* id's are offset by the total number of places and transitions.

6.8.3 QVTc Code

The QVTc code for the Petri Net to PNML(XML) transformation consists of five mappings. There is a separate mapping for `TransToPlaceArc` and for `PlaceToTransArc` elements. We only discuss the details of one of the mappings, the complete code can be found in Annex 8.7.

PlaceToTransArc. This mapping, presented in Listing 6.13, is responsible for creating an `Element`, and three `Attribute` elements to represent a `PlaceToTransArc`. As mentioned previously, the element's id is calculated using the position information (line 162–164). Likewise, the source and target references are found by using the same logic to calculate the id's for the source `Place` and target `Transition` (lines 166 and 168). This mapping has three input and five output variables.

Listing 6.13: PlaceToTransArc in Petri Net to PNML(XML).

```
135 map PlaceToTransArc in petri2xml {
136     check petri(pn:PetriNet) {
137         pn_a:PlaceToTransArc |
138         pn.arcs->includes(pn_a);
139     }
140     enforce xml(net:Element) {
141         realize xml_arc:Element,
142         realize id:Attribute,
143         realize source:Attribute,
144         realize target:Attribute |
145         xml_arc.name := 'arc';
146         xml_arc.children := Sequence{id, source, target};
147         id.name := 'id';
148         net.children := net.children->including(xml_arc);
149     }
150     where(pn2root:PetriNet2Root |
151             pn2root.petriNet = pn;
152             pn2root.net = net;) {
153         realize p2a:Place2TransArc |
154         p2a.pn_a := pn_a;
155         p2a.xml_arc := xml_arc;
156         p2a.id := id;
157         p2a.source := source;
158         p2a.target := target;
159     }
160     map {
161         where() {
162             id.value := pn.places->size().toString() +
163                         pn.transitions->size().toString() +
164                         pn.arcs->indexOf(pn_a).toString();
165             source.name := 'source';
166             source.value :=
                    pn.places->indexOf(pn_a.source).toString();
167             target.name := 'target';
168             target.value := pn.places->size().toString() +
169                         pn.transitions->indexOf(pn_a.target).toString();
170         }
171     }
172 }
```

6.9 Railway to Control

This transformation was developed as part of this research. The example is based on the train benchmark metamodel [102], which contains the most typical EMF model constructs. The transformation considers that the train domain can be used model a

toy model train, and we are interested in automating the signals (semaphores) and switches that control the toy model train behaviour. The transformation is used to create a model of a control system that can be used to monitor the state of the toy model train and, for example, automate the toy railway to allow multiple trains to run on it.

6.9.1 Metamodels

The transformation is exogenous, with the source and target domains are defined by the TrainBenchmark and Control metamodels respectively, presented in Fig. 6.13 and Fig. 6.14. A railway is represented by a `RailwayContainer` element. This element is composed of zero or more `Region` elements, and zero or more `Route` elements. A `Region` represents a section of the railway composed of zero or more `Track` elements. A `Track` element can either be a `Switch` (connecting other track elements) or a `Segment` element with a given *lenght* and a set of *neighbor* elements. A `Route` (i.e. a course that can be followed by a train) is composed of a set of `SwitchPosition` elements, i.e. by following the switches in the given positions the course can be followed. A `Route` has an *entry* and *exit* `Semaphore`.

The control is represented as a `RailwayControl` element. This element is composed of `Route` and `SemaphoreSignal` elements. A `Route` is composed of `TrackSection` and `SwitchSignal` elements. The control can also have *Train* elements that follow a *plan* made up of zero or more *Semaphore signals*. Finally, a `Route` element can join or fork into other routes to allow route composition.

6.9.2 Mappings Specification

These are the high–level description of the rules to transform a Train Benchmark model (rail) to Control model (control).

- For each `rail::Route` element a `control::Route` element is created.

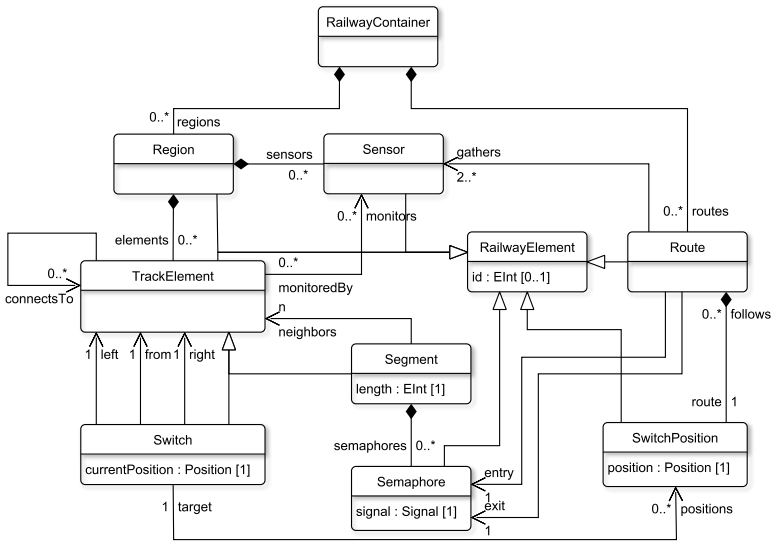


Figure 6.13: Railway metamodel.

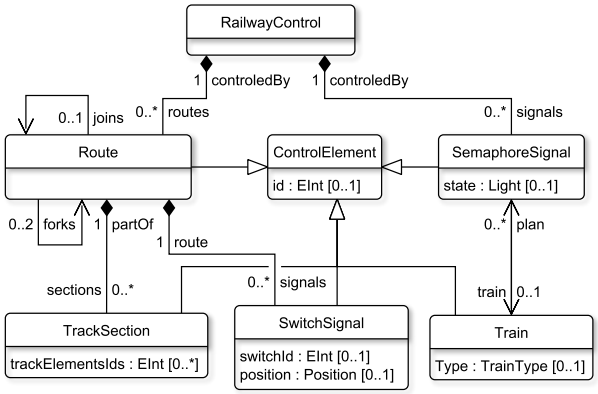


Figure 6.14: Control metamodel.

- For each `rail::Semaphore` used in a `rail::Route`, a `control::SemaphoreSignal` is created.
- For each `rail::Sensor` a `control::TrackSection` is created. The track section is made up of all the `rail::TrackElements` monitored by the sensor (tracked by id).
- For each `rail:SwitchPosition` a `control::SwitchSignal` is created.
- Finally, the `rail::Semaphore` is also used to correctly cre-

ate the *fork* and *join* information of each `control::Route` element.

6.9.3 QVTc Code

The QVTc code for the Railway to Control transformation consists of seven mappings and one query. The query *getSensorElementsIds* collects the *id* information of all the `rail::TrackElement` elements monitored by a `rail::Sensor`. We only discuss the details of two of the mappings, the complete code can be found in Annex 8.14.

Semaphore2signal. This mapping, presented in Listing 6.14, is responsible for transforming a `rail::Semaphore` into a `control::SemaphoreSignal`. The semaphore has to be part of the segments that compose a region: predicates in lines 51–53. The created `control::SemaphoreSignal` is allocated to the corresponding `RailwayControl`, line 57. This mapping has six input and two output variables.

Listing 6.14: Mapping semaphore2signal in the Railway to Control example.

```
49 map semaphore2signal in r2c {
50     check rail (rc:RailwayContainer, r:Region, seg:Segment,
           s:Semaphore |
51         rc.regions->includes(r);
52         r.elements->includes(seg);
53         seg.semaphores->includes(s);
54         ) { }
55     enforce control(c:RailwayControl) {
56         realize ss:SemaphoreSignal |
57         ss.controledBy := c;
58     }
59     where (r2c:Railway2Control |
60         r2c.railway = r;
61         r2c.control = c;
62     ) {
63         realize s2s:Semaphore2Signal |
64         s2s.semaphore := s;
65         s2s.signal := ss;
66     }
67     map {
68         where() {
```

```
69              s2s.id := s.id;
70              s2s.id := ss.id;
71              s.id := s2s.id;
72              s.id := s2s.id;
73              ss.state :=
74                  if s.signal = railway::Signal::FAILURE then
75                      control::Light::RED
76                  else
77                      if s.signal = railway::Signal::STOP then
78                          control::Light::RED
79                      else
80                          control::Light::GREEN
81                      endif
82                  endif;
83          }
84      }
85  }
```

Semaphore2fork. This mapping, presented in Listing 6.15, is responsible for assigning the *fork* relations of a `control::Route`. It uses the information in the `rail::Semaphore` to match the Control elements to the `rail::Routes` that are connected via the `rail::Semaphore`, using the *exit* and *entry* information (lines 166–171). This mapping has 13 input and no output variables.

Listing 6.15: Mapping semaphore2signal in the Railway to Control example.

```
164 map semaphore2fork in r2c {
165     check rail (r:RailwayContainer, sr:Route, fr1:Route, fr2:Route,
            s:Semaphore |
166         r.routes->includes(sr);
167         r.routes->includes(fr1);
168         r.routes->includes(fr2);
169         sr.exit = s;
170         fr1.entry = s;
171         fr2.entry = s;) {

173     }
174     enforce control(c:RailwayControl, sc:Route, fc1:Route,
            fc2:Route) {
175         sc.forks := Sequence{fc1, fc2};
176     }
177     where (r2c:Railway2Control, er2c:Route2Route,
178             fr2c1:Route2Route, fr2c2:Route2Route |
179             r2c.railway = r;
180             r2c.control = c;
181             er2c.railwayRoute = sr;
```

```
182             er2c.controlRoute = sc;
183             fr2c1.railwayRoute = fr1;
184             fr2c1.controlRoute = fc1;
185             fr2c2.railwayRoute = fr2;
186             fr2c2.controlRoute = fc2;

188     ) { }
189 }
```

6.10 XSLT to XQuery

The *XSLT to XQuery* example is part of the ATL Zoo, available online from the ATL Zoo website[5], describes a simplified transformation of XSLT code to XQuery code [16]. XSL (eXtensible Stylesheet Language) is a styling language for XML, and XSLT stands for XSL Transformations. XSLT can be used to transform XML documents into other formats, such as HTML. XQuery is a language designed to query XML documents in order to find and extract elements and attributes. The transformation aims at representing the XSLT as a series of XQueries. We provide an overview of the domains involved and a general description of the transformation. Further details can be found in the ATL Zoo website.

6.10.1 Metamodels

The source domain is defined by a simplified XSLT metamodel, presented in Fig. 6.15, where not all the available constructs have been modelled. For example the *for-each*, and *copy-of* constructs have been omitted. Consequently, this constructs are not taken into consideration in the transformation. An XSLT is represented by the `XSLTNode`, which allow construction of the transformation by composing other `XSLTNode` and (plain XML) `Node` elements. The `XSLTNode` is subclassed in order to provide the XSLT constructs such as `Choose`, `Sort` and `If`.

[5] https://www.eclipse.org/atl/atlTransformations/#XSLT2XQuery, last visited 08-02-2017.

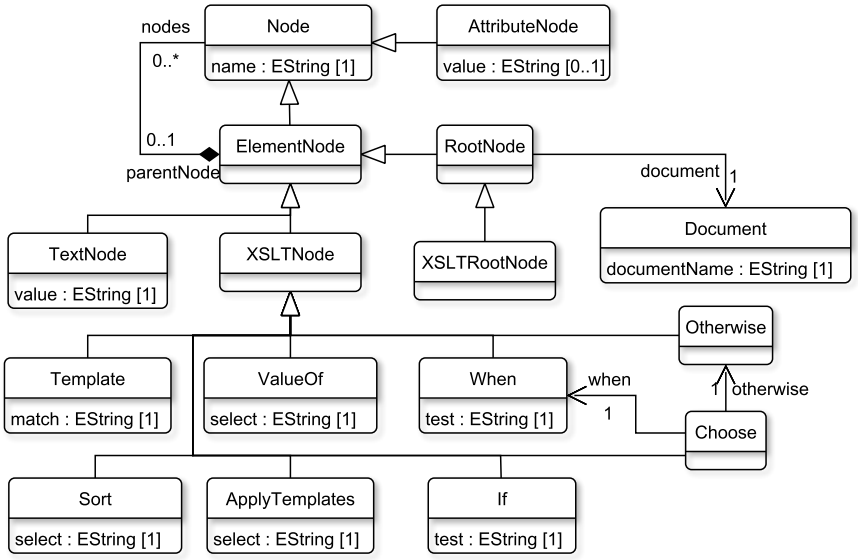


Figure 6.15: XSLT metamodel.

The target domain is defined by the XQuery metamodel presented in Fig. 6.16. An XQuery program is represented by the `XQueryProgram` element, which is composed of zero or more `ExecutableExpression` elements. `ExecutableExpression` can be `FLWOR` expressions, function calls (`FunctionCall`) and function declarations (`FunctionDeclaration`). The main expressions in XQuery are modelled by the `FLWOR` expression, which is composed of `OrderBy`, `Let`, `For`, `Where` and `Return` elements (expressions). `FLOWR` expresions are composed of other expressions, either `XPath` or `Boolean`. The XQuery metamodel also defines elements that belong to the XML domain: `Node`. `ElementNode`, `TextNode`, etc.

6.10.2 Mappings Specification

The transformation rules perform three tasks: create an `XQueryProgram` from an `XSLTRootNode`, transform `XSLT` elements into `XQuery` expressions and copy the required XML elements and attributes.

The transformation can be described as follows:

- For each XSLT `Template` an XQuery `FunctionDeclaration`

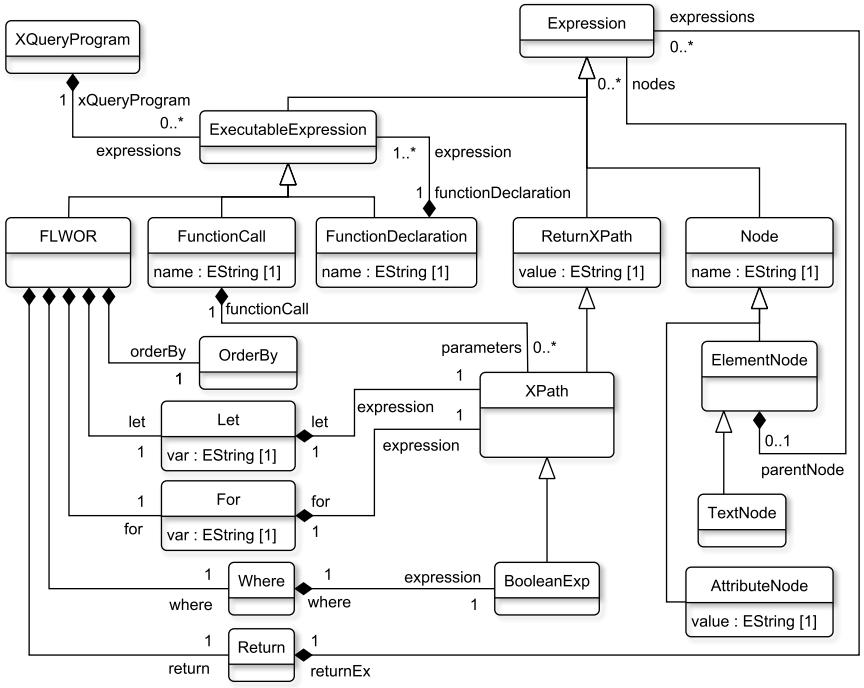


Figure 6.16: XQuery metamodel.

is created. The function declaration is assigned the required *parameters* and `Return` expression. A `FLOWR` and `For` elements are created to represent the template.

- XSLT `If` elements are transformed into a `FLWOR` expression composed of a `Where` and `Return` expressions to represent the *if* logic.

- XSLT Elements and Attributes are copied from the XSLT model to the XQuery model.

6.10.3 QVTc Code

The QVTc code for the XSLT to XQuery transformation consists of 31 mappings. We only discuss the details of two of the mappings, the complete code can be found in Annex 8.12. In general, there are 4–5 mappings for each of the major transformations: `If` to

`FLOWR`, `Template` to `FunctionCall`, and `Element` and `Attribute` copy.

If2FLOWR. This mapping, presented in Listing 6.16, is the base mapping for `If` to `FLOWR` transformations. It creates the required XQuery elements (lines 114–119) and the trace elements. One of the rules that refines this mapping, *If2FLOWR_Top* is also presented in the listing. In that mapping, the `If` node is identified and the correct expression for the `Where` expression is created, line 143.

Listing 6.16: If2FLOWR mapping in XSLT to XQuery example.

```
112 map If2FLOWR in XSLT2XQuery refines FromNode {
113     enforce xq() {
114         realize out:FLWOR,
115         realize varlet:Let,
116         realize letExpression:XPath,
117         realize _where:Where,
118         realize whereExpression:BooleanExp,
119         realize return:Return |
120         out._let := varlet;
121         out._where := _where;
122         out.return := return;
123         varlet.expression := letExpression;
124         varlet.var := '$var';
125         letExpression.value := '$var';
126         _where.expression := whereExpression;
127     }
128     where() {
129         realize fn:If2FLOWR |
130         fn.exp := out;
131         fn.varlet := varlet;
132         fn.letExpression := letExpression;
133         fn._where := _where;
134         fn.whereExpression := whereExpression;
135     }
136 }

138 map If2FLOWR_Top in XSLT2XQuery refines If2FLOWR {
139     xs(node: If |
140         node.parentNode.oclIsTypeOf(XSLT::Template); ) { }
141     where( ) {
142       fn.node := node;
143       whereExpression.value := '$var/' + node.test;
144     }
145 }
```

ApplyTemplate2FunctionCall. This mapping, presented in Listing 6.17, creates a `FunctionCall` from an `ApplyTemplates` element. One of the rules that refines this mapping, *ApplyTemplate2FunctionCall_Top* is also presented in the listing. In that mapping, the value of the function's parameter is assigned (line 200).

Listing 6.17: If2FLOWR mapping in XSLT to XQuery example.

```
179 map ApplyTemplate2FunctionCall in XSLT2XQuery refines FromNode {
180         xs() { }
181         enforce xq() {
182     realize out:FunctionCall,
183     realize parameter:XPath |
184       out.parameters := Sequence{ parameter };
185   }
186   where() {
187     realize fn:ApplyTemplate2FunctionCall |
188     fn.out := out;
189     fn.parameter := parameter;
190   }
191 }

193 map ApplyTemplate2FunctionCall_Top in XSLT2XQuery refines
      ApplyTemplate2FunctionCall {
194     xs(node: ApplyTemplates |
195         node.parentNode.oclIsTypeOf(XSLT::Template);
196     ) { }
197     where( ) {
198       fn.node := node;
199       out.name := 'fct' + node.select;
200       parameter.value := '$var/' + node.select;
201     }
202 }
```

6.11 Execution Plan Synthesis Algorithm Evaluation

This section presents the evaluation results of the EPSA. These results provide information regarding the space exploration aspect of the EPSA.

6.11.1 Method

Similarly to the method used in Sect. 5.6.7, we captured the details of the execution of the EPSA. These details include:

- The best plan, if any, found at each iteration.
- The cost of the best solution.
- The plan of the best solution.
- The iteration number at which the best solution was found.
- The time at which the best iteration was found.
- The total number of iterations.
- The cost of each of the plans found at each iteration.
- The id (hash) of each of the plans found at each iteration. Given that plans with different structure can have the same cost, we are interested in also being able to differentiate between plans of the same cost.
- The success of the exploration, i.e. a solution can be found.

For each transformation, the EPSA algorithm synthesized execution plans for a total of 25 trials. All trials where done in the same computer (i5 processor, 8gb RAM, Windows 10, Java 8), and the results of each run logged after completion.

6.11.2 Results

The results of the execution of the EPSA are summarized in Table 6.4. The table presents, for each transformation, the cost of the best execution plan found, the average iteration (numbered from 0) at which the best execution plan was found, the average time to find the best execution plan and the average maximum total number of iterations (i.e. after how many iterations the algorithm

Table 6.4: Comparison of the EPSA on the validation examples[a].

Example	Best	i_{avg}	t_{avg}(ms)	i_{max}	t_{max}(s)	ε_{avg} (%)
Abstract2Concrete	53	0	0.038	237	8.59	0
BibTeXML2DocBook	31	2	0.46	237	29.3	37.5
DNF	56	104	1424	270	4921	38.9
Mi2Si	33	0	0.012	219	3.00	0
TextualPathExp2PathExp[b]	107	64.9	28.3	306	150	99.2
PathExp2PetriNet	69	0	0.014	219	3.41	0
PetriNet2XML	100	118	370	237	46442	2.26
Railway2Control	271	0	0.15	246	34.6	0
XSLT2XQuery	–	–	–	–	–	–

[a] Given are the example name, the best solution cost, the average number of iterations i_{avg} (numbered from 0), the average time t_{avg} in seconds to find the best solution in an iteration, the maximum number of iterations i_{max}, the maximum time t_{max} in seconds taken to find a solution and the average error rate ε_{avg}. Averages are taken over 25 trials.

[b] A solution was found only in 18 of the 25 trials.

stopped). Finally, the maximum total execution time of the algorithm (i.e. the total execution time of the longest execution) and average error rate. The average error rate measures how many of the iterations failed to produce an execution plan (see Sect. 5.6.7).

Of notable importance is the lack of results for the XSLT2XQuery transformation. In the 25 trials, the EPSA was not able to synthesize an execution plan for this transformation. To have more insight into this particular transformation the time limit for plan construction was disabled and the EPSA executed again. However, even without the time constraint a solution could not be found.

Figure Fig. 6.17 presents the details of the cost of the best solution found in each iteration. The examples are labelled by position as they appear in Table 6.4. In six of the examples the best cost plan reported in Table 6.4 is the best cost plan found in each of the iterations. In examples DNF and PetriNet2XML the best cost plan reported is the minimum of the found best plans. Further, in both cases the data is skewed to the top, indicating that for these examples a single run would be less likely to return the best over-all execution plan found. However, the range in both cases is small, indicating that a low cost solution would be found.

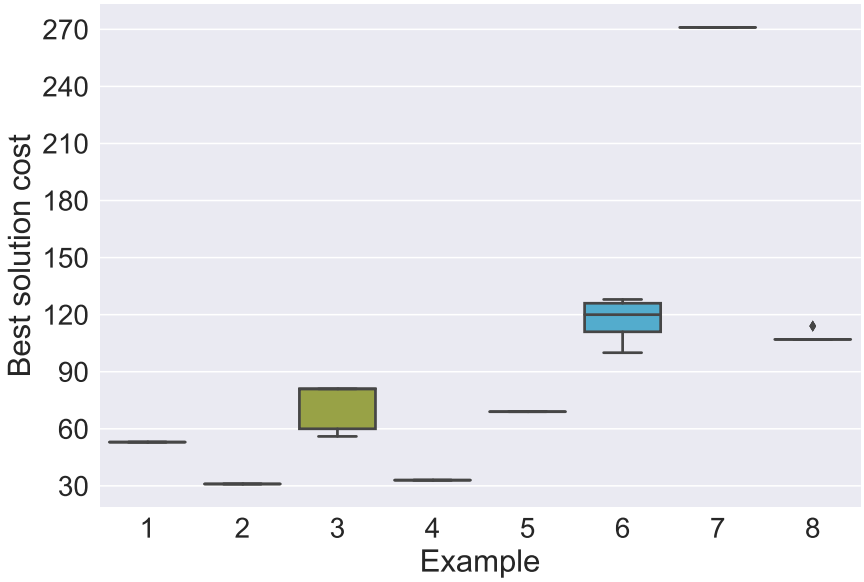


Figure 6.17: Cost of the best solution found at each iteration.

In five of the examples the best solution is found within the third iteration of the algorithm. For the rest of the examples, the best solution is found after around 64, 104 or 118 iterations. For the selected transformations, there seems to be no intermediate result, that is, the best solution is found very fast or much slower (2 orders of magnitude more iterations are needed). It is important to note that the transformations for which finding a solution takes longer have large sizes and high LOC.

However, the table reveals that for the examples that require more iterations, the time to arrive to a solution is not proportional to the number of iterations. For the `DNF` example finding a solution takes considerably longer (three orders of magnitude) than for the `TextualPathExp2PathExp` and `PetriNet2XML` examples. The total number of iterations ($M = 245, SD = 24.8$) suggests that the required iterations for the algorithm to converge to the best solution is also dependant on the particular transformation (recall from Sect. 5.3.6 that the termination condition is based on the convergence of cost of the explored solutions).

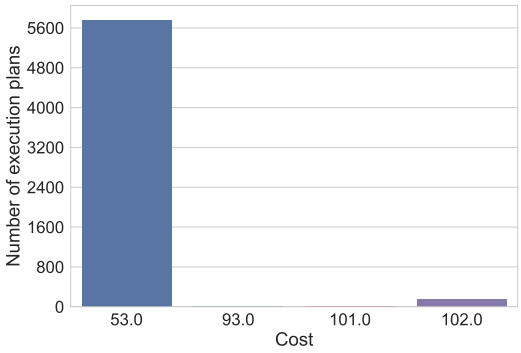
Next, this section presents an overview of the space exploration

for each of the evaluation transformations. The summary presents the plans-per-cost found during one of the trials of the experiment. The information is presented in two plots. The first plot is the distribution of the total number of execution plans per cost and the second plot is the distribution of the distinct number of execution plans per cost. The distributions are used to discuss the behaviour of the EPSA with respect to intensification and diversification.

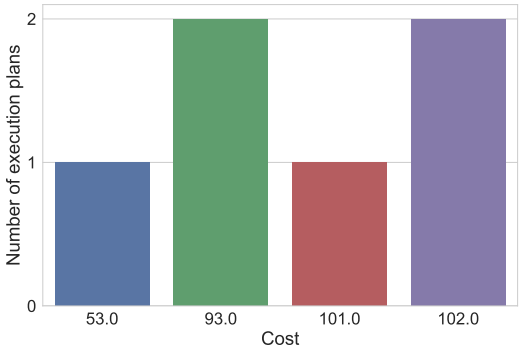
The total plans-per-cost distribution is expected to be skewed right, indicating that the EPSA is more likely to find solutions of lower cost. It is also expected that the total plans-per-cost distribution presents a peak near or on the best cost found, indicating that the EPSA intensifies the search around plans with the best cost. The number of different costs tells little about diversification given that execution plans of different structure can have the same cost. The distinct plans-per-cost distribution provides additional information to discuss diversification. Higher number of alternatives indicates that the EPSA does not always constructs the same execution plans for a given cost, and hence is evidence of diversification. If the peak in the total plans-per-cost distribution coincides with a high number of distinct plans, then the EPSA is balanced. That is, although a lot of solutions where found around an optimal cost (intensification), the solutions are distinct (diversification).

For plans with large number of alternatives found, the x-axis (plan cost) does not show labels for all the costs to avoid cluttering. However, the x-axis is a categorical axis and hence the numerical value of the cost is not correctly represented by the position along the axis.

Abstract2Concrete. In the presented trial, the cost of the solutions ranged from 53.0 to 102 ($M = 54.3, SD = 7.85$). The total plans–per–cost distribution, presented in Fig. 6.18a, has the expected shape. The EPSA synthesized 6 distinct plans in total, and the distinct plans-per-cost distribution presented in Fig. 6.18b suggests that, in this trial, the EPSA was unbalanced, favouring



(a) Total execution plans per Cost.



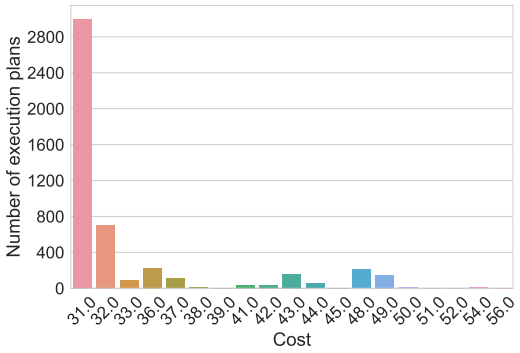
(b) Distinct execution plans per Cost.

Figure 6.18: EPSA results for the Abstract2Concrete example.

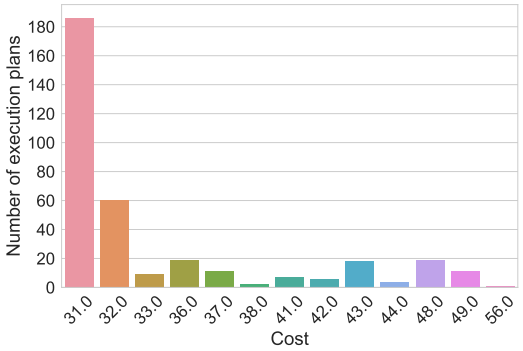
intensification.

BibTeXML2DocBook. In the presented trial, the cost of the solutions ranged from 31.0 to 56.0 ($M = 33.7, SD = 5.31$). The total plans–per–cost distribution, presented in Fig. 6.19a, has the expected shape. The EPSA synthesized 413 distinct plans in total and the distinct plans-per-cost distribution presented in Fig. 6.19b suggests that, in this trial, the EPSA was balanced, with good intensification and diversification.

DNF. In the presented trial, the cost of the solutions ranged from 59.0 to 161 ($M = 85.6, SD = 11.5$). The total plans–per–cost distribution, presented in Fig. 6.20a, has the expected shape. However, note that the peak is more pronounced, with most of the



(a) Total execution plans per Cost.

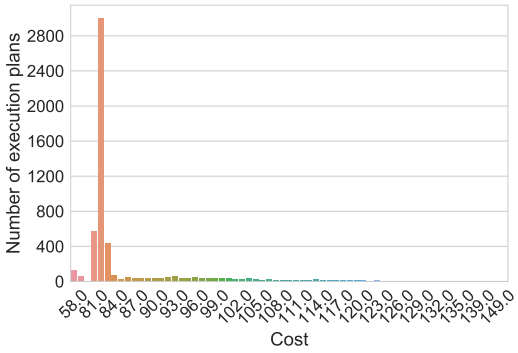


(b) Distinct execution plans per Cost.

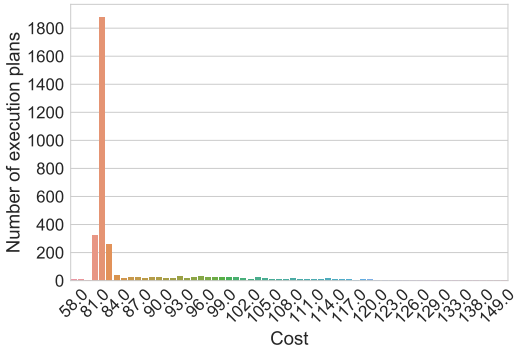
Figure 6.19: EPSA results for the BibTeXML2DocBook example.

number of plans of other cost being orders of magnitude smaller than for the most frequent cost. The EPSA synthesized 3574 distinct plans in total and the distinct plans-per-cost distribution presented in Fig. 6.20b suggests that, in this trial, the EPSA was balanced, with good intensification and diversification. Note that in this trial the best solution is only found until the last 15 iterations which explains the low number of plans of this cost.

Mi2Si. In the presented trial, only two alternative solutions were found, with costs 33.0 and 44.0. The total plans–per–cost, presented in Fig. 6.21a suggests that, in this trial, the EPSA favoured intensification.

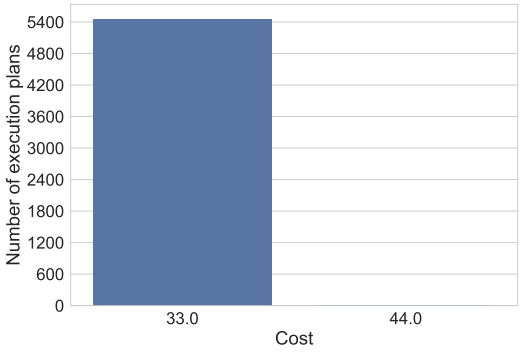


(a) Total execution plans per Cost.



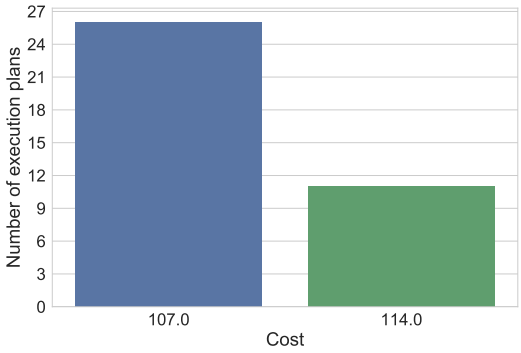
(b) Distinct execution plans per Cost.

Figure 6.20: EPSA results for the DNF example.



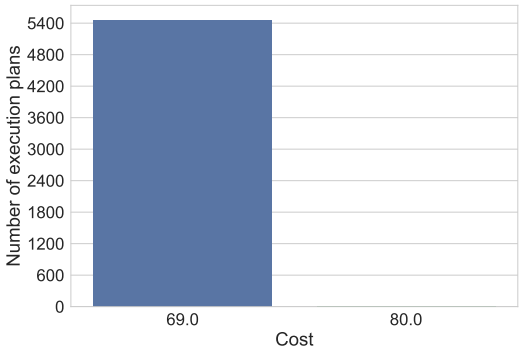
(a) Total execution plans per Cost.

Figure 6.21: EPSA results for the Mi2Si example.



(a) Total execution plans per Cost.

Figure 6.22: EPSA results for the TextualPathExp2PathExp example.

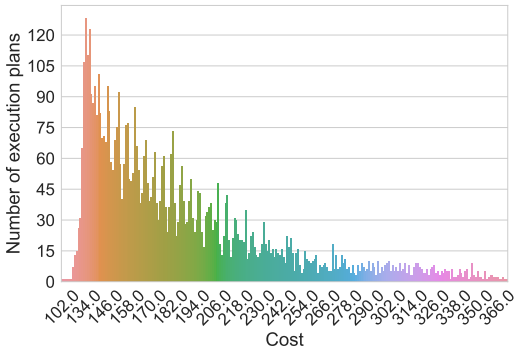


(a) Total execution plans per Cost.

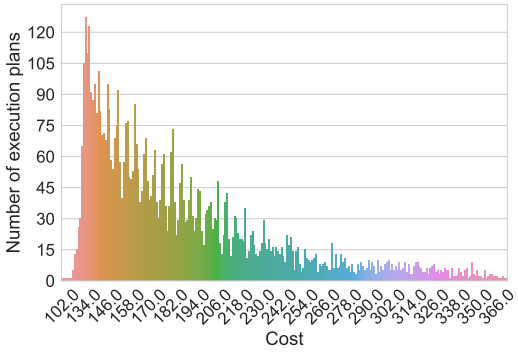
Figure 6.23: EPSA results for the PathExp2PetriNet example.

TextualPathExp2PathExp. In the presented trial, only two alternative solutions were found, with costs 107 and 114. The total plans–per–cost distribution, presented in Fig. 6.22a suggests that, for this trial, the EPSA was balanced, with good intensification and diversification. The reason for this is that the plans of cost 114 represented almost a third of the total number of plans. There is only one plan per cost, i.e. no distinct plans

PathExp2PetriNet. In the presented trial, only two alternative solutions were found, with costs 69 and 80. The total plans–per–cost distribution, presented in Fig. 6.23a suggests that, for this trial, the EPSA favoured intensification. There is only one



(a) Total Execution plans per Cost.

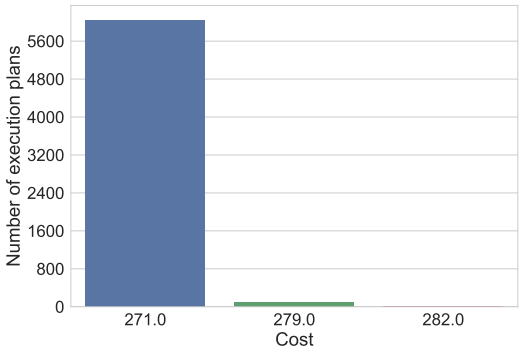


(b) Distinct Execution plans per Cost.

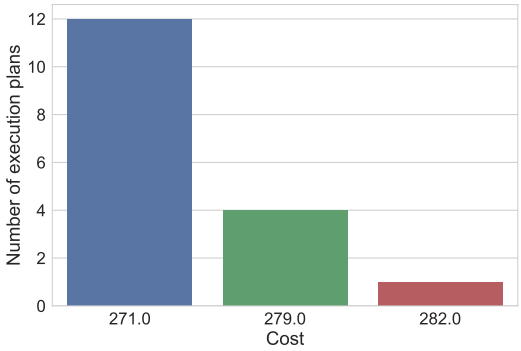
Figure 6.24: EPSA results for the PetriNet2XML.

plan per cost, i.e. no distinct plans

PetriNet2XML. In the presented trial, the cost of the solutions ranged from 102 to 366 ($M = 186, SD = 51.1$). The total plans–per–cost distribution, presented in Fig. 6.24a, has the expected shape. However, note that the peak is not at the best cost found. The EPSA synthesized 5826 distinct plans in total and the distinct plans-per-cost distribution presented in Fig. 6.24b suggests that, in this trial, the EPSA was balanced, with good intensification and diversification. A closer look at the distributions reveals that they are almost identical. This indicates that each unique plan was found a very few times, which in turn is evidence of a very high diversification.



(a) Total Execution plans per Cost.



(b) Distinct Execution plans per Cost.

Figure 6.25: EPSA results for the Railway to Control example.

Railway2Control. In the presented trial, the cost of the solutions ranged from 271 to 282 ($M = 271, SD = 1.02$). The total plans–per–cost distribution, presented in Fig. 6.25a, has the expected shape. The EPSA synthesized 22 distinct plans in total and Fig. 6.25b suggests that, in this trial, the EPSA was balanced.

6.12 Correctness, Performance and Cost Function Evaluation

The evaluation was done over three criteria, one for each of the research hypotheses:

1. Whether the execution of synthesized execution plans result in correct transformations (Hypothesis 1).

2. Whether the performance of the best synthesized execution plan is better than the performance of the *naïve plan* (Hypothesis 2).

3. Whether the performance of the best synthesized execution plan is amongst the best performing solutions (Hypothesis 3).

For the first criterion the synthesized plans were executed using a set of source models of similar size but varying structure. To build enough confidence on the ability of the EPSA to synthesize execution plans that result in correct transformations a total of 30 distinct plans are used to evaluate each transformation. The size of the source models was selected to facilitate manual inspection of the models. Varying the structure of the source model reduces the risk of the synthesized execution plans to only execute correctly for certain models.

The 30 models for each transformation are randomly generated using a semi-automated model generation strategy developed using the EMG tool [83]. The generation script for each transformation is designed to have a balanced number of elements of each type in the source metamodel (the scripts can be found in Annex 9). Further, the values/multiplicity of attributes/associations (or the lack of values) is also randomized based on the conditions defined in the transformation's guard patterns. The goal is that the models have both elements that satisfy and do not satisfy the guard patterns.

For evaluation of the first criterion, the *naïve plan* (see Sect. 5.1) is used as an oracle, that is, the models generated by the synthesized plans will be compared against the plans generated by the *naïve plan*. For the transformations originally from the ATL Zoo, the generated models will also be compared to the models generated from the execution of the original ATL transformation. It is expected that for all transformations for all source models, the target models are identical to those generated by the *naïve plan* (and the original ATL transformation when available).

For the second and third criteria the synthesized plans were executed using one source model with a large number of elements. Using a model with a large number of elements is desirable to evaluate performance, as it highlights the effects of loops and invocations that have NA results. It is expected that the best synthesized execution plan has better performance than the *naïve plan*. Also, it is expected that best synthesized execution plans has a performance similar to the best performing solutions. For this criteria, the results of ANOVA and Tukey HSD tests are presented in order to show the location of the best solution, performance wise, amongst all the solutions.

6.12.1 Method

The evaluation will be performed in the test transformations introduced previously, except for the XSLT2XQuery transformation for which no valid synthesized plans where found. Given that the possible number of synthesized plans is significant, during plan synthesis a random sample of all the plans constructed by the algorithm was taken. Additionally, given that the cost function is being evaluated, the sample was taken based on the cost, that is, the number of samples depends on the total number of distinct costs. Further, for some transformations (like the Railway2Control) although only a few different costs appear in the results, for some of the costs a high number of alternative plans of that cost are found. In this case, several plans of the same cost are included in the sample.

Initially the sample size is limited to 25 execution plans. This limit was set in order to limit the total time needed to run the experiments. The sample was taken from a single trial given that in production the synthesis would only be executed once. The sample was taken from the trial used in the overview of the space exploration in Sect. 6.11.2. During execution, after each iteration a random solution of one of the ants is added to the sample, if the sample size is not reached. Plans are stored in a set using their id,

Table 6.5: Alternative plans for each example.

Example	Sample Size	Min Cost	Max Cost
Abstract2Concrete	6	53	102
BibTeXML2DocBook	17	31	56
DNF	25	81	251
Mi2Si	2	33	44
TextualPathExp2PathExp	6	141	148
PathExp2PetriNet	2	69	80
PetriNet2XML	12	61	364
Railway2Control	8	271	282

thus the same execution plan cannot be added twice to the sample. However, plans with the same cost are allowed. At the end of the EPSA execution an execution plan with the best cost is added to the sample, if not present. Table 6.5 summarizes the sample execution plan information. Note that for the transformations for which few cost alternatives where synthesized the sample size is smaller.

For each transformation two sets of experiments are conducted. The first experiment is used to look at the first criterion. The second experiment is used to look at the second and third criteria.

In the first experiment, correctness of the target model from the synthesized plans is measured by comparing it to the oracle model (or models where available) and by manual inspection. On average, the source models have 20 elements.

For each transformation the procedure was as follows. The *naïve plan* was executed for the 30 source models and the result target models saved to be used as oracles. If the transformation was translated from ATL, the original ATL transformation was executed for the 30 source models and the result target models saved to be used as oracles. The best synthesized execution plan was executed for the 30 source models and the target models saved. An initial evaluation was done using EMF Compare[6] to compare each of the models to the corresponding oracle (and ATL oracle).

[6]EMF Compare tool, the Eclipse Foundation. `http://www.eclipse.org/modeling/emft/?project=compare`.

A final evaluation was done using visual inspection of the models, comparing their structure and the value of the model elements' attributes. All trials where done in the same computer (i7 processor, 8gb RAM, windows 10, Java 8), and the results of each run logged after completion. For the ATL the EMFTVM [106] was used for execution of the transformations. The process was repeated for all the other synthesized execution plans.

In the second experiment, a repeated measures design was used. There was one independent variable: the cost of the execution plan (from the cost function). There was one dependant measure that was analysed: the execution time of the transformation (for a given model). As mentioned previously the size of the models (for each example) should be large in order to highlight the structural differences of the synthesized execution plans. However, due to time constrains the sizes were selected in order to keep the execution time within 20 minutes. The time constraint on the execution time is placed in order to limit the time needed to complete the experiments (worst case: 1 transformation $\times$ 20 minutes $\times$ 25 trials $\times$ 25 samples = 8 days). Execution of each plan was done over 25 trials. All trials where done in the same computer (i7 processor, 8GB RAM, Windows 10, Java 8), and the results of each run logged after completion.

6.12.2 Correctness Results

For the first experiment, the comparison of the generated models for all synthesized execution plans vs. the generated models from the *naïve plan* showed no differences, for all the evaluation transformations. For the evaluation transformations with an ATL version, comparison against the generated models from the original ATL transformation execution also showed no differences. Manual inspection of the target plans also confirmed their correctness, confirming the model comparison results. These results indicate that the synthesized execution plans result in correct transformation execution for all the evaluation transformations.

6.12.3 Performance and Cost Results

This section presents the results of the second experiment. Analysis of the second criterion are presented first, followed by analysis of the third criterion. In all figures the synthesized plans will be labelled with their respective cost and 'naive' will be used to label the *naïve plan*. Given that multiple execution plans can have the same cost, a suffix of the form `_n` is used to differentiate execution plans of the same cost.

Abstract2Concrete. For the second experiment the source model for this transformation has 42000 elements. The results in order to evaluate the second criterion are presented in Fig. 6.26. The graph reveals that the execution time distribution of the *naïve plan* is higher than the best execution plan, which is supported by the numerical values. The median execution time for the best execution plan (128988) is lower than for the *naïve plan* (134420). In fact, it should be noted that the third quartile of the best execution plan's distribution (131887) is lower than the median execution time for the *naïve plan*. We therefore conclude that in general, the best execution plan has a better performance than the *naïve plan*.

Analysis of variance showed a main effect of the plan type on the performance for the plans, $F(1, 48) = 430$, $p < .001$, $\eta_{\mathrm{p}^2} = 0.90$. Further, the results of a post-hoc Tukey HSD ($\alpha = 0.05$) test show that the execution times of both plans are significantly different. These results support the observations from Fig. 6.26.

The results to evaluate the third criterion are presented in Fig. 6.27. The figure shows a scatter plot of the execution times of the synthesized execution plans over the 25 trials. The black line marks the median value of the measurements. Analysis of variance showed that there is no statistically significant effect of cost on the performance for the execution plans, $F(5, 144) = 1.20$, $p = 0.31$, $\eta_{\mathrm{p}^2} = 0.040$. Although the results were not statistically significant, they indicate that the best execution plan has the best

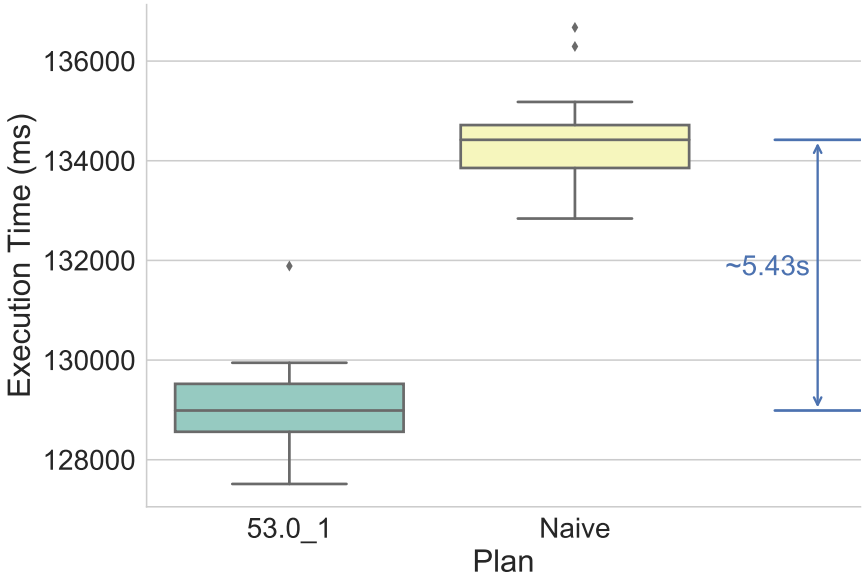


Figure 6.26: Execution time of the best execution plan vs. the *naïve plan* for the Abstract2Concrete example.

performance (on average).

BibTeXML2DocBook. For the second experiment the source model for this transformation has 609 elements. The results in order to evaluate the second criterion are presented in Fig. 6.28. The graph reveals that the execution time distribution of the *naïve plan* is higher than the best execution plan, which is supported by the numerical values. The median execution time for the best execution plan (104642) is lower than for the *naïve plan* (238933). In fact, it should be noted that the third quartile of the best execution plan's distribution (104517) is lower than the median execution time for the *naïve plan*. We therefore conclude that in general, the best execution plan has a better performance than the *naïve plan*.

Analysis of variance showed a main effect of the plan type on the performance for the plans, $F(1, 51) = 28.3$, $p < .001$, $\eta_{p^2} = 0.36$. Further, the results of a post-hoc Tukey HSD ($\alpha = 0.05$) test show that the execution times of both plans are significantly

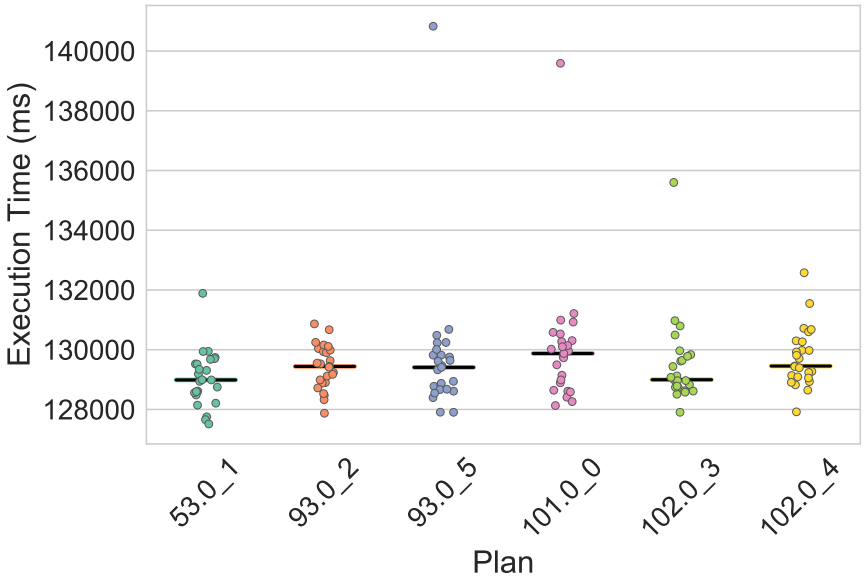


Figure 6.27: Execution time of the synthesized execution plans for the Abstract2Concrete transformation.

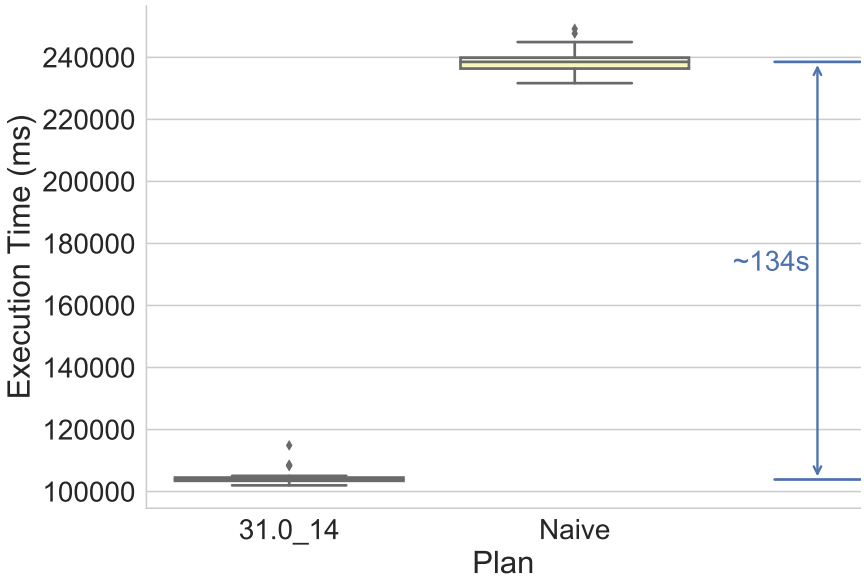


Figure 6.28: Execution time of the best execution plan vs. the *naïve plan* for the BibTeXML2DocBook transformation.

different. These results support the observations from Fig. 6.28.

The results to evaluate the third criterion are presented in Fig. 6.29. The figure shows a scatter plot of the execution times of the synthesized execution plans over the 25 trials. The black line marks the median value of the measurements. Analysis of variance showed a main effect of cost on the performance for the execution plans, $F(15, 430) = 995$, $p = .000$, $\eta_{p^2} = 0.97$. In the figure it can be observed that the execution plans can be grouped into four groups based on the median performance. A Tukey HSD ($\alpha = 0.05$) post-doc test supports this observation:

Group a: 31.0_14, 31.0_21, 32.0_12, 41.0_15, 42.0_1, 43.0_10, 46.0_19, 47.0_3.

Group b: 33.0_22, 37.0_11, 43.0_2, 50.0_9.

Group c: 36.0_13.

Group d: 52.0_0, 54.0_6, 56.0_5.

These results indicate that the best solution, although not being in the best performing group, has a performance amongst the best for the transformation.

DNF. For the second experiment the source model for this transformation has 309 elements. The results in order to evaluate the second criterion are presented in Fig. 6.30. The graph reveals that the execution time distribution of the *naïve plan* is higher than the best execution plan, which is supported by the numerical values. The median execution time for the best execution plan (8062) is lower than for the *naïve plan* (783658). In fact, it should be noted that the third quartile of the best execution plan's distribution (8015) is lower than the median execution time for the *naïve plan*. We therefore conclude that in general, the best execution plan has a better performance than the *naïve plan*.

Analysis of variance showed a main effect of the plan type on the performance for the plans, $F(1, 48) = 824$, $p = .000$, $\eta_{p^2} =$

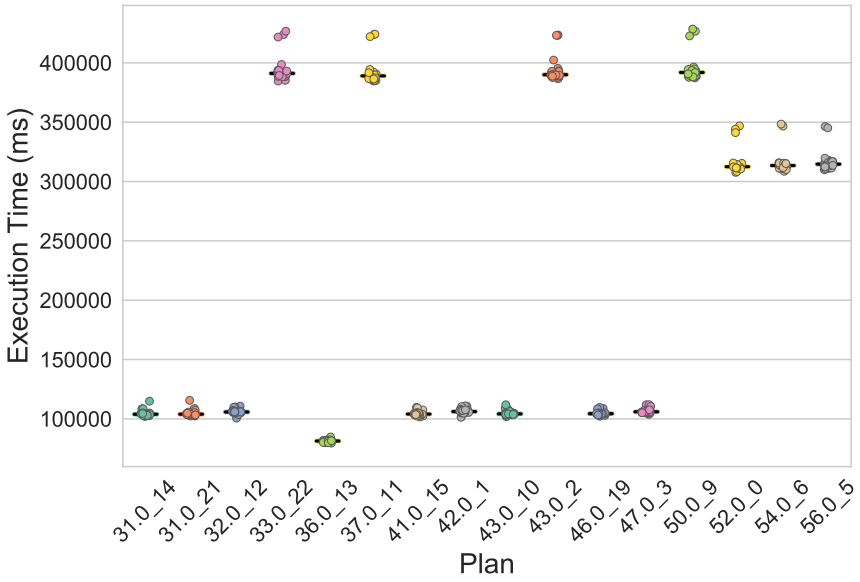


Figure 6.29: Execution time of the synthesized execution plans for the BibTeXML2DocBook example.

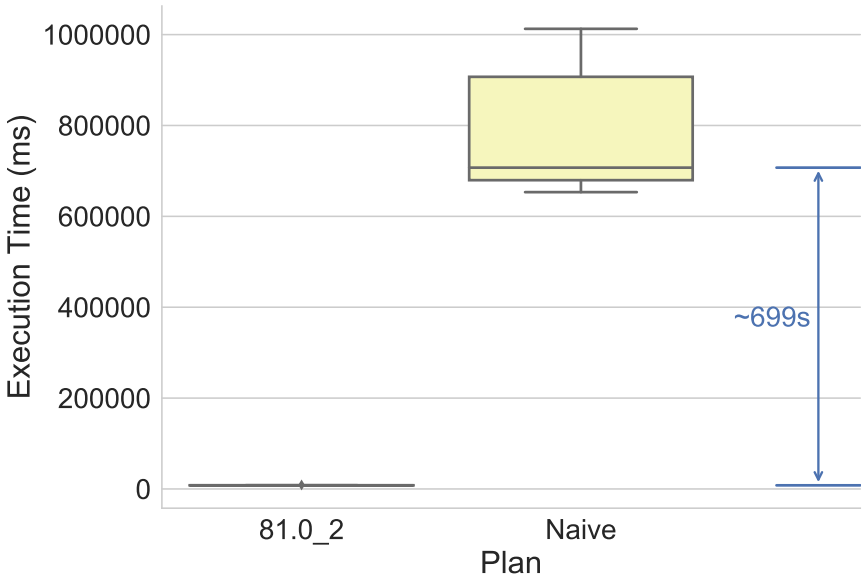


Figure 6.30: Execution time of the best execution plan vs. the *naïve plan* for the DNF transformation.

0.94. Further, the results of a post-hoc Tukey HSD ($\alpha = 0.05$) test show that the execution times of both plans are significantly different. These results support the observations from Fig. 6.30.

The results in order to evaluate the third criterion are presented in Fig. 6.31. The figure shows a scatter plot of the execution times of the synthesized execution plans over the 25 trials. The black line marks the median value of the measurements. Analysis of variance showed a main effect of cost on the performance for the execution plans, $F(12, 312) = 89842$, $p < .001$, $\eta_{\mathrm{p}^2} = 1.00$. In the figure it can be observed that the execution plans can be grouped into three groups based on the median performance. A Tukey HSD ($\alpha = 0.05$) post-doc test supports this observation, but evidences the existence of four groups:

Group a: 101.0_3, 121.0_11, 156.0_6, 81.0_2, 82.0_10, 82.0_14, 83.0_9, 89.0_24, 90.0_20, 91.0_16.

Group b: 145.0_8.

Group c: 147.0_12.

Group d: 160.0_17.

These results indicate that the best solution has a performance amongst the best for the transformation.

Mi2Si. For the second experiment the source model for this transformation has 93 elements. The results in order to evaluate the second criterion are presented in Fig. 6.32. The graph reveals that the execution time distribution of the *naïve plan* is higher than the best execution plan, which is supported by the numerical values. The median age for the best execution plan (391573) is lower than for the *naïve plan* (744785). In fact, it should be noted that the third quartile of the best execution plan's distribution (393297) is lower than the median execution time for the *naïve plan*. We therefore conclude that in general, the best execution plan has a better performance than the *naïve plan*.

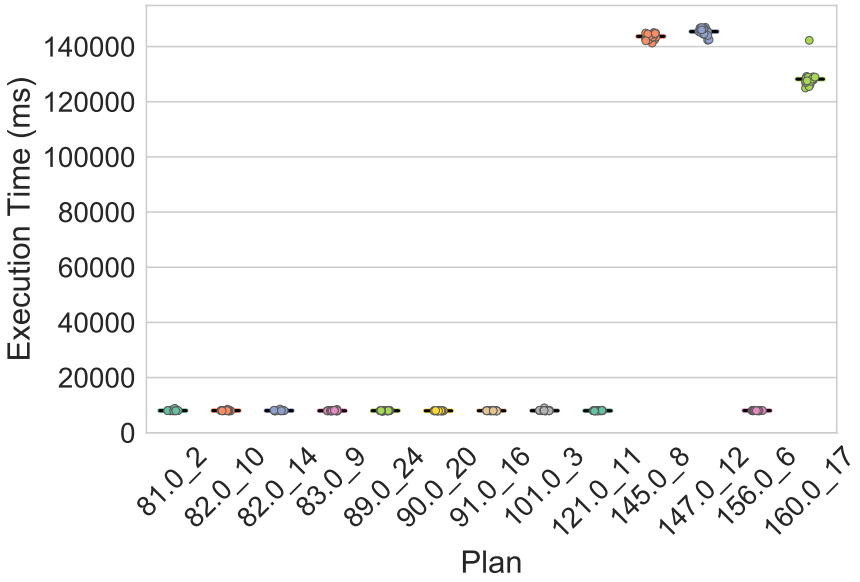


Figure 6.31: Execution time of the synthesized execution plans for the DNF transformation.

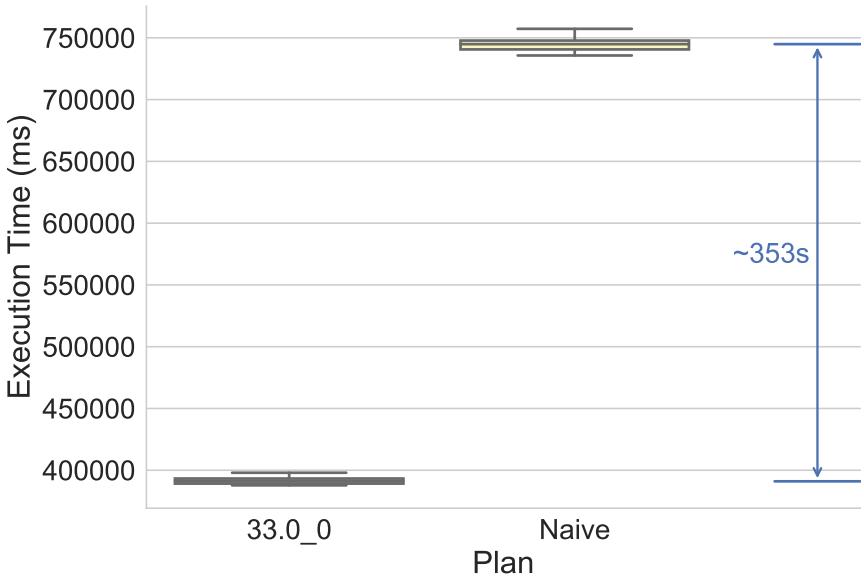


Figure 6.32: Execution time of the best execution plan vs. the *naïve plan* for the Mi2Si transformation.

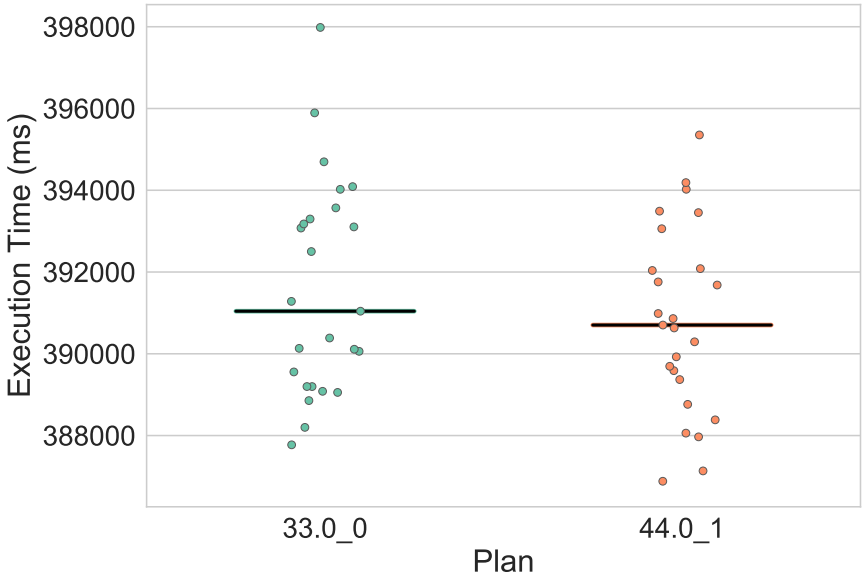


Figure 6.33: Execution time of the synthesized execution plans for the Mi2Si transformation.

Analysis of variance showed a main effect of the plan type on the performance for the plans, $F(1, 48) = 87114$, $p = .000$, $\eta_{p^2} = 1.00$. Further, the results of a post-hoc Tukey HSD ($\alpha = 0.05$) test show that the execution times of both plans are significantly different. These results support the observations from Fig. 6.32.

The results to evaluate the third criterion are presented in Fig. 6.33. The figure shows a scatter plot of the execution times of the synthesized execution plans over the 25 trials. The black line marks the median value of the measurements. synthesized the best solution is the best performing too. Analysis of variance showed that there is no statistically significant effect of cost on the performance for the execution plans, $F(1, 48) = 1.18$, $p = 0.28$, $\eta_{p^2} = 0.024$. Given than only two alternative execution plans were synthesized, the best solution exhibits the best performance.

TextualPathExp2PathExp. For the second experiment the source model for this transformation has 37 elements. The results in order to evaluate the second criterion are presented in Fig. 6.34.

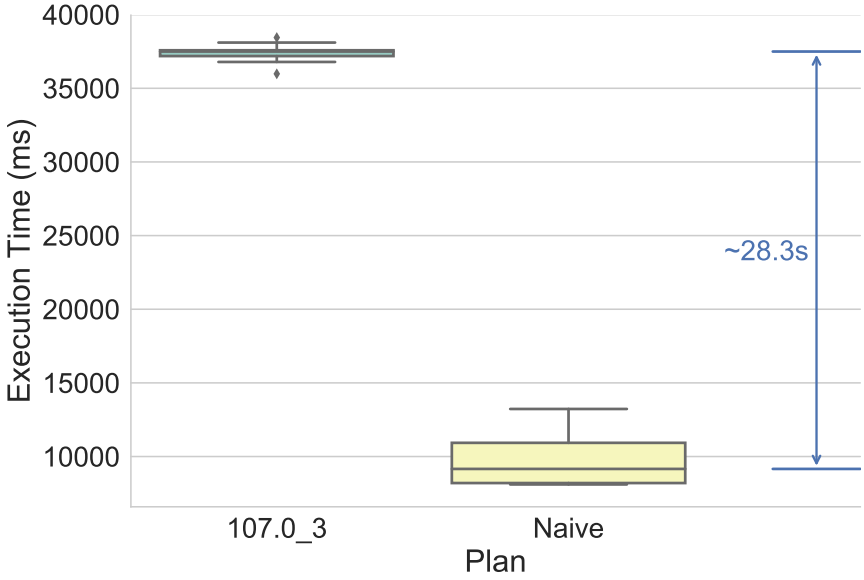


Figure 6.34: Execution time of the best execution plan vs. the *naïve plan* for the TextualPathExpnewtextPathExp transformation.

The graph reveals that the execution time distribution of the *naïve plan* is lower than the best execution plan, which is supported by the numerical values. The median age for the best execution plan (37495) is lower than for the *naïve plan* (9159). In fact, it should be noted that the first quartile of the best execution plan's distribution (37187) is higher than the median execution time for the *naïve plan*. We therefore conclude that in general, the *naïve plan* has a better performance than the best execution plan.

Analysis of variance showed a main effect of cost on the performance for the execution plans, $F(1, 48) = 5912$, $p < .001$, $\eta_{p^2} = 0.99$. Further, the results of a post-hoc Tukey HSD ($\alpha = 0.05$) test show that the execution times of both plans are significantly different. These results support the observations from Fig. 6.34.

The results to evaluate the third criterion are presented in Fig. 6.35. The figure shows a scatter plot of the execution times of the synthesized execution plans over the 25 trials. The black line marks the median value of the measurements. Analysis of variance showed that there is no statistically significant effect of cost on

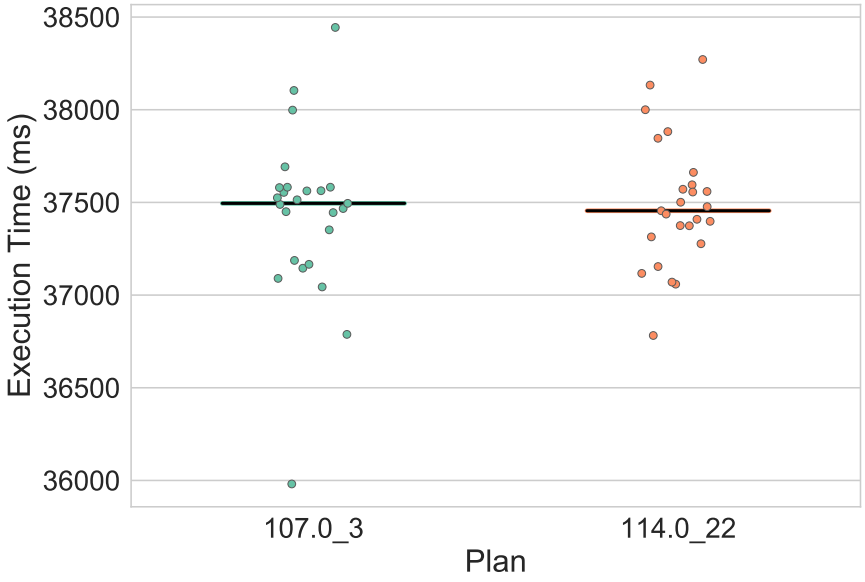


Figure 6.35: Execution time of the synthesized execution plans for the TextualPathExpnewtextPathExp transformation.

the performance for the execution plans, $F(1, 48) = 0.27$,ns). One side this shows that the best solution is the best performing too, amongst the synthesized plans. It also indicates that there is no other synthesized plan that has a better performance than the *naïve plan*.

PathExp2PetriNet. For the second experiment the source model for this transformation has 210 elements. The results in order to evaluate the second criterion are presented in Fig. 6.36. The graph reveals that the execution time distribution of the *naïve plan* is higher than the best execution plan, which is supported by the numerical values. The median execution time for the best execution plan (19012) is lower than for the *naïve plan* (52324). In fact, it should be noted that the third quartile of the best execution plan's distribution (19087) is lower than the median execution time for the *naïve plan*. We therefore conclude that in general, the best execution plan has a better performance than the *naïve plan*.

Analysis of variance showed a main effect of cost on the per-

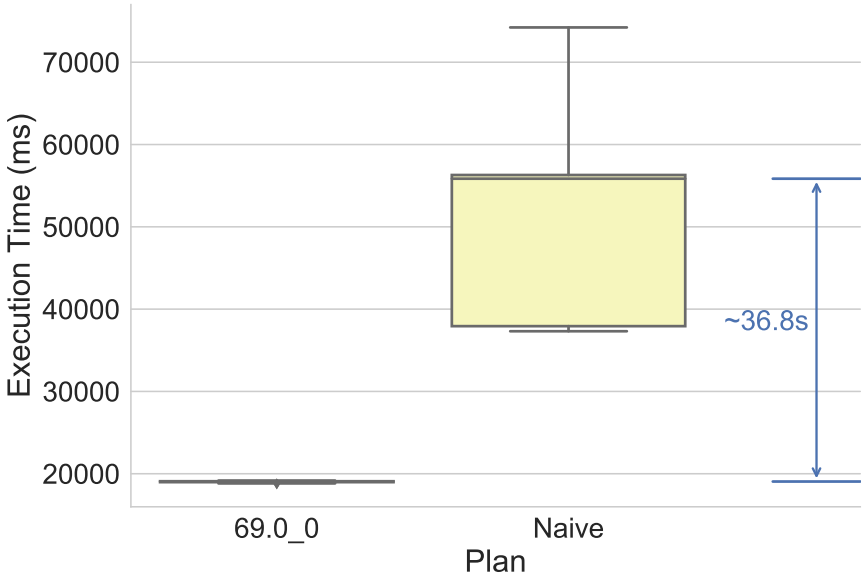


Figure 6.36: Execution time of the best execution plan vs. the *naïve plan* for the PathExp2PetriNet transformation.

formance for the execution plans, $F(1, 48) = 201$, $p < .001$, $\eta_{p^2} = 0.81$. Further, the results of a post-hoc Tukey HSD ($\alpha = 0.05$) test show that the execution times of both plans are significantly different. These results support the observations from Fig. 6.36.

The results to evaluate the third criterion are presented in Fig. 6.37. The figure shows a scatter plot of the execution times of the synthesized execution plans over the 25 trials. The black line marks the median value of the measurements. synthesized the best solution is the best performing too. Analysis of variance showed a main effect of cost on the performance for the execution plans, $F(1, 49) = 5.36$, $p = 0.025$, $\eta_{p^2} = 417$. Further, the results of a post-hoc Tukey HSD ($\alpha = 0.05$) test show that the execution times of the two synthesized plans are significantly different. The results show that the best plan ($(M = 5387, SD = 153)$) has a better performance the the other synthesized plan ($(M = 5459, SD = 289)$).

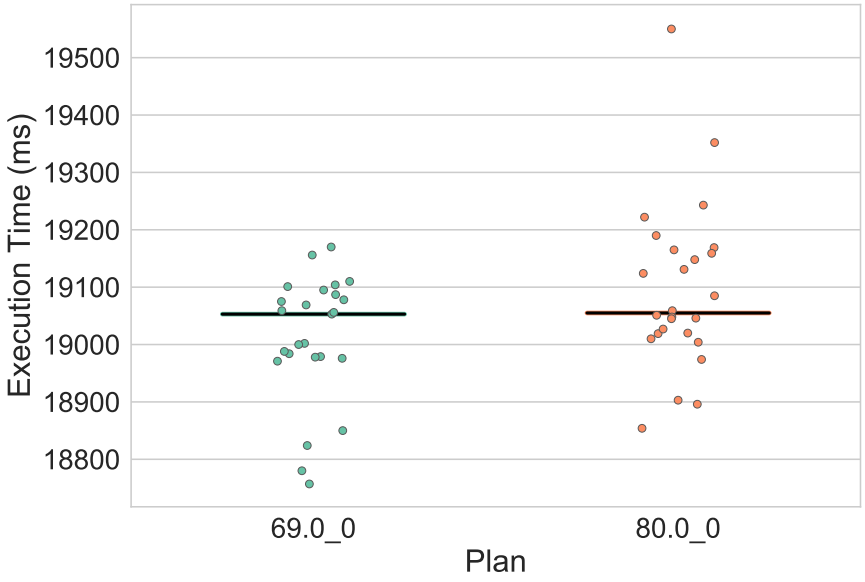


Figure 6.37: Execution time of the synthesized execution plans for the PathExp to Petri Net transformation.

PetriNet2XML. For the second experiment the source model for this transformation has 23000 elements. The results in order to evaluate the second criterion are presented in Fig. 6.38. The graph reveals that the execution time distribution of the *naïve plan* is lower than the best execution plan, which is supported by the numerical values. The median execution time for the best execution plan (380195) is higher than for the *naïve plan* (347836). In fact, it should be noted that the first quartile of the best execution plan's distribution (379779) is higher than the median execution time for the *naïve plan*. We therefore conclude that in general, the best execution plan has a worse performance than the *naïve plan*.

Analysis of variance showed a main effect of cost on the performance for the plans, $F(1, 48) = 288$, $p < .001$, $\eta_{p^2} = 0.86$. Further, the results of a post-hoc Tukey HSD ($\alpha = 0.05$) test show that the execution times of both plans are significantly different. These results support the observations from Fig. 6.38.

The results to evaluate the third criterion are presented in Fig. 6.39. The figure shows a scatter plot of the execution times

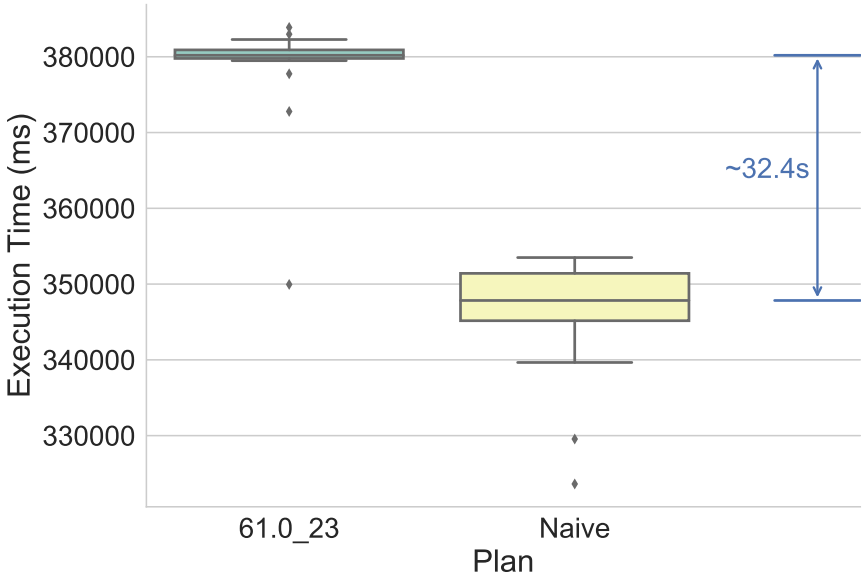


Figure 6.38: Execution time of the best execution plan vs. the *naïve plan* for the PetriNet2XML transformation.

of the synthesized execution plans over the 25 trials. The black line marks the median value of the measurements. Analysis of variance showed a main effect of cost on the performance for the execution plans, $F(21, 528) = 838$, $p = 0$, $\eta_{p^2} = 0.97$. In the figure it can be observed that the execution plans can be grouped into different groups based on the median performance. A Tukey HSD ($\alpha = 0.05$) post-doc test supports this observation:

Group a: 115.0_18, 61.0_23.

Group b: 133.0_6, 136.0_0, 138.0_5, 153.0_1.

Group c: 148.0_2, 152.0_14, 153.0_1.

Group d: 166.0_12, 167.0_13, 170.0_11, 176.0_10.

Group e: 184.0_8, 189.0_9.

Group f: 188.0_3, 201.0_7, 207.0_19, 218.0_4.

Group g: 203.0_16, 207.0_19, 218.0_4.

Group h: 236.0_21.

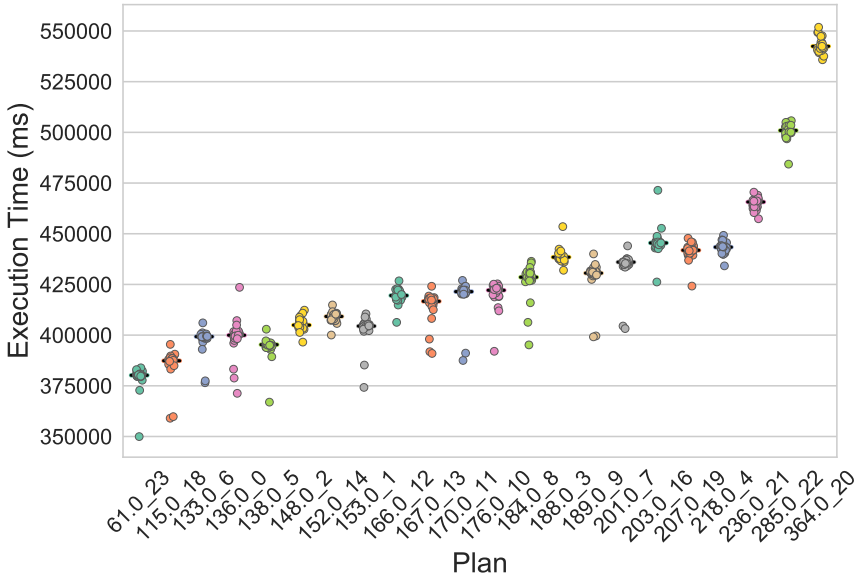


Figure 6.39: Execution time of the synthesized execution plans for the PetriNet2XML transformation.

Group i: 285.0_22.

Group j: 364.0_20.

These results indicate that the best solution has a performance amongst the best for the transformation.

Railway2Control. For the second experiment the source model for this transformation has 41 elements. The results in order to evaluate the second criterion are presented in Fig. 6.40. The graph reveals that the execution time distribution of the *naïve plan* is higher than the best execution plan, which is supported by the numerical values. The median execution time for the best execution plan (2576) is lower than for the *naïve plan* (90532). In fact, it should be noted that the third quartile of the best execution plan's distribution (46440) is lower than the median execution time for the *naïve plan*. We therefore conclude that in general, the best execution plan has a worse performance than the *naïve plan*. It is also important to note that the performance times for the *naïve*

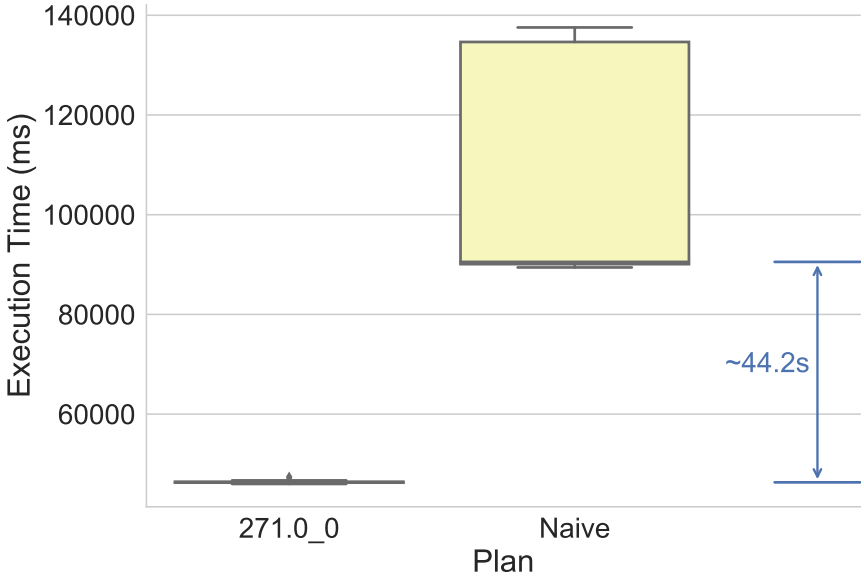


Figure 6.40: Execution time of the best execution plan vs. the *naïve plan* for the Railway2Control transformation.

plan have has a much larger variability than the temperatures in San Francisco (Range: 48139 vs. 181. IQR: 44566 vs. 28.0).

Analysis of variance showed a main effect of cost on the performance for the execution plans, $F(1, 48) = 184$, $p < .001$, $\eta_{p^2} = 0.79$. Further, the results of a post-hoc Tukey HSD ($\alpha = 0.05$) test show that the execution times of both plans are significantly different. These results support the observations from Fig. 6.40.

The results to evaluate the third criterion are presented in Fig. 6.39. The figure shows a scatter plot of the execution times of the synthesized execution plans over the 25 trials. The black line marks the median value of the measurements. Analysis of variance showed that there is no statistically significant effect of cost on the performance for the execution plans, $F(7, 192) = 1.46$, $p = 0.18$, $\eta_{p^2} = 0.051$. Although the results were not statistically significant, they indicate that one plan exhibits a performance slightly better than the rest: plan 279.0_10. Still, the best plan has a performance amongst the best for the transformation.

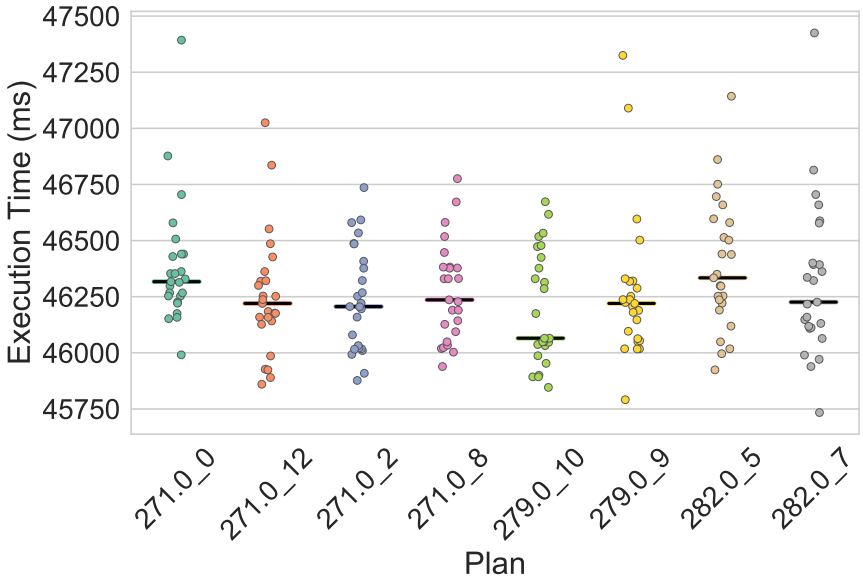


Figure 6.41: Execution time of the synthesized execution plans for the Railway2Control transformation.

6.13 Discussion

The evaluation of the proposed approach yielded the following findings:

1. The EPSA was able to synthesize an execution plan for 8 of the 9 evaluation transformations.
2. Execution of the best synthesized execution plans produced a correct transformation for all 8 synthesized evaluation transformations.
3. Execution of a sample of all synthesized execution plans for each of the 8 synthesized evaluation transformations produced a correct transformation in all the cases.
4. The best synthesized execution plans had a better performance than the naive approach in 6 of the 8 synthesized evaluation transformations.
5. The best synthesized execution plans had the best performance of all synthesized execution plans in 7 of the 8 eval-

uation transformations and it was amongst the best in the other one.

From these findings the following conclusions were drawn: It is possible to systematically synthesize execution plans for QVTc that result in correct transformations. Additionally, it is possible to systematically synthesize execution plans that have a better performance than the naive approach. Finally, the use of a meta-heuristic approach is a viable alternative for execution plan synthesis as it is often the case that the best solution exhibits the best performance.

The failure to synthesize an execution plan for the XSLT2-XQuery transformation limits the applicability of this approach and does not allow the findings to be generalized to all QVTc transformations. Taking a closer look at the execution logs of the algorithm, the construction always gets to a stage in which no further invocations can be added to the execution plan. This behaviour was first discussed in Sect. 4.3 and it indicates that the EPSA fails to produce a thorough execution plan. One possible solution is to make the execution plan a multi-graph. This would allow mappings to be re-invoked in order to consume the surplus elements in order to make the plan thorough. The risk with this modification is that it can result in infinite execution plans. The reason is that adding the additional invocation can in turn produce new surplus elements, for which another mapping must be invoked. This new invocation can result in new surplus elements, for which new invocations must be added. It is evident that this behaviour can be repeated indefinitely.

In general, the results show that the EPSA behaves as expected, with the effects of intensification and diversification (see Sect. 2.6.2) more visible for some transformations than for others. In cases in which a significant number of alternatives are found, the algorithm intensifies the search around solutions of low cost (in some cases the best cost) as expected. These results support the argument that the problem definition and its adaptation to be

solved with the ACO metaheuristic are correct. It is also clear that the number of alternative solutions depends on the transformation. Although transformations with small size and low LOC tend to have smaller space explorations (low number of total execution plans) this is not the case for the PetriNet to PNML example.

Although for some transformations there was little evidence of diversification, these transformations also exhibit very few alternative costs and very few alternative plans per cost. That is, the diversification may have been limited by the number of valid execution plans that can be constructed.

At a first glance it seems relevant to discuss the correlation between the transformation size (mappings/loc) and the execution time results. However, note that the execution time also depends on the size of the source model(s). Hence, it is not possible to directly compare the executions times of the different transformations.

Further research on the characteristics of the transformation and the DDG would be needed in order to determine if the behaviour of the EPSA with respect to intensification and diversification could be improved. The idea is that the EPSA can be further configured for a particular transformation. For example, if we can predict that few alternative execution plans can be constructed, it is possible to configure the algorithm to promote intensification.

The results of the second experiment do not allow the findings to be generalized to all QVTc transformations. To help us get a better picture, Table 6.6 summarizes the conclusions for the second experiment with respect to the performance and cost function. Criterion II is the performance of the best synthesized execution plan vs. the *naïve plan* and criterion III is the effectiveness of the cost function to differentiate between good and bad executions. A '+' indicates that the results support the criteria and a '-' that they oppose the criteria.

Table 6.6: Performance and Cost evaluation for the evaluation examples[Additional data and description].

	Criteria			
	II	Execution time difference(s)	III	Rank
Abstract2Concrete	+	5.43	+	~
BibTeXML2DocBook	+	134	+	2
DNF	+	699	+	1
Mi2Si	+	353	+	~
TextualPathExp2PathExp	-	−28.3	+	~
PathExp2PetriNet	+	36.8	+	1
PetriNet2XML	-	−32.4	+	1
Railway2Control	+	44.2	+	~

Given are the example name, and evidence of the II and III criteria. Criterion II is the difference in median execution time of the best plan vs the *naïve plan* (positive times indicate improvement). Criterion III is the ranking of the best plan vs. the other synthesized plans (ranked based on the median execution time, 1 is the highest rank; ~ indicates no statistical significance between execution plans). Results were taken over 25 trials.

6.13.1 Criterion II

Only two of the seven examples oppose criterion II: TextualPathExp2PathExp and PetriNet2XML. For the TextualPathExp2PathExp transformation a close inspection of the *naïve plan* reveals a $O(n^6)$ complexity while for the best-cost synthesized execution plan it is $O(n^7)$. The additional complexity could explain the differences in performance. The complexity of both plans could also explain why for these transformations the test models only had tens of elements (as opposed to tens of thousands). The additional complexity is the result of an additional loop in the best-cost synthesized execution plan. Since both invocations are from the root mapping, this indicates that the derivation algorithm fails to identify the optimal derivation in the synthesized case. Closer inspection of the specific mapping reveals that both derivations have the same cost and hence one is picked randomly. The derivation

cost formula could be adjusted using the observations from this transformation so that these derivations have different cost and favouring the derivation that results in fewer loops.

In the PetriNet2XML the problem can be attributed to that all mappings, except the main, produce and consume elements of type *XML:Element*. This results in two behaviours. All mappings invoke themselves and all mappings invoke all other mappings. In the *naïve plan* all mappings are scheduled only once, which results in less total invocations. However, with the proposed data dependence analysis it is impossible to identify that not all invocations are needed. A possible solution would be to make the data dependence pattern based. That is, there is not a dependency on a set of types, but on a pattern of elements and their relations. In the PetriNet2XML example this would result in the dependencies to exist only between the *main* mapping and the other mappings, eliminating the self loops in the DDG which will results in no loops in the execution plan.

For the plans that satisfy the criterion, the gain in performance are from a few seconds to almost 700. Given that in practice a model transformation would be expected to be executed repeatedly, an improvement of a few seconds can have considerable effects in the long run. For example, the $5.43s$ of the Abstract2Concrete would result in a gain of around a minute after 10 executions. Such a transformation could be part of a build system intended to produce executable code using information from the abstract UML model, which in turn could be part of a continuous integration workflow. Developers making changes to the abstract model would benefit greatly from the gain in execution time.

More substantial gains, such as the 700 seconds in the DNF example could be critical for transformations used in user interfaces. This type of transformation could be used by a math application to provide a DNF reduction of boolean operations. The improvement in performance signifies the difference between a tool that may appear unresponsive to the user (wait around 11 minutes for a

result) and a tool that provides result in a reasonable time (around 8 seconds).

6.13.2 Criterion III

All the examples support criteria III. For 4 of the 8 examples (Abstract2Concrete, Mi2Si, TextualPathExp2PathExp, and Railway2Control) there was no significant difference between the execution times of synthesized execution plans of different cost. Since we are only interested in differentiating between good and bad executions, the lack of significant difference does not impede the ability to do so. That is, the best solution is amongst the best performing ones. However, these examples could be further analysed to determine the cause of the similarities in performance and use this information to fine tune the cost function. This could lead to a better cost function that no only differentiates between good and bad executions, but that is also a good predictor of performance.

6.14 Threats to Validity

[Threats to validity section...] The major threads to validity of the evaluation are the size of the empirical deduction of the cost function, the sample of model transformations and the source models used. In the case of the source models, given that these models where generated randomly they might not correctly represent the structure and size of models used in real case scenarios. Given that the number of times a mapping is executed depends on the actual number of elements of the types of its input set, then it is possible that some execution plans favour a particular model structure. Recall the *Graph to Graph* example used in Sect. 5.2.3. If the source graph is disconnected, then a execution plan that invokes the *edge2edge* mapping from the *node2node* mapping would attempt to execute *edge2edge* for each execution of *node2node*. Another plan in which *edge2edge* is invoked from the *root* will only attempt to execute the *edge2edge* mapping as many times as edges

exist in the model. Although not all attempts of the first case will succeed (the mapping code will not be executed) given that many will result in NA (not all nodes may have edges) the overhead of attempting the ascription can have a negative effect in the performance for source graphs with considerable number of nodes.

In the case of the size of the sample of model transformations, the failure to synthesize execution plans for the XSLT2XQuery (the example with the largest number of mappings) indicates that the behaviour of the synthesis algorithm or the constructions constraints needs further analysis for larger transformations. One of the possible reasons for this is in the thoroughness validation (see Sect. 5.4.2). Consider a transformation such that $m_w \Psi_t m_v$ and $m_v \Psi_t m_w$, and that a plan is constructed such that the last mapping invoked is m_w. The thoroughness validation will indicate that m_w is a surplus producer, which might result in an invocation to m_v being added to the execution plan. However, now m_v is a surplus producer which might result in an invocation to m_w to be added to the execution plan. This could result in a loop of m_v and m_w invocations being added to the execution plan, with the result that a valid plan cannot be constructed in the allotted time.

The empirical deduction of the cost function for execution plans using a single example can result in a cost function that is highly biased towards the particular example used. The effects of this approach can be easily seen in the BibTeXML2DocBook example, where low cost plans show the worst performance. Or in the Railway2Control example where the difference in performance of the different plans are not statistically significant. In the former it is possible that the cost function is not correctly taking into consideration complexity (i.e. the presence of more loop actions) and in the latter, it is possible that all the plans should actually have the same cost.

6.15 Summary

This chapter presented the examples selected for evaluation of the proposed approach, the computational results of the EPSA and the results obtained for the two experiments designed to test the research hypotheses. The first experiment produced evidence that the best synthesized execution plan results in correct transformations for all the evaluation transformations. The second experiment produced evidence that in all but two of the evaluation transformations, the best synthesized execution plan has a better performance than the *naïve plan*. The second experiment also produced evidence that best plan had either the best performance or one of the best performances. The next chapter presents the conclusions of this project, including a discussion on these results.

7

Conclusions

7.1 Summary of Contributions

Model transformation languages (MTLs) are important for Model Driven Engineering (MDE) as they allow the automation of certain phases of the design process of hardware and software products, in particular at the preliminary and detailed design phases. The implementation of the execution engine of a declarative MTL involves the synthesis of the control component. as part of either the compilation or execution of the program. Synthesis of efficient control components is challenging because the semantics of the MTL must be preserved and because of the size of the solution space of possible execution plans for a particular Model Transformation Program (MTP). In order to overcome the implementation challenge, in particular for the QVT Core [3] (QVTc) language, this research proposed the adoption of compiler theories used in the implementation of General Purpose Languages (GPLs). This research has shown that instruction scheduling based on data dependence analysis is a feasible approach for synthesizing efficient control components.

This project answers the research question, stated in Sect. 1.2:

Can correct and efficient control components for pro-

grams written in the QVTc language be systematically synthesized for QVTc programs.

by providing a method for synthesis and evidence to support the hypotheses that synthesized control components produce correct transformations, that the synthesized control components have a better performance than the naive approach and that the proposed cost function is effective in differentiating between good and bad executions.

This research showed that inter-mapping data dependence information can be used to define a precedence-based partial ordering of the mappings in a QVTc transformation. The inter-mapping data dependence relations are the result of the producer–consumer relations between mappings. A mapping consumes elements of some types, given by the types of the mapping's input variables (normal variables in the QVTc syntax). A mapping produces elements of some types, given by the types of the mapping's output variables (realized variables in the QVTc syntax). A data dependence relation exists between a mapping that produces elements of a given type and the mappings that consume elements of that type. The inter-mapping data dependence relations are captured in a data dependence graph (DDG).

Scheduling using the precedence-based partial order minimizes the number of times a mapping needs to be scheduled and minimizes mapping invocations that have a NA result. This research showed that mapping partial order is a necessary, but not sufficient condition for an execution plan to result in correct transformation execution (Lemma 1). In order to guarantee correctness the concept of *thoroughness* was introduced (Definition 18).

This research also showed that data dependence analysis can be applied to the variables and predicates in a QVTc mapping (intra–mapping relations) and that the variable data dependencies can be used to optimize the constructions of the argument tuples needed for mapping invocations. The result of the intra–mapping data dependence analysis was called *variable derivation*. Variable

derivation results in some of the mapping's input variables to be derived (their value calculated) from a *primary* variable. A loop over all elements of a type is not needed to find values for the derived variables. As a result, the number of loops in an invocation path is minimized and this results in a minimization of the number of times a mapping is invoked. Further, variable derivation is achieved using the information from the mapping's guards (predicates). Hence, derived variables are guaranteed to satisfy the predicate used to determine its derivation. This minimizes the number of invocations with a False result.

After arguing that synthesis of the execution plan is a scheduling problem, this research showed that the problem is in particular a minimum–weight rooted spanning arborescence (MWRSA) problem on the DDG. As such, the execution plan synthesis problem is a combinatorial optimization problem amenable to be solved using the Ant Colony Optimization (ACO) metaheuristic [71, 19]. This research showed that the execution plan synthesis problem can be modelled in the form $P = (S, \Omega, h)$, which is the problem representation used by the ACO metaheuristic [35]. In this model, S is the search space defined by all possible execution plans. Ω are the constraints that determine if an execution plan is feasible and correspond to Definition 19. $h : S \longmapsto \mathbb{R}_0^+$ is the objective function to be minimized, which in this case assigns an execution cost to each execution plan.

This research presented how the variable derivation information can be used to find the optimal invocation paths for a QVTc transformation. Using only the optimal invocation paths to construct execution plans reduces the search space. A cost function that weights the derivations used in an invocation path was derived empirically.

This research provided an implementation of the $\mathcal{MAX}$ – $\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) algorithm [98] used for finding a solution to the execution plan is a synthesis problem. The adaptations include an alternative representation of the execution plan

that can be used as the foraging area of the algorithm, a terminating condition and the calculation of the feasible neighbourhood.

The evaluation of the proposed approach was conducted via two experiments. The first experiment evaluated the correctness of the synthesized execution plans. The second experiment evaluated the performance of the best synthesized execution plan against the naive approach and the accuracy of the cost function to distinguish between good and bad executions. For evaluation a set of nine unseen transformations was used. The evaluation suggested that construction of efficient execution plans for QVTc MTP is feasible and that the gain in performance can be in the range of a few seconds to several hundreds. Small performance gains are important in cases when the MTP is going to be executed repeatedly (e.g. as part of a continuous integration workflow) and big performance gains can make the use of model driven engineering more attractive for domains as user interfaces (where the performance gains of hundreds of seconds result in an increase usability for the end users.)

7.2 Future Work

All the examples and code base of the proposed approach have been made publicly available, with the intention that they can be reused by the model driven engineering community. The code base also includes a working QVTi execution engine, which can be used to execute the synthesized execution plans. This allows other researchers to synthesize execution plans for their QVTc transformations and then execute the best plan using their existing models.

The availability of the development and evaluation examples is also useful as they can be used to continue the research in improving the presented approach with respect to the cost function and the execution plan synthesis algorithm. For the cost function, it would be interesting to analyse the synthesized execution plans

and try to improve it so that it does not only allow the distinction between good and bad plans, but that is also a predictor of performance, i.e. the cost is proportional to the performance. For the synthesis algorithm, the XSLT2XQuery example could be debugged to further understand the reason for not been able to produce a result. The results of the analysis could be used to improve the neighbourhood construction. Given that some of the transformations used are part of the ATL Zoo, it is possible that there exist source models that have been created as part of a real case scenario. Repeating the evaluation with these models would provided additional evidence on the feasibility of the approach and its use in practice.

One of the aspects that was not addressed in this research, and that remains an open question, is the complexity of the DDG. Complexity of graphs has been analysed in domains such as networking in order to understand how network traffic is affected by the network topology [49]. It would be interesting to research on which, if any, of this metrics apply to the DDG and how this affects the synthesis algorithm. Are different synthesis strategies, different ACO settings, stronger/lazier constrains during execution plan construction required based on the DDG topology? For some transformations the best execution plan was found in the first iterations. Is this behaviour related DDG topology, and hence can the DDG topology be used to guide the number of iterations that are done to produce a result?

The Meta Object Facility (MOF) Query/View/Transformation (QVT) Specification was expected to be the "de facto"when it was introduced several years ago. With the recent efforts of the Eclipse QVT Declarative project a full implementation of the QVT Relations [3] (QVTr) and QVTc languages should be finally made available. Although its late arrival might impede it to become the "de facto"language, it can still be used as a referent for evaluation and possibly as a base execution engine for other transformation languages. By using the ideas proposed on this research, the QVT

declarative engine could become the reference for MTLs implementation with respect to adopting proven compiler techniques into the execution of MTPs.

Other areas of future research are, but not limited to, discussed next. Data dependence based execution plan synthesis can be used in other rule based declarative transformation languages. The base of the control component model is the notion of mapping invocation and iteration over elements of a type in order to create the arguments needed for the invocation. The base of the scheduling of rules is the notion of producer–consumer relations between mappings. This behaviour is observed, to some degree, by all rule based declarative transformation languages. That is, during invocation a rule consumes a tuple of elements (arguments) and produces a tuple of elements (result). As such, extending this approach to other transformation languages should be straightforward. For this, three major activities are required:

1. Extend/modify the execution plan to include additional actions based on the language semantics.

2. Adjust the data dependency analysis to cover other semantic rules that determine consumer-producer relations between mappings.

3. Determine how the generated execution plan can be incorporated into the execution.

For example, in the Epsilon Transformation Language (ETL) mappings can be annotated to consume elements that are strictly of the input type. This would require the use of, for example, an additional all-of-type loop action (that does not loop over elements of sub-types). Also, in the Atlas Transformation Language (ATL) language the trace model is implicit and as such dependencies must also be identified by analysing uses of the *resolveTemp* (a language construct to access the trace model). Further, if the language does not provide imperative constructs to make execution plan part of the original transformation code, the execution engine

could require modifications so that it can compile or interpret the execution plan.

Since this research assumed a rewrite enforce execution mode, additional research is necessary to understand what effects other modes of execution have on the DDG and how this affect the synthesis of the execution plan. Additional optimizations are possible by improving the intra–mapping dependence analysis in order to support the analysis more complex predicates, such as `<varA>.<property>->includes (<varB>)`. Additional research should be planned in order to consider this and other common patterns into the dependence analysis and variable derivation. This should further reduce the number of loops in the invocation paths.

In this research the ACO algorithm parameters were chosen based on previous work that solved problems not directly comparable to the execution plan synthesis problem. An interesting area of future research is the effect of the parameter settings on the behaviour of the synthesis algorithm. Further, it would also be interesting to compare the use of different metaheuristics for the synthesis algorithm. This could be then mixed with the DDG complexity analysis to possible determine if particular metaheuristics are better suited for particular DDG topologies. An important question is whether the ACO metaheuristic provides better results than a random search approach and/or if the additional effort is justified.

When alternative transformations exist, which result in equivalent target models, "the transformation designer must be able to identify which alternative transformations produce models with the desired quality attributes" [54]. Another possible direction of future research is to understand if the structure of the synthesized execution plans has an effect on the quality aspects of the generated target models. Or, alternatively, if the expected performance of a synthesized execution plans for a particular transformation alternative can be an additional metric for quantifying the quality of the transformation. For this, additional experiments from the

point of view of the transformation designers should be carried out. These experiments could include how knowing the expected performance of the transformation can guide the development of the MTP and how providing feedback of the MTP's DDG topology could be used by the developer to refactor the transformation in order to improve the execution performance.

Given that termination property of a MTP does depend on the execution plans, an open challenge is that if given a QVTc MTP that is correct (with respect to termination, confluence and behaviour preservation) the proposed definition of transformation correctness could be extended to prove that the resulting QVTi MTP (based on the synthesized execution plan) is correct. For this it would also be necessary to determine the effect of the merging phase in the termination property of the MTP.

Finally, finding a correlation between the transformation size (mappings/loc) and the execution time is an interesting research that can help get a better understanding of the aspects that affect the cost function. For example, if two alternative MTP for the same transformation, with different number of mappings have similar execution times (for the same source model), assuming the MTP with fewer mappings has lower cost, then it might be the case that the cost is not dependant on the number of mappings only, but on additional properties of the execution plan: e.g. depth/breath.

Part IV

Appendix

8

QVTc Transformations

8.1 Families to Persons

Listing 8.1: Complete Families to Persons example.

```
1  import fMM : 'Families.ecore'::Families;
2  import pMM : 'Persons.ecore'::Persons;
3  import f2pMM : 'Families2Persons.ecore'::Families2Persons;

5  transformation F2P {
6      family imports fMM;
7      person imports pMM;
8      imports f2pMM;
9  }

11 -- Last names always come from my Family. *Adams* and *Eves* (i.e.
       root members
12 -- of a family tree might not have a family
13 query F2P::familyName(member : Families::Member) : String {
14     if (not member.familySon.oclIsUndefined()) then
15         member.familySon.lastName
16     else
17         if (not member.familyDaughter.oclIsUndefined()) then
18             member.familyDaughter.lastName
19         else
20             ''
21          endif
22     endif
23 }

25 query F2P::isFemale(member : Families::Member) : Boolean {
26     if (not member.familyMother.oclIsUndefined()) then
27         true
```

```
28      else
29          if (not member.familyDaughter.oclIsUndefined()) then
30              true
31          else
32              false
33          endif
34      endif
35  }

37  map Member2Male in F2P {
38      check family (s : Member |
39          not isFemale(s);) {}
40      enforce person () {
41          realize t : Male |
42      }
43      where () {
44          realize m2m : Member2Male |
45          m2m.member := s;
46          m2m.person := t;
47      }
48      map {
49          where () {
50              t.fullName :=  s.firstName + ' ' + familyName(s);
51          }
52      }
53  }

55  map Member2Female in F2P {
56      check family (s : Member |
57          isFemale(s);) { }
58      enforce person () {
59          realize t : Female |
60      }
61      where () {
62          realize m2m : Member2Female |
63          m2m.member := s;
64          m2m.person := t;
65      }
66      map {
67          where () {
68              t.fullName :=  s.firstName + ' ' + familyName(s);
69          }
70      }
71  }
```

8.2 Upper to Lower

Listing 8.2: Complete Upper to Lower example.

```
1 import SimpleGraph : 'SimpleGraph.ecore#/';
2 import SimpleGraph2Graph : 'SimpleGraph2Graph.ecore#/';
```

```
4  transformation Upper2Lower
5  {
6      upperGraph imports SimpleGraph;
7      lowerGraph imports SimpleGraph;
8      imports SimpleGraph2Graph;
9  }
10 /*
11  * Don't use realize keywords on the two initial domains indicates
        that both models must exist and at least have the root node
        defined?
12  */

14 map graph2graph in Upper2Lower
15 {
16     check enforce upperGraph() {
17         realize g1 : Graph
18     |}

20     enforce lowerGraph() {
21     /*
22              * Enforced domains should at least have one realized
                    variable?
23              */
24 realize g2 : Graph
25     |}

27     where() {
28     /*
29              * Although in the example is not realized, all middle
                    model variables should be realized
30              * or does the middle model is also expected to have at
                    least the initial element?
31              */
32 realize g2g : SimpleGraph2Graph::Graph2Graph
33     |
34         g2g.graph1 := g1;
35         g2g.graph2 := g2;
36     }

38     map
39     {

41         where() {
42             g2g.name := g1.name.toLowerCase();
43             g2g.name := g2.name.toUpperCase();
44             g1.name := g2g.name;
45             g2.name := g2g.name;
46         }
47     }
48 }
```

```
50 map node2node in Upper2Lower
51 {
52     check enforce upperGraph(g1 : Graph
53     |) {
54         realize n1 : Node
55     |
56         n1.graph := g1;
57     }

59     enforce lowerGraph(g2 : Graph
60     |) {
61         realize n2 : Node
62     |
63         n2.graph := g2;
64     }

66     where(g2g : SimpleGraph2Graph::Graph2Graph
67     |
68         g2g.graph1 = g1;
69         g2g.graph2 = g2;) {
70         realize n2n : SimpleGraph2Graph::Node2Node
71     |
72         n2n.owner := g2g;
73         n2n.node1 := n1;
74         n2n.node2 := n2;
75     }

77     map
78     {

80         where() {
81             n2n.label := n1.label.toLowerCase();
82             n2n.label := n2.label.toUpperCase();
83             n1.label := n2n.label;
84             n2.label := n2n.label;
85         }
86     }
87 }

89 map edge2edge in Upper2Lower
90 {
91     enforce upperGraph(g1 : Graph,
92         sn1 : Node,
93         tn1 : Node
94     |) {
95         realize e1 : Edge
96     |
97         e1.graph := g1;
98         e1.source := sn1;
99         e1.target := tn1;
100    }
```

```
102     enforce lowerGraph(g2 : Graph,
103         sn2 : Node,
104         tn2 : Node
105     |) {
106         realize e2 : Edge
107     |
108         e2.graph := g2;
109         e2.source := sn2;
110         e2.target := tn2;
111     }

113     where(g2g : SimpleGraph2Graph::Graph2Graph,
114         sn2n : SimpleGraph2Graph::Node2Node,
115         tn2n : SimpleGraph2Graph::Node2Node
116     |
117         g2g.graph1 = g1;
118         g2g.graph2 = g2;
119         sn2n.owner = g2g;
120         sn2n.node1 = sn1;
121         sn2n.node2 = sn2;
122         tn2n.node1 = tn1;
123         tn2n.node2 = tn2;) {
124         realize e2e : SimpleGraph2Graph::Edge2Edge
125     |
126         e2e.owner := g2g;
127         e2e.edge1 := e1;
128         e2e.edge2 := e2;
129         e2e.source := sn2n;
130         e2e.target := tn2n;
131     }
132 }
```

8.3 HSV to HSL

Listing 8.3: Complete HSV to HSL example.

```
1 import SimpleGraph : 'SimpleGraph.ecore#/';
2 import SimpleGraph2Graph : 'SimpleGraph2Graph.ecore#/';

4 transformation Upper2Lower
5 {
6     upperGraph imports SimpleGraph;
7     lowerGraph imports SimpleGraph;
8     imports SimpleGraph2Graph;
9 }
10 /*
11  * Don't use realize keywords on the two initial domains indicates
        that both models must exist and at least have the root node
        defined?
12  */
```

```
14 map graph2graph in Upper2Lower
15 {
16     check enforce upperGraph() {
17         realize g1 : Graph
18     |}

20     enforce lowerGraph() {
21     /*
22               * Enforced domains should at least have one realized
                     variable?
23               */
24 realize g2 : Graph
25     |}

27     where() {
28     /*
29               * Although in the example is not realized, all middle
                     model variables should be realized
30               * or does the middle model is also expected to have at
                     least the initial element?
31               */
32 realize g2g : SimpleGraph2Graph::Graph2Graph
33     |
34         g2g.graph1 := g1;
35         g2g.graph2 := g2;
36     }

38     map
39     {

41         where() {
42             g2g.name := g1.name.toLowerCase();
43             g2g.name := g2.name.toUpperCase();
44             g1.name := g2g.name;
45             g2.name := g2g.name;
46         }
47     }
48 }

50 map node2node in Upper2Lower
51 {
52     check enforce upperGraph(g1 : Graph
53     |) {
54         realize n1 : Node
55     |
56         n1.graph := g1;
57     }

59     enforce lowerGraph(g2 : Graph
60     |) {
61         realize n2 : Node
62     |
```

```
63          n2.graph := g2;
64      }

66      where(g2g : SimpleGraph2Graph::Graph2Graph
67      |
68          g2g.graph1 = g1;
69          g2g.graph2 = g2;) {
70          realize n2n : SimpleGraph2Graph::Node2Node
71      |
72          n2n.owner := g2g;
73          n2n.node1 := n1;
74          n2n.node2 := n2;
75      }

77      map
78      {

80          where() {
81              n2n.label := n1.label.toLowerCase();
82              n2n.label := n2.label.toUpperCase();
83              n1.label := n2n.label;
84              n2.label := n2n.label;
85          }
86      }
87  }

89  map edge2edge in Upper2Lower
90  {
91      enforce upperGraph(g1 : Graph,
92          sn1 : Node,
93          tn1 : Node
94      |) {
95          realize e1 : Edge
96      |
97          e1.graph := g1;
98          e1.source := sn1;
99          e1.target := tn1;
100     }

102     enforce lowerGraph(g2 : Graph,
103         sn2 : Node,
104         tn2 : Node
105     |) {
106         realize e2 : Edge
107     |
108         e2.graph := g2;
109         e2.source := sn2;
110         e2.target := tn2;
111     }

113     where(g2g : SimpleGraph2Graph::Graph2Graph,
114         sn2n : SimpleGraph2Graph::Node2Node,
```

```
115         tn2n : SimpleGraph2Graph::Node2Node
116     |
117         g2g.graph1 = g1;
118         g2g.graph2 = g2;
119         sn2n.owner = g2g;
120         sn2n.node1 = sn1;
121         sn2n.node2 = sn2;
122         tn2n.node1 = tn1;
123         tn2n.node2 = tn2;) {
124         realize e2e : SimpleGraph2Graph::Edge2Edge
125     |
126         e2e.owner := g2g;
127         e2e.edge1 := e1;
128         e2e.edge2 := e2;
129         e2e.source := sn2n;
130         e2e.target := tn2n;
131     }
132 }
```

8.4 Hstm to Stm

Listing 8.4: Complete Hstm to Stm example.

```
1  import 'HSVTree.ecore'::HSVTree;
2  import 'HSLTree.ecore'::HSLTree;
3  import 'HSV2HSL.ecore'::HSV2HSL;

5  transformation hsv2hsl {
6      hsv imports HSVTree; -- Should specify the correct package
7      hsl imports HSLTree; --
8      imports HSV2HSL;
9  }

11 /* The hsv, hls, rgb operations where adapted from
       https://gist.github.com/mjackson/5311256
12  * The colors are stored in the models with these ranges:
13  * HSV:
14  *  H in [0, 360]
15  *  S in [0, 100]
16  *  V in [0, 100]
17  * HSL:
18  *  H in [0, 360]
19  *  S in [0, 100]
20  *  L in [0, 100]
21  * RGB:
22  *  H in [0, 1]
23  *  S in [0, 1]
24  *  V in [0, 1]
25  */

27 query hsv2hsl::rgb2hsl(rgb:Sequence(Real)) : Sequence(Real) {
```

```
28      let r: Real = rgb->at(1),
29          g: Real = rgb->at(2),
30          b: Real = rgb->at(3),
31          rgbReal = Sequence{r,g,b},
32          max: Real = rgbReal->iterate(x : Real, y:Real=0 | y.max(x)),
33          min: Real = rgbReal->iterate(x : Real, y:Real=1 | y.min(x)),
34          l: Real = (max+min)/2
35          in
36              if max = min then
37                  Sequence{0, 0, l*100}
38              else
39                  let c:Real = max-min,
40                      s:Real = if l < 0.5 then
41                                   c / (max+min)
42                               else
43                                   c / (2-max-min)
44                               endif,
45                      r1:Real = ((max-r)/6 + c/2)/c,
46                      g1:Real = ((max-g)/6 + c/2)/c,
47                      b1:Real = ((max-b)/6 + c/2)/c
48                  in
49                  if max = r then
50                      let h:Real = b1 - g1
51                      in
52                          Sequence{h*360, s*100, l*100}
53                  else
54                      if max = g then
55                          let h: Real = 1/3 + r1 - b1
56                          in
57                              Sequence{h*360, s*100, l*100}
58                      else     -- max = b
59                          let h: Real = 2/3 + g1 - r1
60                          in
61                              Sequence{h*360, s*100, l*100}
62                      endif
63                  endif
64              endif
65  }

67  query hsv2hsl::hsl2rgb(hsl:Sequence(Real)) : Sequence(Real) {

69      let h: Real = hsl->at(1),
70          s: Real = hsl->at(2)/100,
71          l: Real = hsl->at(3)/100,
72          c: Real = 1 - (2*l-1).abs()*s,
73          h2: Real = (h/60),
74          hmod: Real = h2.floor().mod(2) + (h/60 - h2.floor()),
75          x:Real = c*(1-(hmod-1).abs()),
76          m:Real = l - 0.5*c
77       in
78          if h2 < 1 then
79              Sequence{c+m, x+m, m}
```

```
80          else
81              if h2 < 2 then
82                  Sequence{x+m, c+m, m}
83              else
84                  if h2 < 3 then
85                      Sequence{m, c+m, x+m}
86                  else
87                      if h2 < 4 then
88                          Sequence{m, x, c}
89                      else
90                          if h2 < 5 then
91                              Sequence{x+m, m, c+m}
92                          else
93                              Sequence{c+m, m, x+m}
94                          endif
95                      endif
96                  endif
97              endif
98          endif
99  }

101 query hsv2hsl::rgb2hsv(rgb:Sequence(Real)) : Sequence(Real) {
102     let r: Real = rgb->at(1),
103         g: Real = rgb->at(2),
104         b: Real = rgb->at(3),
105         rgbReal = Sequence{r,g,b},
106         max: Real = rgbReal->iterate(x : Real, y:Real=0 | y.max(x)),
107         min: Real = rgbReal->iterate(x : Real, y:Real=1 | y.min(x)),
108         v: Real = max
109         in
110             if max = min then
111                 Sequence{0, 0, v*100}
112             else
113                 let c:Real = max-min,
114                     s:Real = c/max,
115                     r1:Real = ((max-r)/6 + c/2)/c,
116                     g1:Real = ((max-g)/6 + c/2)/c,
117                     b1:Real = ((max-b)/6 + c/2)/c
118                 in
119                 if max = r then
120                     let h:Real = b1 - g1
121                     in
122                         Sequence{h*360, s*100, v*100}
123                 else
124                     if max = g then
125                         let h: Real = 1/3 + r1 - b1
126                         in
127                             Sequence{h*360, s*100, v*100}
128                     else     -- max = b
129                         let h: Real = 2/3 + g1 - r1
130                         in
131                             Sequence{h*360, s*100, v*100}
```

```
132                     endif
133                 endif
134             endif
135 }

137 query hsv2hsl::hsv2rgb(hsv:Sequence(Real)) : Sequence(Real) {
138     let h: Real = hsv->at(1),
139         s: Real = hsv->at(2)/100,
140         v: Real = hsv->at(3)/100,
141         i:Integer = (h/60).floor(),
142         f:Real = h/60 - i,
143         p:Real = v * (1 - s),
144         q:Real = v * (1 - f * s),
145         t:Real = v * (1 - (1 - f) * s)
146      in
147         if i = 0 then
148               Sequence{v, t, p}
149         else
150             if i = 1 then
151                 Sequence{q, v, p}
152             else
153                 if i = 2 then
154                     Sequence{p, v, t}
155                 else
156                     if i = 3 then
157                         Sequence{p, q, v}
158                     else
159                         if i = 4 then
160                             Sequence{t, p, v}
161                         else
162                             Sequence{v, p, q}
163                         endif
164                     endif
165                 endif
166             endif
167         endif
168 }

170 map HSV2HSLRoot in hsv2hsl {
171     check hsv(hsvRoot : HSVNode |
172         hsvRoot.parent = null;) { }
173     check enforce hsl() {
174         realize hslRoot : HSLNode |
175         hslRoot.parent := null;
176     }
177     where( ) {
178         realize middleRoot : HSVNode2HSLNode |
179         middleRoot.hsv := hsvRoot;
180         middleRoot.hsl := hslRoot;
181         middleRoot.parent := null;
182     }
183     map {
```

```
184         where() {
185             middleRoot.name := hsvRoot.name;
186             middleRoot.name := hslRoot.name;
187             hsvRoot.name := middleRoot.name;
188             hslRoot.name := middleRoot.name;
189             middleRoot.rgb := hsv2rgb(hsvRoot.hsv);
190             middleRoot.rgb := hsl2rgb(hslRoot.hsl);
191             hsvRoot.hsv := rgb2hsv(middleRoot.rgb);
192             hslRoot.hsl := rgb2hsl(middleRoot.rgb);
193         }
194     }
195 }

197 map HSV2HSLRecursion in hsv2hsl {
198     check enforce hsv(hsvParent : HSVNode | ) {
199         realize hsvNode : HSVNode |
200         hsvNode.parent := hsvParent;
201     }
202     check enforce hsl(hslParent : HSLNode |) {
203         realize hslNode : HSLNode |
204         hslNode.parent := hslParent;
205     }
206     where(middleParent : HSVNode2HSLNode |
207         middleParent.hsv = hsvParent;
208         middleParent.hsl = hslParent;
209     ) {
210         realize middleNode : HSVNode2HSLNode |
211         middleNode.parent := middleParent;
212         middleNode.hsv := hsvNode;
213         middleNode.hsl := hslNode;
214         }
215     map {
216         where () {
217             middleNode.name := hsvNode.name;
218             middleNode.name := hslNode.name;
219             hslNode.name := middleNode.name;
220             hsvNode.name := middleNode.name;
221             middleNode.rgb := hsv2rgb(hsvNode.hsv);
222             middleNode.rgb := hsl2rgb(hslNode.hsl);
223             hsvNode.hsv := rgb2hsv(middleNode.rgb);
224             hslNode.hsl := rgb2hsl(middleNode.rgb);
225         }
226     }
227 }
```

8.5 UML to RDBMS Minimal

Listing 8.5: CompleteUML to RDBMS Minimal example.

```
1 import MinUML : 'MinimalUML.ecore#/';
2 import MinRDBMS : 'MinimalRDBMS.ecore#/';
```

```
3  import MinUML2RDBMS : 'MinimalUML2RDBMS.ecore#/';

5  transformation umlRdbms {
6      uml imports minimaluml;
7      rdbms imports minimalrdbms;
8      imports minimaluml2rdbms;
9  }

11 /** packageToSchema */
12 map p2s in umlRdbms {
13     uml(p:Package | ) { }
14     enforce rdbms() { realize s:Schema }
15     where() {realize p2s:PackageToSchema |
16         p2s.umlPackage := p; p2s.schema := s; }
17     map {
18         where() { p2s.name := p.name; s.name := p2s.name; } }
19 }
20 /** integerToNumber */
21 map i2n in umlRdbms {
22     uml(p:Package, prim:PrimitiveDataType |
23         prim.namespace = p; prim.name = 'Integer';) { }
24     check enforce rdbms() { sqlType:String | sqlType := 'NUMBER'; }
25     where(p2s:PackageToSchema |
26         p2s.umlPackage = p;) {
27         realize p2n:PrimitiveToName |
28         p2n.owner := p2s; p2n.primitive := prim;
29         p2n.typeName := sqlType; }
30     map {
31         where() { p2n.name := prim.name + '2' + sqlType; } }
32 }
33 /** booleanToBoolean */
34 map b2b in umlRdbms {
35     uml(p:Package, prim:PrimitiveDataType |
36         prim.namespace = p;  prim.name = 'Boolean';) { }
37     check enforce rdbms() { sqlType:String | sqlType := 'BOOLEAN'; }
38     where(p2s:PackageToSchema |
39         p2s.umlPackage = p;) {
40         realize p2n:PrimitiveToName |
41         p2n.owner := p2s; p2n.primitive := prim;
42         p2n.typeName := sqlType; }
43     map {
44         where() { p2n.name := prim.name + '2' + sqlType; } }
45 }
46 /** stringToVarchar */
47 map s2v in umlRdbms {
48     uml(p:Package, prim:PrimitiveDataType |
49         prim.namespace = p;  prim.name = 'String'; ) { }
50     check enforce rdbms() { sqlType:String | sqlType := 'VARCHAR'; }
51     where(p2s:PackageToSchema |
52         p2s.umlPackage = p;) {
53         realize p2n:PrimitiveToName |
54         p2n.owner := p2s; p2n.primitive := prim;
```

```
55          p2n.typeName := sqlType; }
56      map {
57          where() { p2n.name := prim.name + '2' + sqlType; } }
58  }
59  /** classToTable */
60  map c2t in umlRdbms {
61      uml(p:Package, c:Class |
62          c.kind = 'persistent'; c.namespace = p;) { }
63      check enforce rdbms(s:Schema |) {
64          realize t:Table |
65          t.kind := 'base'; t.schema := s; }
66      where(p2s:PackageToSchema |
67          p2s.umlPackage = p; p2s.schema = s;) {
68          realize c2t:ClassToTable |
69          c2t.owner := p2s; c2t.attOwner := c; c2t.table := t; }
70      map {
71          where() { c2t.name := c.name; t.name := c2t.name; } }
72  }
73  /** fromAttribute */
74  map fa in umlRdbms {
75      uml(c:Class, t:PrimitiveDataType, a:Attribute |
76          a.owner = c; a.type = t; ) { }
77      where(fao:AttributeOwner, p2n:PrimitiveToName |
78          fao.attOwner = c; p2n.primitive = t;) {
79          realize fa:AttributeToColumn |
80          fa.attribute := a; fa.owner := fao; fa.type := p2n; }
81      map {
82          where() { fa.name := a.name; } }
83  }
84  /** attributeColumn */
85  map a2c in umlRdbms {
86      check enforce rdbms(t:Table |) {
87          realize c:Column |
88          c.owner := t; }
89      where(c2t:ClassToTable, a2c:AttributeToColumn,
90              p2n:PrimitiveToName |
91              c2t.table = t; a2c.owner = c2t; a2c.type = p2n; ) {
92          a2c.column := c; }
93      map {
94          where() { c.name := a2c.name; c.type := p2n.typeName; } }
95  }
```

8.6 UML to RDBMS

Listing 8.6: Complete UML to RDBMS example.

```
1 import SimpleUML : 'SimpleUML.ecore#/';
2 import SimpleRDBMS : 'SimpleRDBMS.ecore#/';
3 import SimpleUML2RDBMS : 'SimpleUML2RDBMS.ecore#/';

5 transformation umlRdbms
```

```
6  {
7      uml imports simpleuml;
8      rdbms imports simplerdbms;
9      imports simpleuml2rdbms;
10 }

12 /*
13  * -- Package and Schema mapping
14  * class PackageToSchema {
15  * composite classesToTables : Set(ClassToTable) opposites owner;
16  * composite primitivesToNames : Set(PrimitiveToName) opposites
        owner;
17  * name : String;
18  * -- uml
19  * umlPackage : Package;
20  * -- rdbms
21  * schema : Schema;
22  * }
23  */

25 map packageToSchema in umlRdbms
26 {
27     uml() {
28         p:Package
29     }
30     enforce rdbms() {
31         realize s:Schema
32     }
33     where() {
34         realize p2s:PackageToSchema |
35         p2s.umlPackage := p;
36         p2s.schema := s;
37     }
38     map
39     {
40         where()
41         {
42             p2s.name := p.name;
43             p2s.name := s.name;
44             p.name := p2s.name;
45             s.name := p2s.name;
46         }
47     }
48 }

50 /*
51  * -- Primitive data type marshaling
52  * class PrimitiveToName {
53  * owner : PackageToSchema opposites primitivesToNames;
54  * name : String;
55  * -- uml
56  * primitive : PrimitiveDataType;
```

```
57  * -- rdbms
58  * typeName : String;
59  * }
60  */
61 map primitiveToName in umlRdbms
62 {
63     uml(p:Package)
64     {
65         prim:PrimitiveDataType |
66         prim.namespace = p;
67     }
68     check enforce rdbms()
69     {
70         sqlType : String
71     }
72     where(p2s:PackageToSchema |
73         p2s.umlPackage = p;)
74     {
75         realize p2n:PrimitiveToName |
76         p2n.owner := p2s;
77         p2n.primitive := prim;
78         p2n.typeName := sqlType;
79     }
80     map
81     {
82         where()
83         {
84             p2n.name := prim.name + '2' + sqlType;
85         }
86     }
87 }

89 map integerToNumber in umlRdbms refines primitiveToName
90 {
91     uml()
92     {
93         prim.name = 'Integer';
94     }
95     check enforce rdbms()
96     {
97         sqlType := 'NUMBER';
98     }
99 }

101 map booleanToBoolean in umlRdbms refines primitiveToName
102 {
103     uml()
104     {
105         prim.name = 'Boolean';
106     }
107     check enforce rdbms()
108     {
```

```
109             sqlType := 'BOOLEAN';
110         }
111 }

113 map stringToVarchar in umlRdbms refines primitiveToName
114 {
115         uml()
116         {
117             prim.name = 'String';
118         }
119         check enforce rdbms() {
120             sqlType := 'VARCHAR';
121         }
122 }

124 -- Queries can now be at the root of the transformation
125 /*
126 map flattening in umlRdbms refines associationToForeignKey ,
        attributes {}
127 */

129 query umlRdbms::getAllSupers(cls : SimpleUML::Class) :
        Set(SimpleUML::Class)
130 {
131         cls.general->collect(gen |
            getAllSupers(gen))->including(cls)->asSet()
132 }

134 query umlRdbms::getAllAttributes(cls : SimpleUML::Class) :
        Set(SimpleUML::Attribute)
135 {
136         getAllSupers(cls)->collect(c | c.attributes)->asSet()
137 }

139 query umlRdbms::getAllForwards(cls : SimpleUML::Class) : Set(
        SimpleUML::Association)
140 {
141         getAllSupers(cls)->collect(c | c.forward)->asSet()
142 }


145 /*
146  * -- Class and Table mapping
147  * class ClassToTable extends FromAttributeOwner, ToColumn {
148  * owner : PackageToSchema opposites classesToTables;
149  * composite associationToForeignKeys :
150  * OrderedSet(AssociationToForeignKey) opposites owner;
151  * name : String;
152  * -- uml
153  * umlClass : Class;
154  * -- rdbms
155  * table : Table;
```

```
156  * primaryKey : Key;
157  * }
158  */

160 map classToTable in umlRdbms
161 {
162     check enforce uml(p:Package |)
163     {
164         realize c:Class |
165         c.kind := 'persistent';
166         c.namespace := p;
167     }
168     check enforce rdbms(s : Schema |)
169     {
170         realize t:Table |
171         default t.kind := 'base';
172         t.schema := s;
173         t.kind <> 'meta';
174     }
175     where(p2s:PackageToSchema |
176         p2s.umlPackage = p;
177         p2s.schema = s;)
178     {
179         realize c2t:ClassToTable |
180         c2t.owner := p2s;
181         c2t.umlClass := c;
182         c2t.table := t;
183     }
184     map
185     {
186         where()
187         {
188             c2t.name := c.name;
189             c2t.name := t.name;
190             c.name := c2t.name;
191             t.name := c2t.name;
192         }
193     }
194     map
195     {
196         check enforce rdbms()
197         {
198             realize pk:Key,
199             realize pc:Column |
200             pk.owner := t;
201             pk.kind := 'primary';
202             pc.owner := t;
203             default pc.keys := Set(SimpleRDBMS::Key){pk};
204             default pc.type := 'NUMBER';
205             pc.keys->includes(pk);
206         }
207         where()
```

```
208         {
209             c2t.primaryKey := pk;
210             c2t.column := pc;
211         }
212         map
213         {
214             check enforce rdbms()
215             {
216                 pc.name := t.name + '_tid';
217                 pk.name := t.name + '_pk';
218             }
219         }
220     }
221 }
222 /*
223  * -- Association and ForeignKey mapping
224  * class AssociationToForeignKey extends ToColumn {
225  * referenced : ClassToTable;
226  * owner : ClassToTable opposites associationToForeignKeys;
227  * name : String;
228  * -- uml
229  * association : Association;
230  * -- rdbms
231  * foreignKey : ForeignKey;
232  * }
233  */

235 map associationToForeignKey in umlRdbms
236 {
237     check enforce uml(p:Package, sc:Class,  dc:Class |
238         sc.kind = 'persistent';
239         dc.kind = 'persistent';
240         sc.namespace = p;)
241     {
242         realize a:Association |
243         getAllForwards(sc)->includes(a);
244         default a.source := sc;
245         getAllSupers(dc)->includes(a.destination);  -- How does
                associations work with inheritance?
246         default a.destination := dc;
247         default a.namespace := p;
248     }
249     check enforce rdbms(s:Schema, st:Table, dt:Table, rk:Key |
250         st.schema = s;
251         rk.owner = dt;
252         rk.kind = 'primary';)
253     {
254         realize fk:ForeignKey,
255         realize fc:Column |
256         fk.owner := st;
257         fc.owner := st;
258         fk.refersTo := rk;
```

```
259         default fc.foreignKeys := Set(SimpleRDBMS::ForeignKey){fk};
260         fc.foreignKeys->includes(fk);
261     }

263     where(p2s:PackageToSchema, sc2t:ClassToTable, dc2t:ClassToTable |
264         sc2t.owner = p2s;
265         p2s.umlPackage = p;
266         p2s.schema = s;
267         sc2t.table = st;
268         dc2t.table = dt;
269         sc2t.umlClass = sc;
270         dc2t.umlClass = dc;)
271     {
272         realize a2f:AssociationToForeignKey |
273         a2f.owner := sc2t;
274         a2f.referenced := dc2t;
275         a2f.association := a;
276         a2f.foreignKey := fk;
277         a2f.column := fc;
278     }
279     map
280     {
281         where() {
282             a2f.name := if a.destination = dc and a.source = sc
283                             then a.name
284                         else if a.destination <> dc and a.source = sc
285                             then dc.name + '_' + a.name
286                         else if a.destination = dc and a.source <> sc
287                             then a.name + '_' + sc.name
288                         else dc.name + '_' + a.name + '_' + sc.name
289                         endif endif endif;
290             a.name :=   if a.destination = dc and a.source = sc
291                             then a2f.name
292                         else a.name
293                         endif;
294             fk.name := a2f.name;
295             a2f.name := fk.name;
296             fc.name := a2f.name + '_tid';
297         }
298     }
299     map
300     {
301         where()
302         {
303             fc.type := rk.column->first().type;
304         }
305     }
306 }
307 /*
308  * -- attribute mapping
309  * abstract class FromAttributeOwner {
310  * composite fromAttributes : Set(FromAttribute) opposites owner;
```

```
311  * }
312  * abstract class FromAttribute {
313  * name : String;
314  * kind : String;
315  * owner : FromAttributeOwner opposites fromAttributes;
316  * leafs : Set(AttributeToColumn);
317  * -- uml
318  * attribute : Attribute;
319  * }
320  * abstract class ToColumn {
321  * -- rdbms
322  * column : Column;
323  * }
324  * class NonLeafAttribute extends FromAttributeOwner, FromAttribute {
325  * leafs := fromAttributes.leafs;
326  * }
327  * class AttributeToColumn extends FromAttribute, ToColumn {
328  * type : PrimitiveToName;
329  * }
330  */

332 map attributes in umlRdbms
333 {
334     check enforce uml(c:Class |)
335     {
336         realize a:Attribute |
337         default a.owner := c;
338         getAllAttributes(c)->includes(a);
339     }
340     where(fao:FromAttributeOwner |)
341     {
342         realize fa:FromAttribute |
343         fa.attribute := a;
344         fa.owner := fao;
345     }
346     map
347     {
348         where()
349         {
350             fa.kind := a.kind;
351             a.kind := fa.kind;
352         }
353     }
354 }

356 map classAttributes in umlRdbms refines attributes
357 {
358     where(fao:ClassToTable |
359         fao.umlClass = c;)
360     {}
361     map
362     {
```

```
363         where()
364         {
365             fa.name := a.name;
366             a.name := fa.name;
367         }
368     }
369 }

371 map primitiveAttribute in umlRdbms refines attributes
372 {
373     check enforce uml(t:PrimitiveDataType |)
374     {
375         a.type := t;
376     }
377     where(p2n:PrimitiveToName |
378         p2n.primitive = t;)
379     {
380         realize fa : SimpleUML2RDBMS::AttributeToColumn |
381         fa.type := p2n;
382     }
383     map
384     {
385         where()
386         {
387             fa.leafs := Set(SimpleUML2RDBMS::AttributeToColumn) {fa};
388         }
389     }
390 }

392 map complexAttributeAttributes in umlRdbms refines attributes
393 {
394     check uml(ca:Attribute |
395         ca.type = c;)
396     {}
397     where(fao:NonLeafAttribute |
398         fao.attribute = ca;)
399     {}
400     map
401     {
402         where()
403         {
404             /* TODO We should add ca.name because if we have two
                    complex
405             * attributes of the same type, the columns would be
                    identical ?
406             */
407             fa.name := fao.name + '_' + a.name;
408         }
409     }
410 }

412 map complexAttribute in umlRdbms refines attributes
```

```
413 {
414     check uml(t:Class |)
415     {
416         a.type = t;
417     }
418     where()
419     {
420         realize fa:NonLeafAttribute|
421     }
422     map
423     {
424         where()
425         {
426             fa.leafs := fao.fromAttributes.leafs;
427         }
428     }
429 }

431 map classPrimitiveAttributes in umlRdbms refines classAttributes,
        primitiveAttribute {}

433 map classComplexAttributes in umlRdbms refines classAttributes,
        complexAttribute {}

435 map complexAttributePrimitiveAttributes in umlRdbms refines
        complexAttributeAttributes, primitiveAttribute {}

437 map complexAttributeComplexAttributes in umlRdbms refines
        complexAttributeAttributes, complexAttribute {}

439 /*
440  * -- column mapping
441  */
442 query umlRdbms::getAllLeafAtributes(fao :
        SimpleUML2RDBMS::FromAttributeOwner) :
        Set(SimpleUML2RDBMS::AttributeToColumn)
443 {
444     let leafs : Set(SimpleUML2RDBMS::AttributeToColumn) =
            fao.fromAttributes
445             ->selectByKind(SimpleUML2RDBMS::AttributeToColumn) in
446         leafs->includingAll(fao.fromAttributes
447             ->selectByKind(SimpleUML2RDBMS::NonLeafAttribute)
448             ->collect(nla |
                    getAllLeafAtributes(nla.oclAsType(SimpleUML2RDBMS::FromAttributeOwner))))
449 }

451 map attributeColumns in umlRdbms
452 {
453     check enforce rdbms(t:Table |)
454     {
455         realize c:Column |
456         c.owner := t;
```

```
457         c.keys->size() = 0;
458         c.foreignKeys->size() = 0;
459     }
460     where(c2t:ClassToTable |
461         c2t.table = t;)
462     {
463         realize a2c:AttributeToColumn |
464         a2c.column := c;
465         -- default a2c.owner := c2t; -- Default assignments should
                not be used for check.
466         --fao.fromAttributes.leafs->includes(a2c);
467         getAllLeafAtributes(c2t)->includes(a2c);
468     }
469     map
470     {
471         where(p2n:PrimitiveToName |)
472         {
473             ct:String |
474             a2c.type := p2n;
475             ct := c.type;
476             ct := p2n.typeName;
477             p2n.typeName := ct;
478             c.type := ct;
479         }
480     }
481     map
482     {
483         where()
484         {
485             c.name := a2c.name;
486             a2c.name := c.name;
487         }
488     }
489     map
490     {
491         where()
492         {
493             c.kind := a2c.kind;
494             a2c.kind := c.kind;
495         }
496     }

498 } -- end of module UmlRdbmsTransformation
```

8.7 Abstract to Concrete

The example describes a transformation of a simplified UML model to another simplified UML model. The aim of this transformation is to generate, from a source UML model, another UML model

that flattens the inherited operations of a class. That is, a `Class` in the target model will collect all the operations inherited from the closure of it's super classes that are abstract.

Listing 8.7: Complete Abstract to Concrete example.

```
14 import UMLMM : 'ClassMM.ecore'::ClassMM;
15 import ABS2CONC : 'ClassAbstractToConcrete.ecore'::abs2conc;

17 /**
18  * Produces model that highlights which classes need to implement
19  * methods existing in their abstract superclass.
20  */
21 transformation AbstractToConcrete {
22     umlIn imports UMLMM;
23     umlOut imports UMLMM;
24     imports ABS2CONC; }

26 map PackageToPackage in AbstractToConcrete {
27     umlIn (pIn:Package) { }
28     enforce umlOut () {
29         realize pOut:Package |  }
30     where() {
31         realize p2p:PackageToPackage |
32         p2p.pIn := pIn;
33         p2p.pOut := pOut;
34     }
35     map {
36         where() {
37             p2p.name := pIn.name;
38             pOut.name := p2p.name;
39         }
40     }
41 }

43 map AbstractClassToConcreteClass in AbstractToConcrete {
44     umlIn (pIn:Package, cc1:Class, ac:Class |
45             cc1.owner = pIn;
46             not cc1.inheritsFrom.oclIsUndefined();
47             cc1.inheritsFrom = ac;
48             ac.owner = pIn;
49             ac.isAbstract; ) { }
50     enforce umlOut (pOut:Package) {
51         realize cc2 : Class |
52         cc2.owner := pOut;
53     }
54     where (p2p:PackageToPackage |
55             p2p.pIn = pIn;
56             p2p.pOut = pOut; ) {
57         realize p2c : ParentToChild |
58         p2c.parent := ac;
59         p2c.class := cc1;
```

```
60          p2c.concreteClass := cc2;
61          p2c.owner := p2p;
62      }
63      map {
64          where() {
65              p2c.name := cc1.name;
66              cc2.name := p2c.name;
67          }
68      }
69  }

71  map OperationToOperation in AbstractToConcrete {
72      umlIn (pc : Class, sc:Class, aco:Operation |
73              pc.isAbstract;
74              subClasses(pc)->notEmpty();
75              not sc.inheritsFrom.oclIsUndefined();
76              sc.inheritsFrom = pc;
77              aco.owner = pc; ) { }
78      enforce umlOut (cc2 : Class | ) {
79          realize cco:Operation |
80          cco.owner := cc2;
81      }
82      where (p2c : ParentToChild |
83              p2c.class = sc;
84              p2c.parent = pc;
85              p2c.concreteClass = cc2; ) {
86          realize o2o : OperationToOperation |
87          o2o.abstract := aco;
88          o2o.concrete := cco;
89          o2o.class := p2c;
90      }
91      map {
92          where() {
93              o2o.name := aco.name;
94              o2o.name := cco.name;
95              aco.name := o2o.name;
96              cco.name := o2o.name;
97          }
98      }
99  }

101 map DataTypeToDataType in AbstractToConcrete {
102   umlIn(dt:PrimitiveDataType, p:Package |
103         dt.owner = p; ) { }
104   enforce umlOut(po:Package | ) {
105     realize dto:PrimitiveDataType |
106     dto.owner := po;
107   }
108   where(p2p:PackageToPackage |
109         p2p.pIn = p;
110         p2p.pOut = po; ) {
111     realize dt2dt:DataTypeToDataType |
```

```
112     dt2dt.dIn := dt;
113     dt2dt.dOut := dto;
114   }
115   map {
116     where() {
117       dt2dt.name := dt.name;
118       dto.name := dt2dt.name;
119     }
120   }
121 }

123 map ParameterToParameter in AbstractToConcrete {
124     umlIn (aco:Operation, acop:Parameter, dt:PrimitiveDataType |
125             aco.owner.isAbstract;
126             acop.operation = aco;
127             acop.type = dt; ) { }
128     enforce umlOut (cco:Operation, dto:PrimitiveDataType) {
129         realize ccop:Parameter |
130         ccop.operation := cco;
131     }
132     where (o2o : OperationToOperation, d2d:DataTypeToDataType |
133             o2o.abstract = aco;
134             o2o.concrete = cco;
135             d2d.dIn = dt;
136             d2d.dOut = dto; ) {
137         realize p2p : ParameterToParameter |
138         p2p.abstract := acop;
139         p2p.concrete := ccop;
140         p2p.operation := o2o;
141     }
142     map {
143         where() {
144             p2p.name := acop.name;
145             ccop.name := p2p.name;
146             p2p.type := d2d;
147             ccop.type := dto;
148         }
149     }
150 }

152 query AbstractToConcrete::subClasses(class : ClassMM::Class) :
153         Set(ClassMM::Class) {
154     ClassMM::Class.allInstances()->select(c:ClassMM::Class |
155                                     c.inheritsFrom = class)
156 }
```

8.8 BibTeXML to DocBook

The BibTeXML to DocBook example describes a transformation of a BibTeXML model to a DocBook model. BibTeXML [84] is an XML-based format for the BibTeX bibliographic tool. DocBook [107] is an XML-based format for document composition.

Listing 8.8: Complete BibTeXML to DocBook example.

```
9  import BibTexMM : 'BibTeX.ecore'::BibTeX;
10 import DocBookMM : 'DocBook.ecore'::DocBook;
11 import BibTex2DocBookMM : 'BibTex2DocBook.ecore'::Trace;


14 transformation bibtex2docbook {
15     bibtex imports BibTexMM;
16     docbook imports DocBookMM;
17     imports BibTex2DocBookMM;
18 }

20 map Main in bibtex2docbook {
21     check bibtex(bib:BibTeXFile){}
22     enforce docbook() {
23         realize doc:DocBook, realize book:Book, realize art:Article,
24         realize se1:Sect1, realize se2:Sect1, realize se3:Sect1,
25         realize se4:Sect1 |
26         doc.books := OrderedSet{book};
27         book.articles := OrderedSet{art};
28         art.sections_1 := OrderedSet{se1, se2, se3, se4};
29         se1.title := 'References List';
30         se2.title := 'Authors List';
31         se3.title := 'Titles List';
32         se4.title := 'Journals List';
33     }
34     where () {
35         realize bib2doc:Bib2Doc |
36         bib2doc.file := bib;
37         bib2doc.doc := doc;
38     }
39 }

41 map InfoToPara in bibtex2docbook  {
42     enforce docbook(se:Sect1 ) {
43         realize p:Para |
44         se.paras := se.paras->including(p);
45     }
46     where (){        --fb:FromBibtex) {
47         realize e2p:InfoToPara,
48         realize fb:FromBibtex |
49         e2p.para := p;
50     }
```

```
51      map {
52          where() {
53              p.content := fb.info;
54          }
55      }
56  }

58  map EntryToPara in bibtex2docbook refines InfoToPara {
59      check bibtex(entry:BibTeXEntry) {}
60      enforce docbook(se.title = 'References List';) {}
61      where () {
62          realize fb:FromEntry |
63          e2p.fromEntry := fb;
64          fb.entry := entry;
65          fb.info := buildEntryPara(entry);
66      }
67  }

69  map AuthorToPara in bibtex2docbook refines InfoToPara {
70      check bibtex(a:Author |
71          not authorSet()->includes(a.author);
72      ) {}
73      enforce docbook(se.title = 'Authors List';) {}
74      where () {
75          realize fb:FromAuthor |
76          fb.author := a;
77          fb.info := a.author;
78      }
79  }

81  map TitleToPara in bibtex2docbook refines InfoToPara {
82      check bibtex(e:TitledEntry |
83          not titleSet()->includes(e.title);
84      ) {}
85      enforce docbook(se.title = 'Titles List';) {}
86      where (fe:FromEntry |
87          fe.entry = e;) {
88          realize fb:FromTitle |
89          fb.entry := e;
90          fb.info := e.title;
91          fe.leafs := fe.leafs->including(fb);
92      }
93  }

95  map JournalToPara in bibtex2docbook refines InfoToPara {
96      check bibtex(a:Article |
97          not titleSet()->includes(a.journal);
98      ) {}
99      enforce docbook(se.title = 'Journals List';) {}
100     where (fe:FromEntry |
101         fe.entry = a;) {
102         realize fb:FromArticle |
```

```
103             fb.article := a;
104             fb.info := a.journal;
105             fe.leafs := fe.leafs->including(fb);
106         }
107 }

109 -- This helper makes sure we only transform authors once by checking
        what is
110 -- already transformed.
111 -- RETURN: Sequence(BibTeX::Author)
112 query bibtex2docbook::authorSet() : Sequence(String)
113 {
114     BibTex2DocBookMM::FromAuthor.allInstances()->collect(e |
            e.info)->asSequence()
115 }

117 -- This helper makes sure we only transform titles once by checking
        what is
118 -- already transformed.
119 -- RETURN: Sequence(BibTeX::Author)
120 query bibtex2docbook::titleSet() : Sequence(String)
121 {
122     BibTex2DocBookMM::FromTitle.allInstances()->collect(e |
            e.info)->asSequence()
123 }

125 -- This helper makes sure we only transform journals once by
        checking what is
126 -- already transformed.
127 -- RETURN: Sequence(BibTeX::Author)
128 query bibtex2docbook::journalSet() : Sequence(String)
129 {
130     BibTex2DocBookMM::FromArticle.allInstances()->collect(e |
            e.info)->asSequence()
131 }

133 -- This helper builds a string containing all information on a
        given BibTeXEntry.
134 -- Content of the generated string depends on the entry type.
135 -- IN: BibTeX::BibTeXEntry
136 -- RETURN: String
137 query bibtex2docbook::buildEntryPara(entry:BibTexMM::BibTeXEntry) :
        String
138 {
139     '[' + entry.id + ']' +
140     ' ' + entry.oclType().name +
141     (if entry.oclIsKindOf(BibTeX::TitledEntry) then
142             ': ' + entry.oclAsType(BibTeX::TitledEntry)?.title
143         else '' endif) +
144     (if entry.oclIsKindOf(BibTeX::AuthoredEntry) then
145             entry.oclAsType(BibTeX::AuthoredEntry)?.authors?->iterate(e;
                    str : String = ': ' | str + e.author + '; ')
```

```
146         else '' endif) +
147     (if entry.oclIsKindOf(BibTeX::DatedEntry) then
148             ': ' + entry.oclAsType(BibTeX::DatedEntry)?.year
149         else '' endif) +
150     (if entry.oclIsKindOf(BibTeX::BookTitledEntry) then
151             ': ' +
                    entry.oclAsType(BibTeX::BookTitledEntry)?.booktitle
152         else '' endif) +
153     (if entry.oclIsKindOf(BibTeX::ThesisEntry) then
154             ': ' +  entry.oclAsType(BibTeX::ThesisEntry)?.school
155         else '' endif) +
156     (if entry.oclIsKindOf(BibTeX::Article) then
157             ': ' + entry.oclAsType(BibTeX::Article)?.journal
158         else '' endif) +
159     (if entry.oclIsKindOf(BibTeX::Unpublished) then
160             ': ' + entry.oclAsType(BibTeX::Unpublished)?.note
161         else '' endif) +
162     (if entry.oclIsKindOf(BibTeX::Book) then
163             ': ' + entry.oclAsType(BibTeX::Book)?.publisher
164         else '' endif) +
165     (if entry.oclIsKindOf(BibTeX::InBook) then
166             ': ' +
                    entry.oclAsType(BibTeX::InBook)?.chapter.toString()
167         else '' endif)
168 }
```

8.9 DNF

The example describes a transformation of a simplified UML model to another simplified UML model. The aim of this transformation is to generate, from a source UML model, another UML model that flattens the inherited operations of a class. That is, a `Class` in the target model will collect all the operations inherited from the closure of it's super classes that are abstract.

Listing 8.9: Complete DNF example.

```
16 import exprMM : 'DNFMM.ecore'::booleanexp;
17 import traceMM : 'DNF2DNF.ecore'::be2dnf;

19 transformation dnf {
20     bexp imports exprMM;
21     dnf imports exprMM;
22     imports traceMM;
23 }

25 map BooleanExprs in dnf {
```

```
26      bexp(root:BooleanExprs) { }
27      enforce dnf() {
28          realize rootdnf:BooleanExprs |
29      }
30      where() {
31          realize r2r:Root2Root |
32          r2r.be := root;
33          r2r.dnf := rootdnf;
34      }
35  }

37  -- All literals are transformed
38  map Lit2Lit in dnf {
39      check bexp(a:Literal | ) { }
40      enforce dnf() {
41          realize adnf:Literal |
42      } where() {
43          realize e2e:Lit2Lit |
44          e2e.beExpr := a;
45          e2e.dnfExpr := adnf;
46          e2e.name := a.name;
47          e2e.ID := a.ID;
48          e2e.beParentExpr := a.parent;
49          adnf.name := e2e.name;
50          adnf.ID := e2e.ID;
51      }
52  }

54  -- Copy all And expressions that are not OrDistributive or DeMorgan
55  map And2AndNoDist in dnf {
56      check bexp(andbe:And |
57          andbe.parent.oclIsUndefined() or
58              (not andbe.parent.oclIsUndefined() and
59              not andbe.parent.oclIsTypeOf(Not) and
60              not andbe.expr->exists(e | e.oclIsTypeOf(Or)) -- and
                    (andbe.parent.oclIsTypeOf(And) and not
                    andbe.parent.expr->exists(e | e.oclIsTypeOf(Or)))
61              );
62      ) { }
63      enforce dnf() {
64          realize anddnf:And |
65      }
66      where() {
67          realize e2e:Expr2Expr |
68          e2e.beExpr := andbe;
69          e2e.dnfExpr := anddnf;
70          -- If parent is OrDistribution, my parent (if dnf) is my
                sibling
71          e2e.beParentExpr := if not andbe.parent.oclIsUndefined() and
72                                  andbe.parent.oclIsTypeOf(And) and
73                                  andbe.parent.expr->exists(e |
                                        e.oclIsTypeOf(Or)) then
```

```
74                                  andbe.parent.expr->select(e | e <>
                                        andbe)->first()
75                              else
76                                  andbe.parent
77                              endif;
78          e2e.ID := andbe.ID;
79          anddnf.ID := e2e.ID;
80      }
81  }

83  -- Copy all Or expressions that are not OrDistributive or DeMorgan
84  map Or2OrNoDist in dnf {
85      check bexp(orbe:Or) {
86          orbe.parent.oclIsUndefined() or (
87              not orbe.parent.oclIsUndefined() and
88              not orbe.parent.oclIsTypeOf(Not) and
89              (not (orbe.parent.oclIsTypeOf(And) and
90                  orbe.parent.expr->exists(e | e.oclIsTypeOf(Or)))));
91      }
92      enforce dnf() {
93          realize ordnf:Or |
94      }
95      where() {
96          realize e2e:Expr2Expr |
97          e2e.beExpr := orbe;
98          e2e.dnfExpr := ordnf;
99          e2e.beParentExpr := orbe.parent;
100         e2e.ID := orbe.ID;
101         ordnf.ID := e2e.ID;
102     }
103 }

105 -- First push the negations down to the leaves of the syntax tree
        using
106 -- De Morgan's laws and double-negation elimination.
107 -- ~(a+b) = ~a.~b
108 map DeMorganOr in dnf {
109     check bexp(nbe:Not, orbe:Or |
110         nbe.expr->includes(orbe); ) { }
111     enforce dnf() {
112         realize anddnf:And, realize anot:Not, realize bnot:Not |
113         anddnf.expr := Sequence{anot, bnot};
114     }
115     where () {
116         realize not2and:DeMorgan, realize notor2nots:Expr2MultExpr |
117         not2and.beExpr := nbe;
118         not2and.dnfExpr := anddnf;
119         not2and.ID := nbe.ID;
120         not2and.beParentExpr := nbe.parent;
121         anddnf.ID := not2and.ID;
122         notor2nots.beExpr := orbe;
123         notor2nots.dnfExprs := Sequence(exprMM::Not){anot, bnot};
```

```
124         notor2nots.ID := orbe.ID;
125         anot.ID := notor2nots.ID + '_1';
126         bnot.ID := notor2nots.ID + '_2';
127     }
128 }

130 -- ~(a.b) = ~a+~b
131 map DeMorganAnd in dnf {
132     check bexp(nbe:Not, andbe:And |
133         nbe.expr->includes(andbe); ) { }
134     enforce dnf() {
135         realize ordnf:Or, realize anot:Not, realize bnot:Not |
136         ordnf.expr := Sequence{anot, bnot};
137     }
138     where () {
139         realize not2or:DeMorgan, realize notand2nots:Expr2MultExpr |
140         not2or.beExpr := nbe;
141         not2or.dnfExpr := ordnf;
142         not2or.ID := nbe.ID;
143         not2or.beParentExpr := nbe.parent;
144         ordnf.ID := not2or.ID;
145         notand2nots.beExpr := andbe;
146         notand2nots.dnfExprs := Sequence(exprMM::Not){anot, bnot};
147         notand2nots.ID := andbe.ID;
148         anot.ID := notand2nots.ID + '_1';
149         bnot.ID := notand2nots.ID + '_2';
150     }
151 }

153 -- Then float the disjunctions to the top using the distributive law
154  -- (a).(b+c) = (a.b) + (a.c)
155  -- (b+c).a = (b.a) + (c.a)
156  -- 6.(13+17) = (6.13) + (6c.17
157 map OrDistribution in dnf {
158     check bexp(e1:And, e2:Or, a:Expr |
159         e2 <> a;
160         e1.expr->includes(e2);
161         e1.expr->includes(a); ) {}
162     enforce dnf() {
163         realize e3:Or, realize e4:And, realize e5:And |
164         e3.expr := Sequence{e4, e5};
165     }
166     where( ) {
167         realize or2and: OrDistribution, realize
                a2a_c:OrDistributionCopy,
168         realize disor2ands:Expr2MultExpr  |
169         or2and.beExpr := e1;
170         or2and.dnfExpr := e3;
171         or2and.ID := e1.ID;
172         or2and.beParentExpr := e1.parent;
173         e3.ID := or2and.ID;
174         a2a_c.beExpr := a;
```

```
175         a2a_c.dnfAndDist := if e1.expr->indexOf(a) = 1 then e5
176                             else e4
177                             endif;
178         a2a_c.ID := a.ID + '_c';
179         disor2ands.beExpr := e2;
180         disor2ands.dnfExprs := Sequence(exprMM::Not){e4, e5};
181         --disor2ands.ID := e2.ID;
182         e4.ID := e2.ID + '_1';
183         e5.ID := e2.ID + '_2';
184     }
185 }

187 -- Copy expressions in OrDistributive
188 map Expr2ExprCopy in dnf {
189     check bexp(a:Expr | ) { }
190     enforce dnf() {
191         realize adnf:Expr |
192     } where(e2e_c:OrDistributionCopy |
193         e2e_c.beExpr = a;
194     ) {
195         realize e2e_d:Expr2ExprCopy |
196         e2e_d.beExpr := a;
197         e2e_d.dnfExpr := adnf;
198         e2e_d.ID := a.ID + '_c';
199         adnf.ID := e2e_d.ID;
200         adnf.parent := e2e_c.dnfAndDist;
201     }
202 }

204 map Lit2LitCopy in dnf {
205     check bexp(a:Literal) { }
206     enforce dnf() {
207         realize adnf:Literal |
208     } where(e2e_c:OrDistributionCopy |
209         e2e_c.beExpr = a;
210     ) {
211         realize e2e_d:Lit2LitCopy |
212         e2e_d.beExpr := a;
213         e2e_d.dnfExpr := adnf;
214         e2e_d.ID := a.ID + '_c';
215         adnf.ID := e2e_d.ID;
216         adnf.parent := e2e_c.dnfAndDist;
217         e2e_d.name := a.name + '_c';
218         adnf.name := e2e_d.name;
219     }
220 }

222 -- Deep copy expressions
223 map Expr2ExprCopy_Rec in dnf  {
224     check bexp(a:Expr, aparent:Expr |
225         aparent.expr->includes(a);
226     ) { }
```

```
227     enforce dnf() {
228         realize adnf:Expr |
229     } where(e2ep:Expr2ExprCopy |
230         e2ep.beExpr = aparent;
231     ) {
232         realize e2e_r:Expr2ExprCopy |
233         e2e_r.beExpr := a;
234         e2e_r.dnfExpr := adnf;
235         e2e_r.ID := a.ID + '_rc';
236         adnf.ID := e2e_r.ID;
237         adnf.parent := e2ep.dnfExpr;
238     }
239 }

241 map Lit2LitCopy_Rec in dnf {
242     check bexp(a:Literal, aparent:Expr |
243             aparent.expr->includes(a); ) { }
244     enforce dnf() {
245         realize adnf:Literal |
246     } where(e2ep:Expr2ExprCopy |
247             e2ep.beExpr = aparent; ) {
248         realize e2e_r:Lit2LitCopy |
249         e2e_r.beExpr := a;
250         e2e_r.dnfExpr := adnf;
251         e2e_r.ID := a.ID + '_rc';
252         adnf.ID := e2e_r.ID;
253         adnf.parent := e2ep.dnfExpr;
254         e2e_r.name := a.name;
255         adnf.name := e2e_r.name;
256     }
257 }



261 map And2AndCopy in dnf refines Expr2ExprCopy {
262     check bexp(a:And) { }
263     enforce dnf() {
264         realize adnf:And |
265     } where() { }
266 }

268 map Or2OrCopy in dnf refines Expr2ExprCopy {
269     check bexp(a:Or) { }
270     enforce dnf() {
271         realize adnf:Or |
272     } where() { }
273 }

275 map Or2OrCopy_Rec in dnf refines Expr2ExprCopy_Rec {
276     check bexp(a:Or) { }
277     enforce dnf() {
278         realize adnf:Or |
```

```
279     } where() { }
280 }

282 map And2AndCopy_Rec in dnf refines Expr2ExprCopy_Rec {
283     check bexp(a:And) { }
284     enforce dnf() {
285         realize adnf:And |
286     } where() { }
287 }

289 map SetMultiParent in dnf {
290     check bexp(e:Expr |
291         not e.parent.oclIsUndefined(); ) { }
292     where (e2e:Expr2Expr, e2ep:Expr2MultExpr |
293             not e2e.beParentExpr.oclIsUndefined();
294             e2e.beExpr = e;
295             e2ep.beExpr = e2e.beParentExpr; ) {
296         e2e.dnfExpr.parent := let pos = e.parent.expr->indexOf(e) in
297                                 e2ep.dnfExprs->at(pos);
298     }
299 }

301 map SetParent in dnf {
302     check bexp(e:Expr |
303         not e.parent.oclIsUndefined(); ) { }
304     where (e2e:Expr2Expr, e2ep:Expr2Expr |
305             not e2e.beParentExpr.oclIsUndefined();
306             e2e.beExpr = e;
307             e2ep.beExpr = e2e.beParentExpr; ) {
308         e2e.dnfExpr.parent := e2ep.dnfExpr;
309     }
310 }

312 map SetRootParent in dnf {
313     check bexp(e:Expr |
314         e.parent.oclIsUndefined(); ) { }
315     enforce dnf (dnfExpr:Expr, root:BooleanExprs) {
316         root.hasExpr := root.hasExpr->including(dnfExpr);
317     }
318     where (e2e:Expr2Expr |
319             e2e.beExpr = e;
320             e2e.dnfExpr = dnfExpr;) {
321     }
322 }
```

8.10 Mi to Si

The example addresses the issue of transforming an UML Class hierarchy with multiple inheritance to a Java Class hierarchy with

single inheritance. The aim of this transformation is to generate, from an UML (simplified) class hierarchy, a Java (simplified) class hierarchy by defining a multiple-inheritance (MI) interface hierarchy corresponding to the MI UML class hierarchy and to establish implementation links between the class hierarchy and the implementation hierarchy.

Listing 8.10: Complete Mi2Si example.

```
13 import umlmmmi : 'umlMM.ecore'::umlmmmi;
14 import javammsi : 'javaMM.ecore'::javammsi;
15 import uml2java : 'uml2java.ecore'::umlmi2javasi;

17 transformation Mi2Si {
18     uml imports umlmmmi;
19     java imports javammsi;
20     imports uml2java;
21 }

23 map PackageToPackage in Mi2Si {
24   uml(pIn:Package) {}
25   enforce java () {
26       realize pOut:Package
27   }
28   where () {
29     realize p2p:Package2Package |
30     p2p.umlPackage := pIn;
31     p2p.javaPackage := pOut;
32   }
33 }

35 map ClassInPackage in Mi2Si {
36   check uml(p1:Package, c1:Class |
37         p1.containsClass->includes(c1);){
38     }
39     enforce java (p2:Package) {
40         realize c2:Class, realize i:Interface |
41         c2.implements := Sequence{i};
42         p2.containsClass := p2.containsClass->including(c2);
43         p2.containsIterface := p2.containsIterface->including(i);
44     }
45     where (p2p:Package2Package |
46     p2p.umlPackage = p1;
47     p2p.javaPackage = p2;) {
48         realize c2c:RClass2Class, realize c2i:Class2Interface |
49         c2i.umlClass := c1;
50         c2i.javaInterface := i;
51         c2c.umlClass := c1;
52         c2c.javaClass := c2;
53     }
```

```
54      map {
55          where () {
56              c2c.name := c1.name;
57              c2.name := c2c.name;
58              c2i.name := 'I' + c1.name;
59              i.name := c2i.name;
60          }
61      }
62  }

64  map ClassSuperToImplements in Mi2Si {
65      check uml(umlc1:Class, umlc2:Class |
66          umlc1.supers->includes(umlc2);
67      ) { }
68      enforce java (javac1:Class, javai2:Interface) {
69          javac1.implements := javac1.implements->including(javai2);
70      }
71      where (c2toi:Class2Interface, c12c:Class2Class |
72          c12c.umlClass = umlc1;
73          c12c.javaClass = javac1;
74          c2toi.umlClass = umlc2;
75          c2toi.javaInterface = javai2;
76      ) { }
77  }
```

8.11 Text Path Expression to Path Expression

The *Text Path Expression to Path Expression* is the first step of the PathExp to PetriNet transformation. It is a concrete syntax to abstract syntax transformation for Path Expressions. This transformation addresses the problem of going from a text representation of a Path Expression to an abstract syntax representation that is amenable to graphical representation.

Listing 8.11: Complete TextualPathExp to PathExp example.

```
1 import textPathMM : 'TextualPathExp.ecore'::textualPathExp;
2 import pathExpMM : 'PathExp.ecore'::PathExp;
3 import text2pathMM : 'Text2PathExp.ecore'::Text2PathExp;

5 transformation text2model {
6     text imports textPathMM;
7     path imports pathExpMM;
8     imports text2pathMM;
9 }
```

```
11 /* This mapping determines a one-to-one relation between a
12  * TextualPathExp and a PathExp.
13  */
14 map Main in text2model {
15     check text(tpe:TextualPathExp) { }
16     enforce path() {
17         realize pe:PathExp |
18     }
19     where () {
20         realize tpe2pe:TextPath2Path |
21         tpe2pe.textpathexp := tpe;
22         tpe2pe.pathexp := pe;
23     }
24     map {
25         where() {
26             tpe2pe.name := tpe.name;
27             pe.name := tpe2pe.name;
28         }
29     }
30 }

32 -- A PrimitiveTransition always generates a Transition
33 map PrimitiveTransition in text2model {
34     check text(tpe_pt:PrimitiveTrans) { }
35     enforce path(pe:PathExp) {
36         realize pe_t:Transition |
37         pe_t.PathExp := pe;
38     }
39     where (tpe2pe:TextPath2Path |
40             tpe2pe.pathexp = pe; ) {
41         realize t2t:PrimitiveTrans2SubPath |
42         t2t.textTrans := tpe_pt;
43         t2t.pathTrans := pe_t;
44     }
45     map {
46         where() {
47             t2t.name := tpe_pt.name;
48             pe_t.name := t2t.name;
49         }
50     }
51 }

53 -- A PrimitiveTrans that is not multiple generates a state,
54 -- additionally to the transition. The state is the target of the
55 -- transition
56 map SingleTransition in text2model refines PrimitiveTransition {
57     check text(not tpe_pt.isMultiple;) { }
58     enforce path() {
59         realize pe_ts:State |
60         pe_ts.PathExp := pe;
61         pe_t.target := pe_ts;
```

```
62      }
63      where() {
64          t2t.state := pe_ts;
65      }
66  }

68  -- The root transition also generates its source state
69  map RootTransition in text2model refines SingleTransition {
70      check text(tpe:TextualPathExp |
71              tpe.path.transitions->first() = tpe_pt; ) { }
72      enforce path() {
73          realize pe_ss:State |
74          pe_ss.PathExp := pe;
75          pe_t.source := pe_ss;
76      }
77  }

79  -- A Primitive in the root path
80  map TransitionInRoothPath in text2model refines SingleTransition {
81      check text(tpe:TextualPathExp |
82              getPath(tpe_pt) = tpe.path;
83              tpe.path.transitions->first() <> tpe_pt; ) { }
84      where() {
85          t2t.prevTrans := getPreviousTransition(tpe_pt, tpe.path);
86      }
87  }

89  -- A primitive transition in an altTran path it is not the last
90  -- transition in the path
91  map TransitionInAltPath in text2model refines SingleTransition {
92      check text(getPath(tpe_pt).transitions->last() <> tpe_pt;
93              AlternativeTrans.allInstances()
94                  ->exists(at | at.altPaths
95                      ->includes(getPath(tpe_pt))); ) { }
96      where() {
97          t2t.prevTrans :=
98              getPreviousTransition(tpe_pt, getPath(tpe_pt));
99      }
100 }

102 -- PrimitiveTrans in the root or alt path always have as source the
103 -- target of the previous transition
104 map TransitionSource in text2model {
105     check text(tpe_pt:PrimitiveTrans) { }
106     enforce path(pe_t:Transition, pe_s:State) {
107         pe_t.source := pe_s;
108     }
109     where(t2t:PrimitiveTrans2SubPath, t2t_prev:Tansition2Transition |
110             t2t.textTrans = tpe_pt;
111             t2t.prevTrans = t2t_prev.textTrans;
112             t2t.pathTrans = pe_t;
113             t2t_prev.state = pe_s; ) { }
```

```
114 }

116 -- PrimitiveTransitions that are multiple
117 map MultipleTransition in text2model refines PrimitiveTransition {
118     check text(tpe_pt.isMultiple;) { }
119     where() {
120         t2t.prevTrans :=
121             getPreviousTransition(tpe_pt, getPath(tpe_pt));
122         }
123 }

125 -- Source and target of a multiple transition are the same
126 map MultipleTransitionSource in text2model {
127     check text(tpe_pt:PrimitiveTrans ) { }
128     enforce path(pe_t:Transition, pe_s:State) {
129         pe_t.source := pe_s;
130         pe_t.target := pe_s;
131     }
132     where(t2t:PrimitiveTrans2SubPath, t2t_prev:Tansition2Transition |
133             t2t.textTrans = tpe_pt;
134             t2t.textTrans.isMultiple;
135             t2t.prevTrans = t2t_prev.textTrans;
136             t2t.pathTrans = pe_t;
137             t2t_prev.state = pe_s; ) { }
138 }

140 -- A primitive transition in an altTran path it is the last
141 -- transition in the path
142 map LastTransitionInAltPath in text2model
143         refines PrimitiveTransition {
144     check text(tpe_at : AlternativeTrans |
145             getPath(tpe_pt).transitions->last() = tpe_pt;
146             tpe_at.altPaths->includes(getPath(tpe_pt)); ) { }

148     where() {
149         realize t2t:LastPtimTransToSubPath |
150         t2t.prevTrans :=
151             getPreviousTransition(tpe_pt, getPath(tpe_pt));
152         t2t.pathOwner := tpe_at;
153     }
154 }

156 map LastTransitionInAltPathEnds in text2model {
157     check text(tpe_pt:PrimitiveTrans ) { }
158     enforce path(pe_t:Transition, pe_ss:State, pe_ts:State) {
159         pe_t.source := pe_ss;
160         pe_t.target := pe_ts;
161     }
162     where(t2t:LastPtimTransToSubPath, t2t_prev:Tansition2Transition,
163          altt2s:AltTrans2State |
164             t2t.prevTrans = t2t_prev.textTrans;
165             t2t.pathTrans = pe_t;
```

```
166             t2t_prev.state = pe_ss;
167             altt2s.textTrans = t2t.pathOwner;
168             altt2s.state = pe_ts; ) { }
169 }

171 -- This rule generates the State element that closes an input
172 -- AlternativeTransition element. The generated State is the one at
173 -- which the different alternative paths of the AlternativeTransition
174 -- join.
175 map AlternativeTransition in text2model  {
176     check text(tpe_at : AlternativeTrans) {}
177     enforce path(pe:PathExp) {
178         realize pe_s : State |
179         pe_s.PathExp := pe;
180     }
181     where(tpe2pe:TextPath2Path |
182         tpe2pe.pathexp = pe; ) {
183         realize t2t:AltTrans2State |
184         t2t.textTrans := tpe_at;
185         t2t.state := pe_s;
186     }
187 }

189 -- Get the path that contains a transition
190 query text2model::getPath(trans:textPathMM::Transition)
191         : textPathMM::Path {
192      textPathMM::Path.allInstances()
193         ->select(a | a.transitions->includes(trans))
194         ->asSequence()->first()
195 }

197 -- Determine if a transition is the first of a path
198 query text2model::isFirstOfPath(t:textPathMM::Transition) : Boolean {
199     let p : textPathMM::Path = getPath(t)
200         in t = p.transitions->first()
201 }

203 -- Returns the previous non-multiple PrimitiveTransition. If the
204 -- transition is the first of a path in an AltTransition, then the
205 -- previous transition is the one before the AltTransition.
206 query text2model::getPreviousTransition(
207         pt:textPathMM::PrimitiveTrans,
208         p : textPathMM::Path ) : textPathMM::Transition {

210     if isFirstOfPath(pt) then
211         let alt : textPathMM::AlternativeTrans =
212             textPathMM::AlternativeTrans.allInstances()
213             ->select(a | a.altPaths->includes(p))
214             ->asSequence()
215             ->first()
216         in let p2 : textPathMM::Path = getPath(alt)
217             in let i : Integer = p2.transitions->indexOf(alt)
```

```
218                 in let t : textPathMM::Transition =
219                     p2.transitions->at(i-1) in
220                     if t.isMultiple then
221                         getPreviousTransition(
222                             t.oclAsType(textPathMM::PrimitiveTrans),
223                             p2)
224                     else
225                         t
226                     endif
227     else
228         let i : Integer = p.transitions->indexOf(pt)
229             in let t : textPathMM::Transition =
230                 p.transitions->at(i-1) in
231                 if t.isMultiple then
232                     getPreviousTransition(
233                         t.oclAsType(textPathMM::PrimitiveTrans), p)
234                 else
235                     t
236                 endif
237     endif
238 }
```

8.12 Path Expression to Petri Net

The Path Expression to Petri Net example describes a transformation from a path expression to a Petri net. This Annex provides the complete transformation code of the whole transformation sequence that enables to produce an XML Petri net representation (in the PNML format) from a textual definition of a path expression.

Listing 8.12: Complete PathExpression to PetriNet example.

```
1 import pathExpMM : 'PathExp.ecore'::PathExp;
2 import petriNetMM : 'PetriNet.ecore'::PetriNet;
3 import path2petriMM : 'PathExp2PetriNet.ecore'::PathExp2PetriNet;

5 transformation path2petri {
6     path imports pathExpMM;
7     petri imports petriNetMM;
8     imports path2petriMM;
9 }

11 map Main in path2petri {
12     check path(pe:PathExp) {}
13     enforce petri() {
14         realize pn:PetriNet |
```

```
15          pn.name = 'path';
16      }
17      where() {
18          realize pe2pn:PathExp2PetriNet |
19          pe2pn.pathexp := pe;
20          pe2pn.petrinet := pn;
21          pe2pn.name := pe.name;
22          pn.name := pe2pn.name;
23      }
24  }

26  -- since incoming and outgoing have opposites. let the transition
        handle them
27  map StateToPlace in path2petri {
28      check path(s:State, pe:PathExp |
29          pe.states->includes(s);
30      ) { }
31      enforce petri(pn:PetriNet) {
32          realize p:Place |
33          p.name := '';   -- To provide same results as ATL
34          pn.places := pn.places->including(p);
35      }
36      where (pe2pn:PathExp2PetriNet |
37          pe2pn.pathexp = pe;
38          pe2pn.petrinet = pn;
39      ) {
40          realize s2p:State2Place |
41          s2p.state := s;
42          s2p.place := p;
43      }
44  }

46  map Transition2Transition in path2petri {
47      check path(pt:Transition, pe:PathExp, ss:State, ts:State |
48          pe.transitions->includes(pt);
49          pt.source = ss;
50          pt.target = ts;
51      ) { }
52      enforce petri(pn:PetriNet, sp:Place, tp:Place) {
53          realize nt:Transition, realize pn_ia:PlaceToTransArc,
54          realize pn_oa:TransToPlaceArc |
55              pn.transitions := pn.transitions->including(nt);
56              pn.arcs := pn.arcs->including(pn_ia);
57              pn.arcs := pn.arcs->including(pn_oa);
58              pn_ia.weight := 1;
59              pn_ia.source := sp;
60              pn_ia.target := nt;
61              pn_oa.weight := 1;
62              pn_oa.source := nt;
63              pn_oa.target := tp;
64      }
```

```
65      where (pe2pn:PathExp2PetriNet, ss2sp:State2Place,
            ts2tp:State2Place |
66          pe2pn.pathexp = pe;
67          pe2pn.petrinet = pn;
68          ss2sp.state = ss;
69          ss2sp.place = sp;
70          ts2tp.state = ts;
71          ts2tp.place = tp;
72      ) {
73          realize t2t:Trans2Trans, realize t2tInc:Trans2InArc,
74          realize t2tOut:Trans2OutArc  |
75          t2t.pathTrans := pt;
76          t2t.netTrans := nt;
77          t2tInc.source := ss2sp;
78          t2tInc.target := t2t;
79          t2tOut.source := t2t;
80          t2tOut.target := ts2tp;
81          t2t.name := pt.name;
82          nt.name := t2t.name;
83      }
84  }
```

8.13 Petri Net to PNML(XML)

The Petri Net to PNML(XML) example describes a transformation from the PetriNet domain to the XML domain. The XML model will provide an XML representation of the PetriNet in PNML format[1]. The transition is, as with the PathExp to PetriNet (Sect. 8.12), straight forward, with one mapping to transform elements of each of the PetriNet classes into it's XML representation. Note that the used XML metamodel is a simplified version of the XML specification.

Listing 8.13: Complete Petri Net to PNML(XML) example.

```
1 import petriNetMM : 'PetriNet.ecore'::PetriNet;
2 import xmlMM : 'XML.ecore'::XML;
3 import petri2xmlMM : 'PetriNet2XML.ecore'::PetriNet2XML;

5 transformation petri2xml {
6     petri imports petriNetMM;
7     xml imports xmlMM;
8     imports PetriNet2XML;
9 }
```

[1] http://www.pnml.org. last accessed 10/05/2017

```
11 /**
12  * Rule 'Main'
13  * This rule generates the "pnml" root tag from the input PetriNet
        element.
14  * This tag has an "xmlns" attribute and a "net" element as child
        element.
15  * The "net" tag has an "id", a "type" and a "name" attributes, and
        the
16  * following children elements:
17  * a "place" element for each Place of the input PetriNet model
18  * a "transition" element for each Transition of the input PetriNet
        model
19  * an "arc" element for each Arc of the input PetriNet model.
20  */
21 map Main in petri2xml {
22     check petri(pn:PetriNet) {}
23     enforce xml() {
24         realize root:Root,
25         realize xmlns:Attribute,
26         realize id:Attribute,
27         realize type:Attribute,
28         realize net:Element,
29         realize name:Element,
30         realize text:Element,
31         realize val:Text |
32         root.name := 'pnml';
33         root.children := Sequence{xmlns, net};
34         xmlns.name := 'xmlns';
35         xmlns.value :='http://www.example.org/pnpl';
36         net.name := 'net';
37         net.children := Sequence{id, type, name};
38         id.name := 'id';
39         id.value := 'n1';
40         type.name := 'type';
41         type.value := 'http://www.example.org/pnpl/PTNet';
42         name.name := 'name';
43         name.children := Sequence{text};
44         text.name := 'text';
45         text.children := Sequence{val};

47     }
48     where() {
49         realize pn2root:PetriNet2Root |
50         pn2root.petriNet := pn;
51         pn2root.root := root;
52         pn2root.xmlns := xmlns;
53         pn2root.id := id;
54         pn2root.type := type;
55         pn2root.net := net;
56         pn2root.name := name;
57         pn2root.text := text;
```

```
58          pn2root.val := val;
59      }
60      map {
61          where () {
62              val.value := pn.name;
63          }
64      }
65  }

67  map Place in petri2xml {
68      check petri(pn:PetriNet) {
69          pn_s : Place |
70          pn.places->includes(pn_s);
71      }
72      enforce xml(net:Element) {
73          realize xml_place : Element,
74          realize id : Attribute,
75          realize name : Element,
76          realize text : Element,
77          realize val : Text |
78          xml_place.name := 'place';
79          xml_place.children := Sequence{id, name};
80          id.name := 'id';
81          name.name := 'name';
82          name.children := Sequence{text};
83          text.name := 'text';
84          text.children := Sequence{val};
85          net.children := net.children->including(xml_place);
86      }
87      where(pn2root:PetriNet2Root |
88              pn2root.petriNet = pn;
89              pn2root.net = net; ) {
90          realize p2e:Place2Element |
91          p2e.place := pn_s;
92          p2e.xml_place := xml_place;
93          p2e.id := id;
94          p2e.name := name;
95          p2e.text := text;
96          p2e.val := val;

98      }
99      map {
100         where () {
101             id.value := pn.places->indexOf(pn_s).toString();
102             val.value := pn_s.name;
103         }
104     }
105 }

107 map Transition in petri2xml {
108     check petri (pn:PetriNet) {
109         pn_t : Transition |
```

```
110         pn.transitions->includes(pn_t);
111     }
112     enforce xml(net:Element) {
113         realize xml_trans : Element,
114         realize trans_id : Attribute |
115         xml_trans.name := 'transition';
116         xml_trans.children := Sequence{trans_id};
117         trans_id.name := 'id';
118         net.children := net.children->including(xml_trans);
119     }
120     where (pn2root:PetriNet2Root |
121             pn2root.petriNet = pn;
122             pn2root.net = net;) {
123         realize t2e:Transition2Element |
124         t2e.xml_trans := xml_trans;
125         t2e.trans_id := trans_id;
126     }
127     map {
128         where() {
129             trans_id.value := pn.places->size().toString() +
130                     pn.transitions->indexOf(pn_t).toString();
131         }
132     }
133 }

135 map PlaceToTransArc in petri2xml {
136     check petri(pn:PetriNet) {
137         pn_a:PlaceToTransArc |
138         pn.arcs->includes(pn_a);
139     }
140     enforce xml(net:Element) {
141         realize xml_arc:Element,
142         realize id:Attribute,
143         realize source:Attribute,
144         realize target:Attribute |
145         xml_arc.name := 'arc';
146         xml_arc.children := Sequence{id, source, target};
147         id.name := 'id';
148         net.children := net.children->including(xml_arc);
149     }
150     where(pn2root:PetriNet2Root |
151             pn2root.petriNet = pn;
152             pn2root.net = net;) {
153         realize p2a:Place2TransArc |
154         p2a.pn_a := pn_a;
155         p2a.xml_arc := xml_arc;
156         p2a.id := id;
157         p2a.source := source;
158         p2a.target := target;
159     }
160     map {
161         where() {
```

```
162             id.value := pn.places->size().toString() +
163                     pn.transitions->size().toString() +
164                     pn.arcs->indexOf(pn_a).toString();
165             source.name := 'source';
166             source.value :=
                    pn.places->indexOf(pn_a.source).toString();
167             target.name := 'target';
168             target.value := pn.places->size().toString() +
169                     pn.transitions->indexOf(pn_a.target).toString();
170         }
171     }
172 }



176 map TransToPlaceArc in petri2xml {
177     check petri(pn:PetriNet) {
178         pn_a:TransToPlaceArc |
179         pn.arcs->includes(pn_a);
180     }
181     enforce xml(net:Element) {
182         realize xml_arc:Element,
183         realize id:Attribute,
184         realize source:Attribute,
185         realize target:Attribute |
186         xml_arc.name := 'arc';
187         xml_arc.children := Sequence{id, source, target};
188         id.name := 'id';
189         net.children := net.children->including(xml_arc);
190     }
191     where(pn2root:PetriNet2Root |
192             pn2root.petriNet = pn;
193             pn2root.net = net;) {
194        realize t2p:Trans2PlaceArc |
195        t2p.xml_arc := xml_arc;
196        t2p.id := id;
197        t2p.source := source;
198        t2p.target := target;
199     }
200     map {
201         where() {
202             id.value := pn.places->size().toString() +
203                 pn.transitions->size().toString() +
204                 pn.arcs->indexOf(pn_a).toString();
205             source.name := 'source';
206             source.value := pn.places->size().toString() +
207                 pn.transitions->indexOf(pn_a.source).toString();
208             target.name := 'target';
209             target.value :=
                    (pn.places->indexOf(pn_a.target)).toString();
210         }
211     }
```

```
212 }
```

8.14 Railway to Control

The example is based on the train benchmark metamodel [102], which the authors claim contains the most typical class diagram constructs. The transformation considers that the train domain can be used model a toy model train, and we are interested in automate the signals (semaphores) and switches that control the toy model train behaviour. The transformation is used to create a model of a control system that can be used to monitor the state of the toy model train and, for example, automate the toy railway to allow multiple trains to run on it.

Listing 8.14: Complete Railway to Control example.

```
1 import railway : 'Railway.ecore#/';
2 import control : 'Control.ecore#/';
3 import railway2control : 'Railway2Control.ecore#/';

5 transformation r2c {
6     rail imports railway;
7     control imports control;
8     imports railway2control;
9 }

11 map railway2control in r2c {

13     check rail (r:RailwayContainer) { }
14     enforce control () {
15         realize c:RailwayControl |
16     }
17     where () {
18         realize r2c:Railway2Control |
19         r2c.railway := r;
20         r2c.control := c;
21     }
22 }

24 map route2route in r2c {
25     check rail (cont:RailwayContainer, rr:Route |
26         cont.routes->includes(rr);) { }
27     enforce control(ctrl:RailwayControl) {
28         realize cr:Route |
29         cr.controledBy := ctrl;
30     }
```

```
31      where (r2c:Railway2Control |
32              r2c.railway = cont;
33              r2c.control = ctrl;
34      ) {
35          realize r2r:Route2Route |
36          r2r.railwayRoute := rr;
37          r2r.controlRoute := cr;
38      }
39      map {
40          where() {
41              r2r.id := rr.id;
42              r2r.id := cr.id;
43              rr.id := r2r.id;
44              cr.id := r2r.id;
45          }
46      }
47  }

49  map semaphore2signal in r2c {
50      check rail (rc:RailwayContainer, r:Region, seg:Segment,
            s:Semaphore |
51          rc.regions->includes(r);
52          r.elements->includes(seg);
53          seg.semaphores->includes(s);
54          ) { }
55      enforce control(c:RailwayControl) {
56          realize ss:SemaphoreSignal |
57          ss.controledBy := c;
58      }
59      where (r2c:Railway2Control |
60          r2c.railway = r;
61          r2c.control = c;
62      ) {
63          realize s2s:Semaphore2Signal |
64          s2s.semaphore := s;
65          s2s.signal := ss;
66      }
67      map {
68          where() {
69              s2s.id := s.id;
70              s2s.id := ss.id;
71              s.id := s2s.id;
72              s.id := s2s.id;
73              ss.state :=
74                  if s.signal = railway::Signal::FAILURE then
75                      control::Light::RED
76                  else
77                      if s.signal = railway::Signal::STOP then
78                          control::Light::RED
79                      else
80                          control::Light::GREEN
81                      endif
```

```
82                  endif;
83          }
84      }
85  }


88  map sensor2section in r2c {
89      check rail (rr:Route, s:Sensor |
90          rr.gathers->includes(s);) { }
91      enforce control(cr:Route) {
92          realize ts:TrackSection |
93          ts.partOf := cr;
94      }
95      where (r2r:Route2Route |
96              r2r.railwayRoute = rr;
97              r2r.controlRoute = cr;
98      ) {
99          realize s2s:Sensor2Section |
100         s2s.sensor := s;
101         s2s.section := ts;
102     }
103     map {
104         where() {
105             ts.trackElementsIds := getSensorElementsIds(s);
106             ts.id := s.id;
107         }
108     }
109 }

111 map switchPos2signal in r2c {
112     check rail (rr:Route) {
113         sp:SwitchPosition |
114         sp.route = rr;
115     }
116     enforce control(cr:Route) {
117         realize ss:SwitchSignal |
118         ss.route := cr;
119     }
120     where (r2r:Route2Route |
121             r2r.railwayRoute = rr;
122             r2r.controlRoute = cr;) {
123         realize s2s:Switch2Signal |
124         s2s.position := sp;
125         s2s.signal := ss;
126     }
127     map {
128         where() {
129             ss.switchId := sp.target.id;
130             ss.position :=
131                 if sp.position = railway::Position::FAILURE then
132                     control::Position::FAILURE
133                 else
```

```
134                     if sp.position = railway::Position::STRAIGHT then
135                         control::Position::STRAIGHT
136                     else
137                         control::Position::DIVERGING
138                     endif
139                 endif;

141         }
142     }
143 }

145 map semaphore2join in r2c {
146     check rail (rc:RailwayContainer, er:Route, jr:Route, s:Semaphore
            |
147         rc.routes->includes(er);
148         rc.routes->includes(jr);
149         er.exit = s;
150         jr.entry = s;) { }
151     enforce control(c:RailwayControl, ec:Route, jc:Route) {
152         ec.joins := jc;
153     }
154     where (r2c:Railway2Control, er2c:Route2Route, jr2c:Route2Route |
155             er2c.railwayRoute = er;
156             er2c.controlRoute = ec;
157             jr2c.railwayRoute = jr;
158             jr2c.controlRoute = jc;
159             r2c.railway = rc;
160             r2c.control = c;
161     ) { }
162 }

164 map semaphore2fork in r2c {
165     check rail (r:RailwayContainer, sr:Route, fr1:Route, fr2:Route,
            s:Semaphore |
166         r.routes->includes(sr);
167         r.routes->includes(fr1);
168         r.routes->includes(fr2);
169         sr.exit = s;
170         fr1.entry = s;
171         fr2.entry = s;) {

173     }
174     enforce control(c:RailwayControl, sc:Route, fc1:Route,
            fc2:Route) {
175         sc.forks := Sequence{fc1, fc2};
176     }
177     where (r2c:Railway2Control, er2c:Route2Route,
178             fr2c1:Route2Route, fr2c2:Route2Route |
179             r2c.railway = r;
180             r2c.control = c;
181             er2c.railwayRoute = sr;
182             er2c.controlRoute = sc;
```

```
183             fr2c1.railwayRoute = fr1;
184             fr2c1.controlRoute = fc1;
185             fr2c2.railwayRoute = fr2;
186             fr2c2.controlRoute = fc2;

188     ) { }
189 }

191 query r2c::getSensorElementsIds(sensor : railway::Sensor) :
        Sequence(Integer)
192 {
193     sensor.monitors.id
194 }
```

8.15 XSLT to XQuery

The *XSLT to XQuery* example is part of the ATL Zoo, available online from the ATL Zoo website[2], describes a simplified transformation of XSLT code to XQuery code [16]. XSL (eXtensible Stylesheet Language) is a styling language for XML, and XSLT stands for XSL Transformations. XSLT can be used to transform XML documents into other formats, such as HTML. XQuery is a language designed to query XML documents in order to find and extract elements and attributes. The transformation aims at representing the XSLT as a series of XQueries.

Listing 8.15: Complete XSLT to XQuery example.

```
1  import XQuery : 'XQuery.ecore#/';
2  import XSLT : 'XSLT.ecore#/';
3  import XSLT2XQuery : 'XSLT2XQuery.ecore'::XSLT2XQuery;

5  transformation XSLT2XQuery
6  {
7      xs imports XSLT;
8      xq imports XQuery;
9      imports XSLT2XQuery;
10 }

12 map P2P in XSLT2XQuery {
13     xs(xslt:XSLTRootNode) { }
14     enforce xq() {
15         realize out:XQueryProgram,
```

[2] https://www.eclipse.org/atl/atlTransformations/#XSLT2XQuery, last visited 08-02-2017.

```
16          realize flwor:FLWOR,
17          realize for:For,
18          realize forExpression:XPath,
19          realize return:Return |
20          flwor.xQueryProgram := out;
21          flwor.for := for;
22          flwor.return := return;
23          for.expression := forExpression;
24      }
25      where () {
26          realize xslt2program:XSLTRootNode2XQueryProgram |
27          xslt2program.xslt := xslt;
28          xslt2program.out := out;
29          xslt2program.flwor := flwor;
30          xslt2program.for := for;
31          xslt2program.forExpression := forExpression;
32          xslt2program.return := return;
33          for.var := '$var';
34          forExpression.value := 'document(\"xmlFile.xml\")/*';
35      }
36  }

38  map Template2FLOWR in XSLT2XQuery {
39      xs(xslt:XSLTRootNode, template:Template |
40                  xslt.nodes->includes(template);
41                  template.match <> '/'; ) { }
42      enforce xq(prog:XQueryProgram) {
43          realize out:FunctionDeclaration,
44          realize flwor:FLWOR,
45          realize for:For,
46          realize forExpression:XPath,
47          realize return:Return |
48          out.expression := Sequence {flwor};
49          out.xQueryProgram := prog;
50          flwor.xQueryProgram := out;
51          flwor.for := for;
52          flwor.return := return;
53          for.expression := forExpression;
54          forExpression.value := '$paramVar';
55      }
56      where(xslt2program:XSLTRootNode2XQueryProgram |
57                  xslt2program.xslt = xslt;
58                  xslt2program.out = prog; ) {
59          realize template2flwor:Template2FLOWR |
60          template2flwor.template := template;
61          template2flwor.out := out;
62          template2flwor.flwor := flwor;
63          template2flwor.for := for;
64          template2flwor.forExpression := forExpression;
65          template2flwor.return := return;
66          template2flwor.match := template.match;
67          out.name := 'fct' + template.match;
```

```
69      }
70  }

72  map NodeFrom_Root in XSLT2XQuery {
73    xs(xslt:XSLTRootNode, t:Template |
74          t.parentNode = xslt;
75          t.match = '/'; ) { }
76    enforce xq(root_return:Return) { }
77    where(xslt2program:XSLTRootNode2XQueryProgram |
78          xslt2program.xslt = xslt;
79          xslt2program.return = root_return; ) { }
80  }

82  map NodeFrom_Template in XSLT2XQuery {
83    xs(t:Template |
84          t.match <> '/'; ) { }
85    enforce xq(t_return:Return) { }
86    where(template2flwor:Template2FLOWR |
87          template2flwor.template = t;
88          template2flwor.return = t_return; ) { }
89  }

91  map NodeFrom_ElementNode in XSLT2XQuery {
92      xs(elementNode:ElementNode | ) { }
93      enforce xq(parentNode:ElementNode) { }
94      where(fn:ElementNode2ElementNode |
95                  fn.out = parentNode; ) { }
96  }

98  map FromNode in XSLT2XQuery {
99    where () { realize fn:FromNode }
100 }

102 map NestedNode in XSLT2XQuery refines FromNode {
103     xs(parent:Node, node:Node |
104         node.parentNode = parent;
105     ) { }
106     where (fno:FromNodeOwner | ) {
107         fn.node := node;
108         fn.owner := fno;
109     }
110 }

112 map If2FLOWR in XSLT2XQuery refines FromNode {
113     enforce xq() {
114         realize out:FLWOR,
115         realize varlet:Let,
116         realize letExpression:XPath,
117         realize _where:Where,
118         realize whereExpression:BooleanExp,
119         realize return:Return |
```

```
120         out._let := varlet;
121         out._where := _where;
122         out.return := return;
123         varlet.expression := letExpression;
124         varlet.var := '$var';
125         letExpression.value := '$var';
126         _where.expression := whereExpression;
127     }
128     where() {
129         realize fn:If2FLOWR |
130         fn.exp := out;
131         fn.varlet := varlet;
132         fn.letExpression := letExpression;
133         fn._where := _where;
134         fn.whereExpression := whereExpression;
135     }
136 }

138 map If2FLOWR_Top in XSLT2XQuery refines If2FLOWR {
139     xs(node: If |
140         node.parentNode.oclIsTypeOf(XSLT::Template); ) { }
141     where( ) {
142       fn.node := node;
143       whereExpression.value := '$var/' + node.test;
144     }
145 }

147 map If2FLOWR_FromRoot in XSLT2XQuery refines If2FLOWR_Top,
      NodeFrom_Root {
148         xs(node.parentNode = t; ) { }
149     enforce xq() {
150         out.returnEx := root_return;
151         }
152 }

154 map If2FLOWR_FromTemplate in XSLT2XQuery refines If2FLOWR_Top,
      NodeFrom_Template {
155         xs(node.parentNode = t; ) { }
156         enforce xq() {
157         out.returnEx := t_return;
158         }
159 }

161 map If2FLOWR_FromElementNode in XSLT2XQuery refines If2FLOWR_Top,
      NodeFrom_ElementNode {
162         xs(node.parentNode = elementNode; ) { }
163         enforce xq() {
164         out.parentNode := parentNode;
165         }
166 }

168 map If2FLOWR_Recursive in XSLT2XQuery refines If2FLOWR, NestedNode {
```

```
169     xs(node: If |
170         not node.parentNode.oclIsTypeOf(XSLT::Template); ) {
171     }
172     enforce xq(fno_return:Return |) { }
173     where(fno.return = fno_return;) {
174         out.returnEx := fno_return;
175         whereExpression.value := '$var/' + node.test;
176     }
177 }

179 map ApplyTemplate2FunctionCall in XSLT2XQuery refines FromNode {
180         xs() { }
181         enforce xq() {
182     realize out:FunctionCall,
183     realize parameter:XPath |
184       out.parameters := Sequence{ parameter };
185   }
186   where() {
187     realize fn:ApplyTemplate2FunctionCall |
188     fn.out := out;
189     fn.parameter := parameter;
190   }
191 }

193 map ApplyTemplate2FunctionCall_Top in XSLT2XQuery refines
      ApplyTemplate2FunctionCall {
194     xs(node: ApplyTemplates |
195         node.parentNode.oclIsTypeOf(XSLT::Template);
196     ) { }
197     where( ) {
198       fn.node := node;
199       out.name := 'fct' + node.select;
200       parameter.value := '$var/' + node.select;
201     }
202 }

204 map ApplyTemplate2FunctionCall_FromRoot in XSLT2XQuery refines
      ApplyTemplate2FunctionCall_Top, NodeFrom_Root {
205         xs(node.parentNode = t; ) { }
206         enforce xq() {
207         out.returnEx := root_return;
208         }
209 }

211 map ApplyTemplate2FunctionCall_FromTemplate in XSLT2XQuery refines
      ApplyTemplate2FunctionCall_Top, NodeFrom_Template {
212         xs(node.parentNode = t;    ) { }
213         enforce xq() {
214         out.returnEx := t_return;
215         }
216 }
```

```
218 map ApplyTemplate2FunctionCall_FromElementNode in XSLT2XQuery
      refines ApplyTemplate2FunctionCall_Top, NodeFrom_ElementNode {
219         xs(node.parentNode = elementNode; ) { }
220         enforce xq() {
221         out.parentNode := parentNode;
222         }
223 }

225 map ApplyTemplate2FunctionCall_Recursive in XSLT2XQuery refines
      ApplyTemplate2FunctionCall, NestedNode {
226     xs(node: ApplyTemplates |
227         not node.parentNode.oclIsTypeOf(XSLT::Template); ) { }
228     enforce xq(fno_return:Return |) { }
229     where( fno.return = fno_return; ) {
230         out.returnEx := fno_return;
231         out.name := 'fct' + node.select;
232         parameter.value := '$var/' + node.select;
233     }
234 }

236 map ValueOf2ReturnXPath in XSLT2XQuery refines NestedNode {
237         xs(node:ValueOf) {}
238         enforce xq() {
239         realize out:ReturnXPath |
240         }
241         where() {
242         out.value := '$var/' + node.select;
243         }
244 }

246 map ElementNode2ElementNode in XSLT2XQuery refines FromNode {
247           xs( ) {}
248           enforce xq() {
249                 realize out:ElementNode |
250           }
251           where() {
252                 realize fn:ElementNode2ElementNode |
253                 fn.out := out;
254           }
255 }

257 map ElementNode2ElementNode_Top in XSLT2XQuery refines
      ElementNode2ElementNode {
258             xs(node:ElementNode |
259                           (node.name <> 'xsl:otherwise') and
260                           (node.name <> 'xsl:when') and
261                           (node.name <> 'xsl:choose') and
262                           (node.name <> 'xsl:copy-of') and
263                           (node.name <> 'xsl:sort') and
264                           (node.name <> 'xsl:foreach') and
265                           (node.name <> 'xsl:if') and
266                           (node.name <> 'xsl:apply-templates') and
```

```
267                                 (node.name <> 'xsl:value-of') and
268                                 (node.name <> 'xsl:template') and
269                                 (node.name <> 'xsl:stylesheet');
270                                 node.parentNode.oclIsTypeOf(XSLT::Template);
                                        ) { }
271             where( ) {
272               fn.node := node;
273               out.name := node.name;
274             }
275 }

277 map ElementNode2ElementNode_FromRoot in XSLT2XQuery refines
        ElementNode2ElementNode_Top, NodeFrom_Root {
278           xs(node.parentNode = t; ) { }
279           enforce xq() {
280             out.returnEx := root_return;
281           }
282 }

284 map ElementNode2ElementNode_FromTemplate in XSLT2XQuery refines
        ElementNode2ElementNode_Top, NodeFrom_Template {
285         xs(node.parentNode = t; ) { }
286         enforce xq() {
287             out.returnEx := t_return;
288         }
289 }

291 map ElementNode2ElementNode_FromElementNode in XSLT2XQuery refines
        ElementNode2ElementNode_Top, NodeFrom_ElementNode {
292   xs(node.parentNode = elementNode;
293   ) { }
294   enforce xq() {
295     out.parentNode := parentNode;
296   }
297 }

299 map ElementNode2ElementNode_Recursive in XSLT2XQuery refines
        ElementNode2ElementNode, NestedNode {
300     xs(node:ElementNode |
301     (node.name <> 'xsl:otherwise') and
302     (node.name <> 'xsl:when') and
303     (node.name <> 'xsl:choose') and
304     (node.name <> 'xsl:copy-of') and
305     (node.name <> 'xsl:sort') and
306     (node.name <> 'xsl:foreach') and
307     (node.name <> 'xsl:if') and
308     (node.name <> 'xsl:apply-templates') and
309     (node.name <> 'xsl:value-of') and
310     (node.name <> 'xsl:template') and
311     (node.name <> 'xsl:stylesheet');
312     not node.parentNode.oclIsTypeOf(XSLT::Template);
313     ) { }
```

```
314     enforce xq(fno_return:Return |) {
315     }
316     where(
317         fno.return = fno_return;
318     ) {
319         out.returnEx := fno_return;
320     }
321 }

323 map Attribute2Attribute in XSLT2XQuery refines FromNode {
324     xs() {}
325     enforce xq() {
326         realize out:AttributeNode |
327     }
328     where () {
329         realize fn:Attribute2Attribute |
330         fn.exp := out;

332     }
333 }

335 map Attribute2Attribute_Top in XSLT2XQuery refines
        Attribute2Attribute {
336     xs(node: AttributeNode |
337         (node.name <> 'xsl:otherwise') and
338         (node.name <> 'xsl:when') and
339         (node.name <> 'xsl:choose') and
340         (node.name <> 'xsl:copy-of') and
341         (node.name <> 'xsl:sort') and
342         (node.name <> 'xsl:foreach') and
343         (node.name <> 'xsl:if') and
344         (node.name <> 'xsl:apply-templates') and
345         (node.name <> 'xsl:value-of') and
346         (node.name <> 'xsl:template') and
347         (node.name <> 'xsl:stylesheet') ;
348     ) { }
349     where( ) {
350       fn.node := node;
351       out.name := node.name;
352       out.value := node.value;
353     }
354 }

356 map Attribute2Attribute_FromRoot in XSLT2XQuery refines
        Attribute2Attribute_Top, NodeFrom_Root {
357   xs(node.parentNode = t;
358   ) { }
359   enforce xq() {
360     out.returnEx := root_return;
361   }
362 }
```

```
364 map Attribute2Attribute_FromTemplate in XSLT2XQuery refines
        Attribute2Attribute_Top, NodeFrom_Template {
365   xs(node.parentNode = t;
366   ) { }
367   enforce xq() {
368     out.returnEx := t_return;
369   }
370 }

372 map Attribute2Attribute_FromElementNode in XSLT2XQuery refines
        Attribute2Attribute_Top, NodeFrom_ElementNode {
373   xs(node.parentNode = elementNode;
374   ) { }
375   enforce xq() {
376     out.parentNode := parentNode;
377   }
378 }
```

9

EMG Model Generation Scripts

9.1 Abstract to Concrete

The example describes a transformation of a simplified UML model to another simplified UML model. The aim of this transformation is to generate, from a source UML model, another UML model that flattens the inherited operations of a class. That is, a `Class` in the target model will collect all the operations inherited from the closure of it's super classes that are abstract.

Listing 9.1: Model generator code for the Abstract to Concrete example.

```
1  pre {
2  /**
3          For convenience make total a mult of 3 > 10
4     p = # of Packages
5     c = # classes
6     o = # operations
7     r = # parameters
8     d = # PrimitiveDataTypes
9   */
10         var p:Integer = total/10;
11         // Three random numbers 0..total
12         var quant = nextAddTo(3, total);
13         var c;
14         var o;
15         if (quant.at(0) > quant.at(1)) {
16                 c = quant.at(1);
```

```
17                  o = quant.at(0)-c;
18          }
19          else {
20                  c = quant.at(0);
21                  o = quant.at(1)-c;
22          }
23          var r = quant.at(2);
24  }

26  $instances p
27  @list ps
28  operation Package create()
29  {
30          self.name = 'P_' + nextCamelCaseString(20, 5);
31  }

33  $instances p
34  @list dts
35  operation PrimitiveDataType create()
36  {
37          self.name = 'String';
38          self.owner = nextFromListAsSample("ps");
39  }

41  $instances c
42  @list clss
43  operation Class create()
44  {
45          self.name = 'C_' + nextCamelCaseString(20, 5);
46          self.isAbstract = nextBoolean();
47          self.owner = nextFromList("ps");
48          var op = new Operation();
49          if (nextBoolean()) {
50                  op.name = 'get' + nextCamelCaseString(20, 5);
51          }
52          else {
53                  op.name = 'set' + nextCamelCaseString(20, 5);
54          }
55          op.owner = self;
56  }

58  $instances o
59  operation Operation create()
60  {
61          if (nextBoolean()) {
62                  self.name = 'get' + nextCamelCaseString(20, 5);
63          }
64          else {
65                  self.name = 'set' + nextCamelCaseString(20, 5);
66          }
67          self.owner = nextFromList("clss");
68  }
```

```
70 $instances r
71 operation Parameter create()
72 {
73         self.name = 'R_' + nextCamelCaseString(20, 5);
74         self.‘operation‘ = nextFromCollection(Operation.all());
75         self.type :=
               self.‘operation‘.owner.owner.ownedTypes.selectOne(t |
               t.isTypeOf(PrimitiveDataType));
76 }


79 $probability 0.3
80 pattern ClassInheritance
81         p:Package,
82         cc1 : Class
83           from: p.ownedTypes,
84         ac : Class
85           from: p.ownedTypes
86           guard: ac.isAbstract = true and ac <> cc1
87         {
88                 onmatch {
89                         cc1.inheritsFrom = ac;
90                 }
91         }

93 post {
94         "GenDone".println();
95 }
```

9.2 BibTeXML to DocBook

The BibTeXML to DocBook example describes a transformation of a BibTeXML model to a DocBook model. BibTeXML [84] is an XML-based format for the BibTeX bibliographic tool. DocBook [107] is an XML-based format for document composition.

Listing 9.2: Model generator code for the BibTeXML to DocBook example.

```
1 pre {
2         var id:Integer = 1;
3         var num_files:Integer = total/10;
4         var num_art = 0;
5         var num_tech = 0;
6         var num_unpub = 0;
7         var num_man = 0;
8         var num_proc = 0;
```

```
9         var num_inproc = 0;
10        var num_booklet = 0;
11        var num_book = 0;
12        var num_coll = 0;
13        var num_inbook = 0;
14        var num_misc = 0;
15        var num_pthesis = 0;
16        var num_mthesis = 0;
17        while (total > 0) {
18                var pick = nextInteger(12);
19                switch (pick) {
20                        case 0: num_art += 1;
21                        case 1: num_tech += 1;
22                        case 2: num_unpub += 1;
23                        case 3: num_man += 1;
24                        case 4: num_proc += 1;
25                        case 5: num_inproc += 1;
26                        case 6: num_booklet += 1;
27                        case 7: num_book += 1;
28                        case 8: num_coll += 1;
29                        case 9: num_inbook += 1;
30                        case 10: num_misc += 1;
31                        case 11: num_pthesis += 1;
32                        case 12: num_mthesis += 1;
33                }
34                total -= 1;
35        }
36        var auth = (num_art + num_tech + num_unpub + num_inproc +
37                                num_book + num_coll + num_inbook +
38                                num_pthesis + num_mthesis) * 2;
39        var journals:Sequence = Sequence{"Academy of Management
          Journal","Academy of Management Review","Accounting,
          Organizations and Society","Administrative Science
          Quarterly","American Economic Review","Contemporary
          Accounting Research","Econometrica","Entrepreneurship
          Theory and Practice","Harvard Business Review","Human
          Relations","Human Resource Management","Information
          Systems Research","Journal of Accounting and
          Economics","Journal of Accounting Research","Journal of
          Applied Psychology","Journal of Business
          Ethics","Journal of Business Venturing","Journal of
          Consumer Psychology","Journal of Consumer
          Research","Journal of Finance","Journal of Financial and
          Quantitative Analysis","Journal of Financial
          Economics","Journal of International Business
          Studies","Journal of Management","Journal of Management
          Information Systems","Journal of Management
          Studies","Journal of Marketing","Journal of Marketing
          Research","Journal of Operations Management","Journal of
          Political Economy","Journal of the Academy of Marketing
          Science","Management Science","Manufacturing and Service
          Operations Management","Marketing Science","MIS
```

```
            Quarterly","Operations Research","Organization
            Science","Organization Studies","Organizational Behavior
            and Human Decision Processes","Production and Operations
            Management","Quarterly Journal of Economics","Research
            Policy","Review of Accounting Studies","Review of
            Economic Studies","Review of Finance","Review of
            Financial Studies","Sloan Management Review","Strategic
            Entrepreneurship Journal","Strategic Management
            Journal","The Accounting Review"};
}

$instances num_files
@list files
operation BibTeXFile create()
{
}

$instances num_art
operation Article create()
{
        self.id = id.asString();
        id += 1;
        self.title = nextCamelCaseString(20, 5);
        self.year = nextInteger(1950, 2016).asString();
        self.journal = nextFromCollection(journals);
}

$instances num_tech
operation TechReport create()
{
        self.id = id.asString();
        id += 1;
        self.title = nextCamelCaseString(20, 5);
        self.year = nextInteger(1950, 2016).asString();
}

$instances num_unpub
operation Unpublished create() {
        self.id = id.asString();
        id += 1;
        self.title = nextCamelCaseString(20, 5);
        self.note = nextString("LETTER_LOWER", 20);
}

$instances num_man
operation Manual create() {
        self.id = id.asString();
        id += 1;
        self.title = nextCamelCaseString(20, 5);
}

$instances num_proc
```

```
83 operation Proceedings create()
84 {
85         self.id = id.asString();
86         id += 1;
87         self.title = nextCamelCaseString(20, 5);
88         self.year = nextInteger(1950, 2016).asString();
89 }

91 $instances num_inproc
92 operation InProceedings create()
93 {
94         self.id = id.asString();
95         id += 1;
96         self.title = nextCamelCaseString(20, 5);
97         self.year = nextInteger(1950, 2016).asString();
98         self.booktitle = nextCamelCaseString(20, 5);
99 }

101 $instances num_booklet
102 operation Booklet create()
103 {
104         self.id = id.asString();
105         id += 1;
106         self.year = nextInteger(1950, 2016).asString();
107 }

109 $instances num_book
110 operation Book create()
111 {
112         self.id = id.asString();
113         id += 1;
114         self.title = nextCamelCaseString(20, 5);
115         self.year = nextInteger(1950, 2016).asString();
116         self.publisher = nextCamelCaseString(20, 5);
117 }

119 $instances num_coll
120 operation InCollection create()
121 {
122         self.id = id.asString();
123         id += 1;
124         self.title = nextCamelCaseString(20, 5);
125         self.year = nextInteger(1950, 2016).asString();
126         self.publisher = nextCamelCaseString(20, 5);
127         self.booktitle = nextCamelCaseString(20, 5);
128 }

130 $instances num_inbook
131 operation InBook create()
132 {
133         self.id = id.asString();
134         id += 1;
```

```
135         self.title = nextCamelCaseString(20, 5);
136         self.year = nextInteger(1950, 2016).asString();
137         self.publisher = nextCamelCaseString(20, 5);
138         self.chapter = nextInteger(10);
139 }


142 $instances num_misc
143 operation Misc create()
144 {
145         self.id = id.asString();
146         id += 1;
147 }

149 $instances num_pthesis
150 operation PhDThesis create()
151 {
152         self.id = id.asString();
153         id += 1;
154         self.title = nextCamelCaseString(20, 5);
155         self.year = nextInteger(1950, 2016).asString();
156         self.school = nextCamelCaseString(20, 5);
157 }

159 $instances num_mthesis
160 operation MasterThesis create()
161 {
162         self.id = id.asString();
163         id += 1;
164         self.title = nextCamelCaseString(20, 5);
165         self.year = nextInteger(1950, 2016).asString();
166         self.school = nextCamelCaseString(20, 5);
167 }

169 $instances AuthoredEntry.all().size() * 2
170 @list authors
171 operation Author create()
172 {
173         self.author = nextCapitalisedString("LETTER_LOWER", 8) +
174                         ", " + nextCapitalisedString("LETTER_LOWER",
                                    8);
175 }

177 pattern BibFileItems
178         e:BibTeXEntry
179         {
180                 onmatch {
181                         nextFromList("files").entries.add(e);
182                 }
183         }

185 pattern AuthoredEntryFirst
```

```
186         ae:AuthoredEntry
187         {
188                 onmatch {
189                         // At least one author
190                         ae.authors.add(nextFromListAsSample("authors"));
191                 }
192         }

194 pattern AuthoredEntryMore
195         ae:AuthoredEntry
196         {
197                 onmatch {
198                         var more = nextInteger(2);
199                         for (i in 1.to(more)) {
200                                 var other =
                                        nextFromListAsSample("authors");
201                                 if (other <> null)
202                                         ae.authors.add(other);
203                         }
204                 }
205         }


208 post {
209         "GenDone".println();
210 }
```

9.3 DNF

The example describes a transformation of a simplified UML model to another simplified UML model. The aim of this transformation is to generate, from a source UML model, another UML model that flattens the inherited operations of a class. That is, a `Class` in the target model will collect all the operations inherited from the closure of it's super classes that are abstract.

Listing 9.3: Model generator code for the DNF example.

```
1 pre {
2   var id = 1;
3   var exps = nextAddTo(2, (total*0.6).round());
4   var ands = exps.at(0);
5   var ors = exps.at(1);
6 }

8 operation Expr assignID() {
9   self.ID = id.asString();
```

```
10   id += 1;
11 }

13 operation createLiteral() : Literal {
14   var lit = new Literal();
15   lit.assignID();
16   lit.name = encode(id);
17   return lit;
18 }

20 $instances 1
21 operation BooleanExprs create() { }


24 $instances ands
25 operation And create() {
26   self.assignID();
27   var lits = nextInteger(0, 2);
28   if (lits > 0) {
29     for (l in 1.to(lits)) {
30       var lit = createLiteral();
31       lit.parent = self;
32     }
33   }
34 }

36 $instances ors
37 operation Or create() {
38   self.assignID();
39 }

41 $probability 0.3
42 pattern RootExps
43   be : BooleanExprs,
44   exp : Expr
45     from: Expr.all()
46         .select(e | e.isTypeOf(And) or e.isTypeOf(Or))
47         .select(ex | not be.hasExpr.includes(ex))
48   {
49     onmatch {
50       be.hasExpr.add(exp);
51     }
52   }

54 operation Expr isRoot() : Boolean {
55   return BooleanExprs.all().first().hasExpr.includes(self);
56 }

58 operation Expr addChild(child:Expr) {
59     child.parent = self;
60 }
```

```
62 // Ands and Ors at least 2 childs
63 pattern MinExps
64   parent : Expr
65     // All expresions at this point are empty
66     from: Expr.all()
67         .select(e | e.isTypeOf(And) or e.isTypeOf(Or))
68         .select(ex | ex.expr.size() < 2)
69   {
70     onmatch {
71       var childs = Expr.all()
72                 .select(e | e.isTypeOf(And) or e.isTypeOf(Or))
73             .excluding(parent)
74             .excludingAll(parent.closure(e | e.parent))
75             .select(ex | ex.parent.isUndefined() and
76                                     not ex.isRoot());
77       //parent.ID.println("Parent ID: ");
78       //parent.expr.size.println("Initial child: ");
79       if (childs.isEmpty()) {
80         -- No more And/Ors to nest
81         //"Have to add lits ".println();
82         var lits = 2 - parent.expr.size();
83         for (l in 1.to(lits)) {
84           var lit = createLiteral();
85           //lit.ID.println("Child id: ");
86           lit.parent = parent;
87         }
88         var prob = nextDouble();
89         if (prob > 0.8) {
90           //"Not ".println();
91           var neg = new Not();
92           neg.assignID();
93           if (parent.parent.isDefined()) {
94             neg.parent = parent.parent;
95             parent.parent = neg;
96           }
97           else {
98             BooleanExprs.all().first().hasExpr.add(neg);
99             parent.parent = neg;
100          }

102        }
103        return;
104      }
105      else {
106        // No nested Distributions
107        var child = nextFromCollection(childs);
108        if ((child.isTypeOf(Or) and
109                                    parent.isTypeOf(And) and
110                                    (child.orDistInSuccessor() or
111                                            parent.orDistInAncestor()))
                                                or
112                            (child.orDistInSuccessor() and
```

```
113                         (parent.orDistInAncestor() or
114                         parent.orDistInSuccessor()))) {
115           var lit = createLiteral();
116           lit.parent = parent;
117         }
118         else {
119           childs.remove(child);
120           parent.addChild(child);
121         }
122         if (parent.expr.size() == 2) {
123           if (parent.expr.forAll(e | e.isTypeOf(Literal))) {
124             var prob = nextDouble();
125             if (prob > 0.8) {
126               var neg = new Not();
127               neg.assignID();
128               if (parent.parent.isDefined()) {
129                 neg.parent = parent.parent;
130                 parent.parent = neg;
131               }
132               else {
133                 BooleanExprs.all().first().hasExpr.add(neg);
134                 parent.parent = neg;
135               }
136             }
137           }
138           //"Done".println();
139           return;
140         }
141         if (not childs.isEmpty()) {
142           var child = nextFromCollection(childs);
143           // No nested Distributions
144           if ((child.isTypeOf(Or) and
145                                 parent.isTypeOf(And) and
146                                 (parent.orDistInSuccessor() or
147                                         parent.orDistInAncestor()))
                                             or
148               (child.orDistInSuccessor() and
149               (parent.orDistInAncestor() or
150               parent.orDistInSuccessor()))) {
151             var lit = createLiteral();
152             lit.parent = parent;
153             if (parent.expr.forAll(e | e.isTypeOf(Literal))) {
154               var prob = nextDouble();
155               if (prob > 0.8) {
156                 var neg = new Not();
157                 neg.assignID();
158                 if (parent.parent.isDefined()) {
159                   neg.parent = parent.parent;
160                   parent.parent = neg;
161                 }
162                 else {
163                   BooleanExprs.all().first().hasExpr.add(neg);
```

```
164                   parent.parent = neg;
165                 }
166               }
167             }
168             return;
169           }
170           parent.addChild(child);
171         }
172         else {
173           // No more children, fill with lits
174           var lit = createLiteral();
175           lit.parent = parent;
176           if (parent.expr.forAll(e | e.isTypeOf(Literal))) {
177             var prob = nextDouble();
178             if (prob > 0.8) {
179               //"Not ".println();
180               var neg = new Not();
181               neg.assignID();
182               if (parent.parent.isDefined()) {
183                 neg.parent = parent.parent;
184                 parent.parent = neg;
185               }
186               else {
187                 BooleanExprs.all().first().hasExpr.add(neg);
188                 parent.parent = neg;
189               }
190             }
191           }
192         }
193       }
194     }
195   }

197 pattern OrphanExpr
198   orphan : Expr
199     from: Expr.all()
200         .select(e | e.parent.isUndefined())
201   {
202     onmatch {
203       BooleanExprs.all.first().hasExpr.add(orphan);
204     }
205   }

207 /**
208   Find the first And in the hirarchy, if any,
209   that is an OrDistribution
210 */
211 operation Expr orDistInAncestor() : Boolean {
212   if (self.parent.isUndefined()) {
213     return false;
214   }
215   else {
```

```
216     var parentTest = self.parent.isTypeOf(And) and
217         self.parent.expr.exists(e | e.isTypeOf(Or));
218     if (parentTest) {
219       return true;
220     }
221     else {
222       return self.parent.orDistInAncestor();
223     }
224   }
225 }

227 operation Expr orDistInSuccessor() : Boolean {
228   if (self.expr.isEmpty()) {
229     return false;
230   }
231   else {
232     var childTest = self.isTypeOf(And) and
233         self.expr.exists(e | e.isTypeOf(Or));
234     if (childTest) {
235       return true;
236     }
237     else {
238       return self.expr.exists(e | e.orDistInSuccessor());
239     }
240   }
241 }


244 operation Expr isRoot() : Boolean {
245   return BooleanExprs.all().first().hasExpr.includes(self);
246 }

248 post {
249   "GenDone".println();
250 }
```

9.4 Mi to Si

The example addresses the issue of transforming an UML Class hierarchy with multiple inheritance to a Java Class hierarchy with single inheritance. The aim of this transformation is to generate, from an UML (simplified) class hierarchy, a Java (simplified) class hierarchy by defining a multiple-inheritance (MI) interface hierarchy corresponding to the MI UML class hierarchy and to establish implementation links between the class hierarchy and the implementation hierarchy.

Listing 9.4: Model generator code for the Mi2Si example.

```
13 import umlmmmi : 'umlMM.ecore'::umlmmmi;
14 import javammsi : 'javaMM.ecore'::javammsi;
15 import uml2java : 'uml2java.ecore'::umlmi2javasi;

17 transformation Mi2Si {
18     uml imports umlmmmi;
19     java imports javammsi;
20     imports uml2java;
21 }

23 map PackageToPackage in Mi2Si {
24   uml(pIn:Package) {}
25   enforce java () {
26       realize pOut:Package
27   }
28   where () {
29     realize p2p:Package2Package |
30     p2p.umlPackage := pIn;
31     p2p.javaPackage := pOut;
32   }
33 }

35 map ClassInPackage in Mi2Si {
36   check uml(p1:Package, c1:Class |
37         p1.containsClass->includes(c1);){
38     }
39     enforce java (p2:Package) {
40         realize c2:Class, realize i:Interface |
41         c2.implements := Sequence{i};
42         p2.containsClass := p2.containsClass->including(c2);
43         p2.containsIterface := p2.containsIterface->including(i);
44     }
45     where (p2p:Package2Package |
46     p2p.umlPackage = p1;
47     p2p.javaPackage = p2;) {
48         realize c2c:RClass2Class, realize c2i:Class2Interface |
49         c2i.umlClass := c1;
50         c2i.javaInterface := i;
51         c2c.umlClass := c1;
52         c2c.javaClass := c2;
53     }
54     map {
55         where () {
56             c2c.name := c1.name;
57             c2.name := c2c.name;
58             c2i.name := 'I' + c1.name;
59             i.name := c2i.name;
60         }
61     }
62 }
```

```
64 map ClassSuperToImplements in Mi2Si {
65     check uml(umlc1:Class, umlc2:Class |
66         umlc1.supers->includes(umlc2);
67     ) { }
68     enforce java (javac1:Class, javai2:Interface) {
69         javac1.implements := javac1.implements->including(javai2);
70     }
71     where (c2toi:Class2Interface, c12c:Class2Class |
72         c12c.umlClass = umlc1;
73         c12c.javaClass = javac1;
74         c2toi.umlClass = umlc2;
75         c2toi.javaInterface = javai2;
76     ) { }
77 }
```

9.5 Text Path Expression to Path Expression

The *Text Path Expression to Path Expression* is the first step of the PathExp to PetriNet transformation. It is a concrete syntax to abstract syntax transformation for Path Expressions. This transformation addresses the problem of going from a text representation of a Path Expression to an abstract syntax representation that is amenable to graphical representation.

Listing 9.5: Model generator code for the TextualPathExp to PathExp example.

```
1  package PathExp2PetriNet;

3  import java.io.IOException;
4  import java.util.ArrayDeque;
5  import java.util.ArrayList;
6  import java.util.Collections;
7  import java.util.Deque;
8  import java.util.HashMap;
9  import java.util.List;
10 import java.util.ListIterator;
11 import java.util.Map;
12 import java.util.Map.Entry;
13 import java.util.stream.Collectors;

15 import org.apache.commons.math3.random.RandomDataGenerator;
16 import org.eclipse.emf.common.util.EList;
17 import org.eclipse.emf.common.util.URI;
18 import org.eclipse.emf.ecore.EClass;
```

```
19 import org.eclipse.emf.ecore.EFactory;
20 import org.eclipse.emf.ecore.EObject;
21 import org.eclipse.emf.ecore.EPackage;
22 import org.eclipse.emf.ecore.EStructuralFeature;
23 import org.eclipse.emf.ecore.EcorePackage;
24 import org.eclipse.emf.ecore.resource.Resource;
25 import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;
26 import org.eclipse.emf.ecore.xmi.XMLResource;
27 import org.eclipse.emf.ecore.xmi.impl.EcoreResourceFactoryImpl;
28 import org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl;


31 public class TextPathGen {

33   private static final String IN_METAMODEL_PATH =
34     "examples\\uk.ac.york.qvtd.examples.qvtcore\\qvtcsrc" +
35     "\\PathExp2PetriNet\\TextualPathExp.ecore";

37   private static EFactory factory;

39   private static ResourceSetImpl resourceSet;

41     private static EObject ep;

43   /** The generator. */
44   private final RandomDataGenerator generator =
45     new RandomDataGenerator();

47   private char[] chars =
48     "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
49         .toCharArray();

51   private final Integer MAX_ALT_PATHS = 3;

53   private EClass ptEClass;

55   private EClass atEClass;


58   private EClass pEClass;

60   private EClass tpeEClass;

62   private int size;


65   /**
66    * Use a specific seed
67    */
68   public TextPathGen(long seed) {
69     generator.reSeed(seed);
70   }
```

```
73  private EObject createAlternativeTransition(int total) {
74    size++;
75    if (atEClass == null) {
76      atEClass = getEClass("AlternativeTransition");
77    }
78    EObject  t = factory.create(atEClass);
79    // Can have 2-3 paths
80    EStructuralFeature atlPathsSF =
81        atEClass.getEStructuralFeature("altPaths");
82    int maxPaths = total > MAX_ALT_PATHS ? MAX_ALT_PATHS : total;
83    int numPaths = generator.nextInt(2,  maxPaths);
84    for (int i=0; i<numPaths; i++) {
85      Object paths = t.eGet(atlPathsSF);
86      if (paths instanceof EList) {
87        ((EList)paths).add(createPath());
88      }
89    }
90    return t;
91  }


94  private EObject createPath() {
95    if (pEClass == null) {
96      pEClass = getEClass("Path");
97    }
98    size++;
99    EObject p = factory.create(pEClass);
100   return p;
101 }


104 private void createPaths(EObject rootPath, int total) {
105   atEClass = getEClass("AlternativeTrans");
106   pEClass = getEClass("Path");
107   ptEClass = getEClass("PrimitiveTrans");
108   tpeEClass = getEClass("TextualPathExp");
109   EStructuralFeature pathSF =
110     tpeEClass.getEStructuralFeature("path");
111   EStructuralFeature transSF =
112     pEClass.getEStructuralFeature("transitions");
113   EStructuralFeature atlPathsSF =
114     atEClass.getEStructuralFeature("altPaths");

116   Deque<EObject> paths = new ArrayDeque<EObject>();
117   paths.push((EObject) rootPath.eGet(pathSF));
118   EObject currentPath;
119   while(!paths.isEmpty()) {
120     currentPath = (EObject) paths.pop();
121     // Paths always end in a PrimitiveTransition
122     EList trans = (EList) currentPath.eGet(transSF);
```

```
123       if (trans.isEmpty()) {
124         trans.add(createPrimitiveTransition(nextString(20)));
125         total --;
126       }
127       if (total <= 0) {
128         continue;
129       }
130       // The next transition is either Prim or Alternative
131       // For an alternative we would like at least 2 paths,
132       // which means we need at least 2 instances left
133       if (total >= 3) {
134         if (generator.getRandomGenerator().nextBoolean()) {
135           EObject at = createAlternativeTransition(total);
136           EList atPaths = (EList) at.eGet(atlPathsSF);
137           total -= atPaths.size() + 1;
138           trans.add(0, at);
139           trans.add(0, createPrimitiveTransition(nextString(20)));
140           for (Object p : atPaths) {
141             paths.push((EObject) p);
142           }
143           continue;
144         }
145       }
146       // else {
147       // We can only create Primitives
148       trans.add(0, createPrimitiveTransition(nextString(20),
149                                              total>1));
150       total--;
151       paths.push(currentPath);
152     }

154   }

156   private EObject createPrimitiveTransition(String name) {
157     return createPrimitiveTransition(name, false);
158   }

160   private EObject createPrimitiveTransition(String name,
161         boolean multiple) {
162     if (ptEClass == null) {
163       ptEClass = getEClass("PrimitiveTransition");
164     }
165     EObject  t = factory.create(ptEClass);
166     size++;
167     EStructuralFeature nameSF =
168       ptEClass.getEStructuralFeature("name");
169     t.eSet(nameSF, name);
170     if (multiple) {
171       EStructuralFeature multSF =
172         ptEClass.getEStructuralFeature("isMultiple");
173       t.eSet(multSF, generator.getRandomGenerator().nextBoolean());
174     }
```

```
175     return t;
176   }


179   private EObject createTextualPathExp(String name) {
180     if (tpeEClass == null) {
181       tpeEClass = getEClass("TextualPathExp");
182     }
183     EObject exp = factory.create(tpeEClass);
184     size++;
185     EStructuralFeature nameSF =
186       tpeEClass.getEStructuralFeature("name");
187     exp.eSet(nameSF, name);
188     EObject p = createPath();
189     EStructuralFeature pathSF =
190       tpeEClass.getEStructuralFeature("path");
191     exp.eSet(pathSF, p);
192     return exp;
193   }


196   public void generate(int total, String path) {
197     size = 0;
198     // Load the TextualPathExpPackage, so schema location points to
199     //the Ecore
200     resourceSet = new ResourceSetImpl();
201     // Register input metamodel
202     String mallard_root = System.getenv("MALLARD");
203         URI mmUri =
204           URI.createFileURI(mallard_root + IN_METAMODEL_PATH);
205         Resource mm = resourceSet.getResource(mmUri, true);
206         ep = mm.getContents().get(0);
207         if (ep instanceof EPackage) {
208           factory = ((EPackage)ep).getEFactoryInstance();
209         }

211     // Register factory
212     resourceSet.getResourceFactoryRegistry()
213       .getExtensionToFactoryMap()
214       .put("xmi", new XMIResourceFactoryImpl());
215     URI uri = URI.createURI("file:/" + path);
216     Resource model = resourceSet.createResource(uri);

218     // 1. Create the TextualPathExp and randomly assign the size
219     int n = total/10;
220     Map<EObject, Integer> textPathExps;
221     if (n > 1) {
222       textPathExps = nextAddTo(n, total).stream()
223           .collect(Collectors
224             .toMap(i -> createTextualPathExp(nextString(20)),
225                                             i -> i));
226     }
```

```
227     else {
228       textPathExps = new HashMap<>(1);
229       textPathExps.put(createTextualPathExp(nextString(20)), total);
230     }
231     // 2. Create the structure of each TextualPathExp
232     for (Entry<EObject, Integer> entry : textPathExps.entrySet()) {
233         model.getContents().add(entry.getKey());
234         createPaths(entry.getKey(), entry.getValue());
235     }

237     // 3. Save
238     HashMap<Object, Object> savingOptions =
239         new HashMap<Object, Object>();
240     savingOptions.put(XMLResource.OPTION_ENCODING, "UTF-8");
241     savingOptions.put(XMLResource.OPTION_LINE_DELIMITER, "\n");
242     savingOptions
243       .put(XMLResource.OPTION_SCHEMA_LOCATION, Boolean.TRUE);
244     savingOptions
245       .put(XMLResource.OPTION_SCHEMA_LOCATION_IMPLEMENTATION,
246           Boolean.TRUE);
247     savingOptions
248       .put(XMLResource.OPTION_LINE_WIDTH, Integer.valueOf(132));
249     try {
250       model.save(savingOptions);
251     } catch (IOException e) {
252       // TODO Auto-generated catch block
253       e.printStackTrace();
254     }
255     System.out.println("Final size: " + size);
256     resourceSet.getResources().clear();
257     resourceSet = null;
258   }


261   private EClass getEClass(String name) {
262     for (EObject eo : ep.eContents()) {
263       if (eo instanceof EClass) {
264         EClass ec = (EClass)eo;
265         if (ec.getName().equals(name)) {
266           return ec;
267         }
268       }
269     }
270     return null;
271   }


274   private List<Integer> nextAddTo(int n, int m) {
275         assert n > 1;
276         int len = n-1;
277         int[] index = generator.nextPermutation(m, len);
278         List<Integer> values = new ArrayList<>();
```

```
279         for (int i = 0; i < len; i++) {
280             values.add(index[i]);
281         }
282         values.add(0, 0);
283         values.add(m);
284         Collections.sort(values);
285         List<Integer> result = new ArrayList<>();
286         ListIterator<Integer> it = values.listIterator(1);
287         while (it.hasNext()) {
288             int low = it.previous();
289             it.next();
290             int high = it.next();
291             result.add(high-low);
292         }
293         return result;
294     }
```

```
299   private String nextString(int length) {
300         StringBuilder sb = new StringBuilder();

302         for (int i = 0; i < length; i++) {
303             sb.append(chars[generator.nextInt(0, chars.length-1)]);
304         }
305         return sb.toString();
306     }

308   /**
309    *
310    */
311   public static void registerEPackages() {
312     // Register Ecore
313     EcorePackage ecore = EcorePackage.eINSTANCE;
314     Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap()
315       .put("ecore", new EcoreResourceFactoryImpl());
316     Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap()
317       .put("xmi", new XMIResourceFactoryImpl());
318   }

320 }
```

9.6 Path Expression to Petri Net

The Path Expression to Petri Net example describes a transformation from a path expression to a Petri net. This Annex provides the complete transformation code of the whole transformation se-

quence that enables to produce an XML Petri net representation (in the PNML format) from a textual definition of a path expression.

Listing 9.6: Complete PathExpression to PetriNet example.

```
pre {
        var id = 0;
        var num_exp = total/10;
        // Two random numbers 0..total
        var quant = nextAddTo(2, total).sortBy(q | q);
        var num_s = 1 + quant.at(0);  // Min 2 States
        var num_t = quant.at(1);
}


$instances num_exp
@list paths
operation PathExp create() {
        self.name = nextCamelCaseString(15, 10);
}

$instances num_s
@list states
operation State create() {
        //nextFromList("paths").states.add(self);
}

$instances num_t
operation Transition create() {
        self.name = id.asString();
        id += 1;
        nextFromList("paths").transitions.add(self);
}


pattern Transition
        exp:PathExp,
        tra:Transition
          from: exp.transitions
        {
                onmatch {
                        var size = 0;
                        var freeSources = State.all().select(s |
                            s.incoming.size() == size);
                        while (freeSources.isEmpty()) {
                                size += 1;
                                freeSources = State.all().select(s |
                                    s.incoming.size() == size);
                        }
                        size = 0;
```

```
44                          var freeTarget = State.all().select(s |
                                s.outgoing.size() == size);
45                          while (freeTarget.isEmpty()) {
46                                  size += 1;
47                                  freeTarget = State.all().select(s |
                                        s.outgoing.size() == size);
48                          }
49                          var source = nextFromCollection(freeSources);
50                          var target = nextFromCollection(freeTarget);
51                          tra.source = source;
52                          tra.target = target;
53                          exp.states.add(source);
54                          exp.states.add(target);
55                  }
56          }


59  post {
60          "GenDone".println();
61  }
```

9.7 Petri Net to PNML(XML)

The Petri Net to PNML(XML) example describes a transformation from the PetriNet domain to the XML domain. The XML model will provide an XML representation of the PetriNet in PNML format[1]. The transition is, as with the PathExp to PetriNet (Sect. 8.12), straight forward, with one mapping to transform elements of each of the PetriNet classes into it's XML representation. Note that the used XML metamodel is a simplified version of the XML specification.

Listing 9.7: Model generator code for the Petri Net to PNML(XML) example.

```
1  pre {
2          var id = 0;
3          var num_nets = 1;
4          // Two random numbers 0..total
5          var quant = nextAddTo(2, total).sortBy(q | q);
6          var num_p = 1 + quant.at(0);  // Min 2 Places
7          // For each transition we need two arcs
8          var num_t = (quant.at(1)/3).ceiling();
9  }
```

[1] http://www.pnml.org. last accessed 10/05/2017

```
$instances num_nets
@list net
operation PetriNet create() {
        self.name = nextCamelCaseString(15, 10);
}

$instances num_p
@list places
operation Place create() {
        self.name = "P_" + nextString("lower", 15);
        //nextFromList("net").places.add(self);
}

$instances num_t
operation Transition create() {
        self.name = "T_" + nextString("lower", 15);
        //nextFromCollection(PetriNet.all().select(pn |
            pn.places.size() >= 2)).transitions.add(self);
        nextFromList("net").transitions.add(self);
}


pattern Transition
        net:PetriNet,
        tra:Transition
          from: net.transitions
        {
                onmatch {
                        var size = 0;
                        var freeSources = Place.all().select(s |
                            s.incoming.size() == size);
                        while (freeSources.isEmpty()) {
                                size += 1;
                                freeSources = Place.all().select(s |
                                    s.incoming.size() == size);
                        }
                        size = 0;
                        var freeTarget = Place.all().select(s |
                            s.outgoing.size() == size);
                        while (freeTarget.isEmpty()) {
                                size += 1;
                                freeTarget = Place.all().select(s |
                                    s.outgoing.size() == size);
                        }
                        var source = nextFromCollection(freeSources);
                        var target = nextFromCollection(freeTarget);
                        var a1:Arc = new PlaceToTransArc();
                        a1.weight = nextInteger(10);
                        a1.source = source;
                        net.places.add(source);
                        a1.target = tra;
```

```
57                          net.arcs.add(a1);
58                          var a2:Arc = new TransToPlaceArc();
59                          a1.weight = nextInteger(10);
60                          a2.source = tra;
61                          a2.target = target;
62                          net.places.add(target);
63                          net.arcs.add(a2);
64                  }
65          }


68 post {
69          "GenDone".println();
70 }
```

9.8 Railway to Control

The example is based on the train benchmark metamodel [102], which the authors claim contains the most typical class diagram constructs. The transformation considers that the train domain can be used model a toy model train, and we are interested in automate the signals (semaphores) and switches that control the toy model train behaviour. The transformation is used to create a model of a control system that can be used to monitor the state of the toy model train and, for example, automate the toy railway to allow multiple trains to run on it.

Listing 9.8: Model generator code for the Railway to Control example.

```
1 pre {

3         var cont:Integer = total/10;
4         var seg = 0;
5         var sw = 0;
6         var r = 0;
7         var sem = 0;
8         var pos = 0;
9         var sen = 0;     // At least 2
10        var reg = 0;

12        while (total > 0) {
13                reg += 1;
14                total -= 1;
15                r += 1;
16                total -= 1;
17                sw += 1;
```

```
18                  total -= 1;
19                  seg += 2;
20                  total -= 2;
21                  pos += 1;
22                  total -= 1;
23                  sen += 2;
24                  total -= 2;
25                  sem += 2;
26                  total -= 2;
27          }
28          var id = 0;
29  }


32  $instances sem
33  @list semaphores
34  operation Semaphore create()
35  {
36          self.id = id;
37          id += 1;
38  }

40  $instances r
41  @list routes
42  operation Route create()
43  {
44          self.id = id;
45          id += 1;
46          self.entry = nextFromListAsSample('semaphores');
47          self.exit = nextFromListAsSample('semaphores');
48  }

50  $instances sw
51  @list switches
52  operation Switch create()
53  {
54          self.id = id;
55          id += 1;

57  }

59  $instances pos
60  operation SwitchPosition create()
61  {
62          self.id = id;
63          id += 1;
64          self.target = nextFromListAsSample('switches');
65          self.route = nextFromList('routes');

67          switch(nextInteger(2)) {
68                  case 0: self.position = Position#FAILURE;
69                  case 1: self.position = Position#STRAIGHT;
```

```
70                  case 2: self.position = Position#DIVERGING;
71          }
72  }


75  $instances seg
76  operation Segment create()
77  {
78          self.id = id;
79          id += 1;
80          self.length = nextInteger(20)+1;
81          var items : Real = Semaphore.all().size().asReal();
82          var elems = items.log10(null);
83          elems = 5.pow(elems).min(items).round();
84          elems = nextInteger(elems-1)+1;
85          self.semaphores.addAll(nextSample(Semaphore.all(), elems));
86          items = TrackElement.all().size().asReal();
87          elems = items.log10(null);
88          elems = 5.pow(elems).min(items).round();
89          elems = nextInteger(elems-1)+1;
90          if (elems < 2) {
91                  elems = 2;
92          }
93          self.neighbors.addAll(nextSample(TrackElement.all(), elems));
94          var left = nextFromList('switches');
95          if (nextBoolean()) {
96                  left.left = self;
97          }
98          else {
99                  if (left.left.isDefined()) {
100                         self.connectsTo.add(left.left);
101                 }
102         }
103         var right = nextFromList('switches');
104         if (nextBoolean()) {

106                 right.right = self;
107         }
108         else {
109                 if (right.right.isDefined()) {
110                         self.connectsTo.add(right.right);
111                 }
112         }
113         var fr = nextFromList('switches');
114         if (nextBoolean()) {

116                 fr.`from` = self;
117         }
118         else {
119                 if (fr.`from`.isDefined()) {
120                         self.connectsTo.add(fr.`from`);
121                 }
```

```
122         }
123 }


126 $instances reg
127 @list regions
128 operation Region create()
129 {
130         self.id = id;
131         id += 1;
132         var items = TrackElement.all().size().asReal();
133         var elems = items.log10(null);
134         elems = 5.pow(elems).min(items).round();
135         elems = nextInteger(elems-1)+1;
136         self.elements.addAll(nextSample(TrackElement.all(), elems));
137 }


140 $instances sen
141 operation Sensor create()
142 {
143         self.id = id;
144         id += 1;
145         var region = nextFromList('regions');
146         region.sensors.add(self);
147         if (region.elements.size() > 0) {
148                 var items = region.elements.size().asReal();
149                 var elems = items.log10(null);
150                 elems = 5.pow(elems).min(items).round();
151                 elems = nextInteger(elems-1)+1;
152                 var monitors = nextSample(region.elements, elems);
153                 self.monitors.addAll(monitors);
154                 var routes = monitors.select(m | m.isTypeOf(Switch))
155                                 .collect(sw |
                                        sw.positions.collect(swp |
                                        swp.route))
156                                 .flatten();
157                 for (r in routes) {
158                         r.gathers.add(self);
159                 }

161         }
162 }


165 operation RailwayContainer create()
166 {
167         self.routes.addAll(Route.all());
168         self.regions.addAll(Region.all());
169 }
```

```
172 post {
173         "GenDone".println();
174 }
```

Bibliography

1. LINA & INRIA ATLAS group: Specification of the ATL Virtual Machine. http://www.eclipse.org/atl/documentation/old/ATL_VMSpecification[v00.01].pdf. (2005).
2. Object Management Group (OMG): MDA Guide Version 1.0.1. OMG Document Number: omg /2003-06-01. http://www.omg.org/cgi-bin/doc?omg/03-06-01. (2003).
3. Object Management Group (OMG): Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. OMG Document Number: formal /2015-02-01. http://www.omg.org/spec/QVT/1.2/. (2015).
4. Object Management Group (OMG): OMG Meta Object Facility (MOF) Core Specification. OMG Document Number: formal /2015-06-05. http://www.omg.org/spec/MOF/2.5/. (2015).
5. Object Management Group (OMG): OMG Unified Modeling Language (OMG UML), superstrucutre. OMG Document Number: formal /2011-08-06. http://www.omg.org/spec/UML/. (2011).
6. Object Management Group (OMG): Request for Proposals: MOF 2.0 Query/Views/Transformations RFP. OMG Document Number: ad/ 2002-04-10. http://www.omg.org/cgi-bin/doc?ad/2002-4-10. (2002).
7. Abdel-Kader, Rehab F.: Particle Swarm Optimization for Constrained Instruction Scheduling. VLSI Design 2008(4), 7:1–7:7 (2008)
8. Agrawal, Aditya, Karsai, Gabor, Neema, Sandeep, Shi, Feng, Vizhanyo, Attila: The design of a language for model transformations. Software & Systems Modeling 5(3), 261–288 (2006)
9. Aho, A. V. Johnson, S. C. Ullman, J. D. Code Generation for Expressions with Common Subexpressions. J. ACM 24(1), 146–160 (1977)
10. Aho, Alfred V. Lam, Monica S. Sethi, Ravi, Ullman, Jeffrey D. Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)

11. Armstrong, R.A. Slade, S.V. Eperjesi, F. An introduction to analysis of variance (ANOVA) with special reference to data from clinical experiments in optometry. Ophthalmic and Physiological Optics 20(3), 235–241 (2000)

12. Battacharyya, Shuvra S. Murthy, Praveen K. A., Lee Edward: Software Synthesis from Data Flow Graphs. Kluwer Academic Publishers (1996)

13. Bernstein, D. Rodeh, M. Gertner, I. On the complexity of scheduling problems for parallel/pipelined machines. IEEE Transactions on Computers 38(9), 1308–1313 (1989)

14. Bézivin, Jean: In search of a basic principle for model driven engineering. UPGRADE 5(2), 21–24 (2004)

15. Bézivin, Jean: On the unification power of models. Software & Systems Modeling 4(2), 171–188 (2005)

16. Bézivin, Jean, Dupé, Grégoire, Jouault, Frédéric, Pitette, Gilles, Rougui, Jamal Eddine: First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In: 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture (2003)

17. Bézivin, Jean, Gerbé, Olivier: Towards a precise definition of the OMG/MDA framework. In: Proceedings of the 16$^{\text{th}}$ Annual International Conference on Automated Software Engineering (ASE). Pp. 273–280 (2001)

18. Bilardi, Gianfranco, Pingali, Keshav: Algorithms for computing the static single assignment form. Journal of the ACM (JACM) 50(3), 375–425 (2003)

19. Blum, Christian, Calvo, Borja: A matheuristic for the minimum weight rooted arborescence problem. Journal of Heuristics 21(4), 479–499 (2015)

20. Blum, Christian, Roli, Andrea: Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. ACM Comput. Surv. 35(3), 268–308 (2003)

21. Steven Bosems: A Performance Analysis of Model Transformations and Tools. MA thesis: Department of Electrical Engineering Mathematics and Computer Science, University of Twente. (2011).

22. Cabot, Jordi, Clarisó, Robert, Guerra, Esther, de Lara, Juan: Verification and validation of declarative model-to-model transformations through invariants. Journal of Systems and Software 83(2), 283–302 (2010)

23. Callow, Glenn, Kalawsky, Roy: A Satisficing Bi-Directional Model Transformation Engine using Mixed Integer Linear Programming. Journal of Object Technology 12(1) (2013)

24. Cayley, A. A Theorem on Trees. Quart. J. Math 23, 376–378 (1889)

25. Chaudhuri, Surajit: An Overview of Query Optimization in Relational Systems. In: Proceedings of the 17$^{\text{th}}$ ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), pp. 34–43. ACM, Seattle, Washington, USA (1998)

26. Christoph, Alexander: Graph Rewrite Systems for Software Design Transformations. In: Objects, Components, Architectures, Services, and Applications for a Networked World. Ed. by M. Aksit, M. Mezini, and R. Unland, pp. 76–86. Springer Berlin Heidelberg(2003)

27. Crespo, Yania, Marques, Jos Manuel, Rodríguez, Juan José: On the translation of multiple inheritance hierarchies into single inheritance hierarchies. In: Proceedings of the Inheritance Workshop at ECOOP, pp. 30–37 (2002)

28. Czarnecki, K. Helsen, S. Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–645 (2006)

29. Dao, Michel, Huchard, Marianne, Libourel Rouge, Thèrése, Pons, Anne, Villerd, Jean: Proposals for Multiple to Single Inheritance Transformation. In: Lahire, Ph. G., Arevalo, H., Astudillo, A.P., Black, E., Ernst, M., Huchard, M., Sakkinen, P., Valtchev (eds.) Proceecings of the 3$^{\text{rd}}$ Workshop on MechAnisms for SPEcialization, Generalization and inHerItance (MASPEGHI), pp. 21–26, Oslo, Norway (2004)

30. Di Ruscio, Davide, Eramo, Romina, Pierantonio, Alfonso: Model Transformations. In: Formal Methods for Model-Driven Engineering, pp. 91–136. Springer Berlin / Heidelberg(2012)

31. Dickerson, C.E. Mavris, D. A Brief History of Models and Model Based Systems Engineering and the Case for Relational Orientation. Systems Journal, IEEE 7(4), 581–592 (2013)

32. Dorigo, M. Gambardella, L. M. Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Transactions on Evolutionary Computation 1(1), 53–66 (1997)

33. Dorigo, Marco, Di Caro, Gianni: New Ideas in Optimization. In: ed. by D. Corne, M. Dorigo, F. Glover, D. Dasgupta, P. Moscato, R. Poli, and K.V. Price, pp. 11–32. McGraw-Hill Ltd., UK, Maidenhead, UK, England(1999)

34. Dorigo, Marco, Gambardella, Luca Maria: "Proceedings of the 4th International Conference on Parallel Problem Solving from Nature (PPSN), Berlin, Germany, September 22–26". In: ed. by H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996. Chap. A study of some properties of Ant-Q, pp. 656–665. ISBN: 978-3-540-70668-7.

35. Dorigo, Marco, Stützle, Thomas: "The Ant Colony Optimization Metaheuristic: Algorithms, Applications, and Advances". In: Handbook of Metaheuristics. Ed. by F. Glover and G.A. Kochenberger. Boston, MA: Springer US, 2003, pp. 250–285. ISBN: 978-0-306-48056-0.

36. Duddy, K. Gerber, A. Lawley, M. Raymond, K. Steel, J. Model transformation: a declarative, reusable patterns approach. In: Proceedings of the 7th Conference on International Enterprise Distributed Object Computing Conference. Pp. 174–185 (2003)

37. Eberhart, R. Kennedy, J. A new optimizer using particle swarm theory. In: Proceedings of the Sixth International Symposium on Micro Machine and Human Science (MHS). Pp. 39–43 (1995)

38. Ehrig, Hartmut, Ehrig, Karsten, de Lara, Juan, Taentzer, Gabriele, Varró, Dániel, Varró-Gyapay, Szilvia: Termination Criteria for Model Transformation. In: Fundamental Approaches to Software Engineering. Ed. by M. Cerioli, pp. 49–63. Springer Berlin Heidelberg(2005)

39. Ferrante, Jeanne, Ottenstein, Karl J. Warren, Joe D. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9(3), 319–349 (1987)

40. Fouquet, Francois, Barais, Olivier, Jézéquel, Jean-Marc: Building a Kermeta Compiler using Scala: an Experience Report. In: Proc. Scala Days (2010)

41. Gajski, Daniel D, Abdi, Samar, Gerstlauer, Andreas, Schirner, Gunar: Embedded System Design: Modeling, Synthesis and Verification. Springer Publishing Company, Incorporated (2009)

42. Gardner, T. Griffin, C. Koehler, J. Hauser, R. A review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations towards the final Standard. In: Proceedings of the Workshop on Metamodelling for MDA (2003)

43. Greenyer, Joel, Kindler, Ekkart: Comparing relational model transformation technologies: implementing Query/View/Transformation with Triple Graph Grammars. Software & Systems Modeling 9(1), 21–46 (2010)

44. Grune, Dick, Reeuwijk, Kees van, Bal, Henri E. Jacobs, Ceriel J.H. Langendoen, Koen: Modern Compiler Design. Springer Publishing Company, Incorporated (2012)

45. Halbwachs, Nicolas, Raymond, Pascal, Ratel, Christophe: Generating efficient code from data-flow programs. In: Programming Language Implementation and Logic Programming, pp. 207–218 (1991)

46. Hamzheei, Mahdi, Farahani, Reza Zanjirani, Rashidi-Bajgan, Hannaneh: An ant colony-based algorithm for finding the shortest bidirectional path for automated guided vehicles in a block layout. The International Journal of Advanced Manufacturing Technology 64(1), 399–409 (2013)

47. Heffernan, Mark, Wilken, Kent: Data-Dependency Graph Transformations for Instruction Scheduling. Journal of Scheduling 8(5), 427–451 (2005)

48. Hennessy, John L. Gross, Thomas: Postpass Code Optimization of Pipeline Constraints. ACM Trans. Program. Lang. Syst. 5(3), 422–448 (1983)

49. Hernández, Javier Martin, Mieghem, Piet Van: Classification of Graph Metrics. Report, Last accesed 11/05/2017. Delft University of Technology (2011)

50. Hildebrandt, Stephan, Lambers, Leen, Giese, Holger: "Complete Specification Coverage in Automatically Generated Conformance Test Cases for TGG Implementations". In: Proceedings of the 6$^{\text{th}}$ International Conference on Theory and Practice of Model Transformations (ICMT), Budapest, Hungary, June 18-19, 2013. Ed. by K. Duddy and G. Kappel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 174–188. ISBN: 978-3-642-38883-5.

51. Horn, Tassilo: "Model Querying with FunnyQT". In: Proceedings of the 6th International Conference on Theory and Practice of Model Transformations (ICMT), Budapest, Hungary, June 18-19. Ed. by K. Duddy and G. Kappel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 56–57.

52. Hou, E. S. H. Ansari, N. Ren, Hong: A genetic algorithm for multiprocessor scheduling. IEEE Transactions on Parallel and Distributed Systems 5(2), 113–120 (1994)

53. Hoyos Rodríguez, Horacio, Kolovos, Dimitrios: Declarative Model Transformation Execution Planning. In: Brucker, Achim D. Cabot, Jordi, Herrera, Adolfo Sánchez-Barbudo (eds.) Proceedings of the 16th International Workshop on OCL and Textual Modeling (OCL), Satellite event of MODELS. (2016)

54. Insfran, Emilio, Gonzalez-Huerta, Javier, Abrahão, Silvia: Design Guidelines for the Development of Quality-Driven Model Transformations. In: Petriu, Dorina C. Rouquette, Nicolas, Haugen, Øystein (eds.) Model Driven Engineering Languages and Systems, pp. 288–302. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)

55. Jarke, Matthias, Koch, Jurgen: Query Optimization in Database Systems. ACM Comput. Surv. 16(2), 111–152 (1984)

56. Jouault, Frédéric, Allilaire, Freddy, Bézivin, Jean, Kurtev, Ivan: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)

57. Jouault, Frédéric, Kurtev, Ivan: On the architectural alignment of ATL and QVT. In: Proceedings of the 2006 ACM symposium on Applied computing. SAC '06, pp. 1188–1195. ACM (2006)

58. Jouault, Frédéric, Kurtev, Ivan: Transforming Models with ATL. In: Satellite Events at the MoDELS 2005 Conference. Ed. by J.-M. Bruel, pp. 128–138. Springer Berlin Heidelberg(2006)

59. Karaboga, Dervis, Basturk, Bahriye: A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm. Journal of Global Optimization 39(3), 459–471 (2007)

60. Kent, Stuart: Model Driven Engineering. In: Integrated Formal Methods. Ed. by M. Butler, L. Petre, and K. Sere, pp. 286–298. Springer Berlin Heidelberg(2002)

61. Kessentini, Marouane, Sahraoui, Houari, Boukadoum, Mounir: Model Transformation as an Optimization Problem. In: Model Driven Engineering Languages and Systems, pp. 159–173. Springer Berlin / Heidelberg(2008)

62. Kiegeland, Jörg, Eichler, Hajo: Enabling comprehensive tool support for QVT. Eclipse Summit Europe (2007)

63. Kolovos, Dimitrios S. Paige, Richard F. Polack, Fiona A. C. The Epsilon Transformation Language. In: Proceedings of the First International Conference on Theory and Practice of Model Transformations (ICMT), Zürich, Switzerland, July 1-2. Ed. by A. Vallecillo, J. Gray, and A. Pierantonio, pp. 46–60. Springer Berlin Heidelberg, Berlin, Heidelberg(2008)

64. Korson, Tim, McGregor, John D. Understanding Object-oriented: A Unifying Paradigm. Commun. ACM 33(9), 40–60 (1990)

65. Kowalski, Robert: Algorithm = Logic + Control. Commun. ACM 22(7), 424–436 (1979)

66. Krishnamurthy, Ravi, Zaniolo, Carlo: "Optimization in a logic based language for knowledge and data intensive applications". In: Proceedings of the International Conference on Extending Database Technology (EDBT), Venice, Italy, March 14–18. Ed. by J.W. Schmidt, S. Ceri, and M. Missikoff. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 16–33. ISBN: 978-3-540-39095-4.

67. Kumar, Vipin, Lin, Yow-Jian: A data-dependency-based intelligent backtracking scheme for PROLOG. The Journal of Logic Programming 5(2), 165–181 (1988)

68. Kurtev, Ivan: State of the Art of QVT: A Model Transformation Language Standard. In: Applications of Graph Transformations with Industrial Relevance, pp. 377–393. Springer Berlin / Heidelberg(2008)

69. von Laszewski, Gregor: "Intelligent Structural Operators for the k-way Graph Partitioning Problem". http://surface.syr.edu/npac/30. 1991.

70. Lauder, Marius, Anjorin, Anthony, Varró, Gergely, Schürr, Andy: Bidirectional Model Transformation with Precedence Triple Graph Grammars. In: Modelling Foundations and Applications. Ed. by A. Vallecillo, J.-P. Tolvanen, E. Kindler, H. Störrle, and D. Kolovos, pp. 287–302. Springer Berlin Heidelberg(2012)

71. Mateo, Sergi, Blum, Christian, Fua, Pascal, Türetgen, Engin: "Hybrid Algorithms for the Minimum-Weight Rooted Arborescence Problem". In: Proceedings of the 8th International Conference on Swarm Intelligence (ANTS), Brussels, Belgium, September 12-14. Ed. by M. Dorigo, M. Birattari, C. Blum, A.L. Christensen, A.P. Engelbrecht, R. Groß, and T. Stützle. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 61–72. ISBN: 978-3-642-32650-9.

72. Mens, Tom, Van Gorp, Pieter: A Taxonomy of Model Transformation. Electronic Notes in Theoretical Computer Science 152, 125–142 (2006)

73. Mernik, Marjan, Heering, Jan, Sloane, Anthony M. When and how to develop domain-specific languages. ACM Comput. Surv. 37(4), 316–344 (2005)

74. Mottu, Jean-Marie, Baudry, Benoit, Traon, Yves Le: Model Transformation Testing: Oracle Issue. In: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW), pp. 105–112. IEEE Computer Society, Washington, DC, USA (2008)

75. Muchnick, Steven S. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1997)

76. Muller, P.A. Fleurey, F. Vojtisek, D. Drey, Z. Pollet, D. Fondement, F. Studer, P. Jézéquel, J.M. On executable meta-languages applied to model transformations. In: Model Transformations In Practice Workshop (2005)

77. Muller, P.A. Fleurey, Franck, Jézéquel, J.M. Weaving Executability into Object-Oriented Meta-languages. In: Model Driven Engineering Languages and Systems. Ed. by L. Briand and C. Williams, pp. 264–278. Springer Berlin Heidelberg(2005)

78. Neumann, Frank, Witt, Carsten: Ant Colony Optimization and the minimum spanning tree problem. Theoretical Computer Science 411(25), 2406–2413 (2010)

79. Otter, Richard: The number of trees. Annals of Mathematics (1948)

80. Paige, R.F. Kolovos, D.S. Rose, L.M. Drivalos, N. Polack, F.A.C. The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In: Proceedings of the 14$^{\text{th}}$ IEEE International Conference on Engineering of Complex Computer Systems, pp. 162–171 (2009)

81. Pierce, Benjamin C. Types and Programming Languages. The MIT Press (2002)

82. Pingali, Keshav, Beck, Micah, Johnson, Richard, Moudgill, Mayan, Stodghill, Paul: Dependence flow graphs: an algebraic approach to program dependencies. In: Proceedings of the 18$^{\text{th}}$ ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), pp. 67–78. ACM, Orlando, Florida, USA (1991)

83. Popoola, Saheed, Kolovos, Dimitrios S. Rodriguez, Horacio Hoyos: "EMG: A Domain-Specific Transformation Language for Synthetic Model Generation". In: Proceedings of the 9$^{\text{th}}$ International Conference on the Theory and Practice of Model Transformations (ICMT), Held as Part of STAF 2016, Vienna, Austria, July 4-5. Ed. by P. Van Gorp and G. Engels. Cham: Springer International Publishing, 2016, pp. 36–51. ISBN: 978-3-319-42064-6.

84. Previtali, Luca, Lurati, Brenno, Wilde, Erik: BibTEXML: An XML Representation of Bibtex. In: Poster Proceedings of the 10th International World Wide Web Conference, pp. 64–65. ACM Press, Hong Kong (2001)

85. Queralt, Pascual, Hoyos, Luis, Boronat, Artur, Carsí, José Á, Ramos, Isidro: Un Motor de Transformación de Modelos con Soporte para el Lenguaje QVT Relations. In: Proceedings of Desarrollo de Software Dirigido por Modelos (DSDM), junto a JISBD (2006)

86. Rao, V. Venkata, Sridharan, R. Minimum-weight rooted not-necessarily-spanning arborescence problem. Networks 39(2), 77–87 (2002)

87. Schmidt, Douglas C: Model-driven engineering. Computer 39(2), 25 (2006)

88. Schürr, Andy, Klar, Felix: 15 Years of Triple Graph Grammars. In: Ehrig, Hartmut, Heckel, Reiko, Rozenberg, Grzegorz, Taentzer, Gabriele (eds.) Proceedings of the 4th International Conference on Graph Transformations (ICGT), Leicester, United Kingdom, September 7-13, pp. 411–425. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)

89. Selic, Bran: What will it take? A view on adoption of model-based methods in practice. Software & Systems Modeling 11(4), 513–526 (2012)

90. Selinger, P. Griffiths, Astrahan, M. M. Chamberlin, D. D. Lorie, R. A. Price, T. G. Access Path Selection in a Relational Database Management System. In: Proceedings of the 1979 ACM International Conference on Management of Data (SIGMOD), pp. 23–34. ACM, Boston, Massachusetts (1979)

91. Shadish, William R.. Cook, Thomas D, Campbell, Donald Thomas: Experimental and quasi-experimental designs for generalized causal inference. Wadsworth Cengage learning (2002)

92. Smotherman, Mark, Krishnamurthy, Sanjay, Aravind, P. S. Hunnicutt, David: Efficient DAG Construction and Heuristic Calculation for Instruction Scheduling. In: Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO), pp. 93–102. ACM, Albuquerque, New Mexico, Puerto Rico (1991)

93. Spinellis, Diomidis: Notable design patterns for domain-specific languages. Journal of Systems and Software 56(1), 91–99 (2001)

94. Steel, J. Lawley, M. Model-based test driven development of the Tefkat model-transformation engine. In: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE), pp. 151–160 (2004)

95. Steinberg, David, Budinsky, Frank, Paternostro, Marcelo, Merks, Ed: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley Professional (2009)

96. Stevens, Perdita: A Simple Game-Theoretic Approach to Checkonly QVT Relations. In: Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT), pp. 165–180. Springer-Verlag, Zurich, Switzerland (2009)

97. Stevens, Perdita: Bidirectional model transformations in QVT: semantic issues and open questions. Software & Systems Modeling 9, 7–20 (2010)

98. Stützle, Thomas, Hoos, Holger H. MAX–MIN Ant System. Future Generation Computer Systems 16(8), 889–914 (2000)

99. Stützle, Thomas, López-Ibáñez, Manuel, Pellegrini, Paola, Maur, Michael, de Oca, Marco Montes, Birattari, Mauro, Dorigo, Marco: Parameter Adaptation in Ant Colony Optimization. Tech. rep. TR/-IRIDIA/2010-002, Bruxelles, Belgium: Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle (IRIDIA) (2010)

100. Sweany, P.H. Beaty, S.J. Dominator-path Scheduling - A Global Scheduling Method. In: Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO), 1992. Pp. 260–263 (1992)

101. Taentzer, G. Ehrig, K. Guerra, E. De Lara, J. Lengyel, L. Levendovszky, T. Prange, U. Varró, D. Varró-Gyapay, S. Model transformation by graph transformation: A comparative study. In: Proceddings of the Workshop on Model Transformation in Practice (2005)

102. Ujhelyi, Zoltán, Bergmann, Gábor, Hegedüs, Ábel, Horváth, Ákos, Izsó, Benedek, Ráth, István, Szatmári, Zoltán, Varró, Dániel: EMF-IncQuery: An integrated development environment for live model queries. Sci. Comput. Program. 98(P1), 80–99 (2015)

103. Varró, Dániel, Pataricza, András: Generic and Meta-transformations for Model Transformation Engineering. In: UML 2004 - The Unified Modeling Language. Modelling Languages and Applications. Ed. by T. Baar, A. Strohmeier, A. Moreira, and S. Mellor, pp. 290–304. Springer Berlin Heidelberg(2004)

104. Varró, Gergely, Deckwerth, Frederik, Wieber, Martin, Schürr, Andy: An algorithm for generating model-sensitive search plans for pattern matching on EMF models. Software & Systems Modeling 14(2), 597–621 (2015)

105. Vazirani, Vijay V. Approximation Algorithms. Springer Berlin Heidelberg (2003)

106. Wagelaar, Dennis, Tisi, Massimo, Cabot, Jordi, Jouault, Frédéric: Towards a general composition semantics for rule-based model transformation. In: Proceedings of the 14th international conference on Model driven engineering languages and systems (MoDELS), pp. 623–637. Springer-Verlag (2011)

107. Walsh, Norm, Muellner, Leonard: DocBook: The Definitive Guide with CD-ROM. O'Reilly & Associates, Inc., Sebastopol, CA, USA (1999)

108. Wang, Gang, Gong, Wenrui, Derenzi, Brian, Kastner, Ryan: Exploring Time/Resource Trade-offs by Solving Dual Scheduling Problems with the Ant Colony Optimization. ACM Trans. Des. Autom. Electron. Syst. 12(4) (2007)

109. Wang, Gang, Gong, Wenrui, Kastner, Ryan: Instruction Scheduling Using MAX-MIN Ant System Optimization. In: Proceedings of the 15th ACM Great Lakes Symposium on VLSI (GLSVLSI), pp. 44–49. ACM, Chicago, Illinois, USA (2005)

110. Wang, JinFeng, Wu, Xuehua, Fan, Xiaoliang: A two-stage ant colony optimization approach based on a directed graph for process planning. The International Journal of Advanced Manufacturing Technology (2015)

111. Wilhelm, Reinhard, Seidl, Helmut: Compiler Design, Virtual Machines. Springer Heidelberg (2010)

112. Willink, Edward: An extensible OCL virtual machine and code generator. In: Proceedings of the 12th Workshop on OCL and Textual Modelling (OCL), pp. 13–18. ACM (2012)

113. Willink, Edward D. "Optimized declarative transformation First Eclipse QVTc results". In: Proceedings of the 3rd Workshop on Scalable Model Driven Engineering (BigMDE), Vienna, Austria, July 9. Ed. by D. Kolovos, D. Di Rusco, N. Matragkas, J. Sánchez Cuadrado, I. Rath, and M. Tisi. CEUR-WS, 2016.

114. Willink, Edward, Hoyos, Horacio, Kolovos, Dimitris: Yet Another Three QVT Languages. In: Theory and Practice of Model Transformations. Ed. by K. Duddy and G. Kappel, pp. 58–59. Springer Berlin Heidelberg(2013)

115. Wimmer, M. Kappel, G. Schoenboeck, J. Kusel, A. Retschitzegger, W. Schwinger, W. A Petri Net Based Debugging Environment for QVT Relations. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 3–14 (2009)

116. Wolfe, Michael, Banerjee, Utpal: Data dependence and its application to parallel processing. International Journal of Parallel Programming 16(2), 137–178 (1987)