A Multi-Paradigm Foundation for Model Transformation Language Engineering

Eugene Syriani

Supervisor: Professor Hans Vangheluwe

School of Computer Science
McGill University
Montreal, Quebec, Canada

February 4th, 2011

A thesis submitted to McGill University in partial
fulfilment of the requirements of the degree of
Doctor of Philosophy in Computer Science

Acknowledgements

First of all, I wish to thank my parents. They have been a blessing for me with their moral and financial support during my scholar life. Thank you for giving me a place where I could live and work at the same time and for all the sacrifices you have made for me. A special thanks to my mother who continuously kept on motivating me. Thank you to my brother and my two sisters for being understanding and patient with me during the dense years of my thesis.

I would like to thank my supervisor, Hans Vangheluwe, for being a mentor for my research and giving me the opportunity to freely be creative in my thoughts, while leading me on the right path. I thank him for introducing me to the modelling community and for supporting me financially, especially for allowing me to travel and attend many international conferences. I thank him for all the effort, devotion, and personal time he has spent on my work through all these years.

I thank all the members of our MSDL labs both in Montreal and in Antwerp. Thank you to the AnSyMo group for their feedback on the work I did in Antwerp.

Thank you to Juan de Lara for giving me precious advice on my work in Madrid.

I would also like to thank the Natural Sciences and Engineering Research Council of Canada for supporting me financially during the last four years.

I thank Indrani Vasudeva Murthy for revising a preliminary version of this thesis.

I thank Jürgen Dingel, Antonio Vallecillo, Jörg Kienzle, and Clark Verbrugge for reviewing this thesis.

Finally, I wish to thank all my friends and family whose support meant a lot to me.

Abstract

Systems developed today are increasing in complexity. Model-Driven Engineering (MDE) attempts to solve the issues related to complexity through the use of models to describe systems at different levels of abstraction. Multi-Paradigm Modelling (MPM) promotes modelling all parts of the system, at the most appropriate level(s) of abstraction, using the most appropriate formalism(s), to reduce accidental complexity. MPM principles state that transformations too should be modelled explicitly. Model transformations are at the very heart of MDE. Transformations allow one to execute, analyse, synthesize code, optimize, compose, synchronize, and evolve models.

Despite a robust theoretical foundation, model transformation still suffers from scaling and correctness problems. The growing interest in model transformation has lead to a plethora of model transformation languages. They provide tremendous value for developers, but in all existing implementations, the transformation language is hard-coded. This thesis contributes to the engineering of model transformation languages at the foundation level, following MPM principles. It proposes a framework for designing transformation languages tailored to the problem to be solved. As a result, model transformation languages engineered in this framework maximally constrain the modeller to only use the constructs needed. The aim is to increase the modeller's productivity, by raising the level of abstraction at which transformations can be specified and by lowering the mismatch between model transformation languages and their application domain.

After thoroughly analyzing the uses of model transformation and their supporting languages, we extract what is common to approaches and express model transformation at the level of their primitive building blocks. We introduce T-Core, a collection of transformation language primitives for model transformation. A Python implementation of T-Core is developed. It offers an API of primitive transformation operations that act on models represented as graphs. This opens the door for non-MDE developers to "properly" interact and manipulate models, making the link between the programming world and the modelling world. In the framework developed, model transformation languages are modelled explicitly. This supports developers in creating custom-built transformation languages. The approach semi-automatically generates model transformation languages adapted to the application domain. MoTif is another model transformation language engineered with this framework. Its syntax and semantics are completely modelled, as well as its execution engine. MoTif is the result of merging T-Core with DEVS, a discrete-event simulation formalism. It thus introduces the notion of time in model transformation. This allows one to easily model reactive systems and consequently optimize and calibrate them. Finally, the notion of exception handling in model transformation is explored to strengthen the robustness and dependability of the software built using this technology.

Abbrégé

Les systèmes développés aujourd'hui sont de plus en plus complexes. Pour résoudre les problèmes liés à la complexité, l'Ingénierie Dirigée par les Modèles (IDM) utilise des modèles qui décrivent les systèmes à différents niveaux d'abstraction. La Modélisation à Paradigmes Multiples (MPM) renchérie cette approche en modélisant toutes les composantes du système, aux niveaux d'abstraction les plus appropriés, tout en utilisant les formalismes les plus adéquats, afin de réduire toute complexité accidentelle. Les principes MPM stipulent que les transformations doivent aussi être modélisées explicitement. Les transformations de modèles sont au cœur de l'IDM. Elles permettent d'exécuter, d'analyser, de générer le code, d'optimiser, de composer, de synchroniser et de faire évoluer les modèles.

Bien que la transformation de modèles soit basée sur de solides théories, les problèmes de mise à l'échelle et de validité restent néanmoins encore à résoudre. Vu l'intérêt suscité par la transformation de modèles, on observe de nos jours une vaste sélection de langages de transformation de modèles. Bien qu'ils apportent une énorme plus-value au développeur, l'implémentation de ces langages de transformation demeure cependant codée en dur. Cette thèse contribue aux fondements de l'ingénierie de langages de transformation de modèles, tout en suivant les principes MPM. Elle propose un système qui permet la conception de langages de transformation adaptés au problème à résoudre. Ces langages de transformation restreignent au maximum le modélisateur à n'utiliser que les concepts nécessaires. Le but est d'accroître la productivité du modélisateur, en élevant le niveau d'abstraction auquel les transformations sont spécifiées, tout en réduisant l'inadéquation du langage de transformation de modèles avec son domaine d'application.

Après avoir analysé les différents usages des transformations de modèles et de leurs langages, nous avons identifié et extrait la partie commune à toutes les approches. Ceci permet alors de définir les transformations de modèles à partir des concepts essentiels qui les composent. Nous présentons alors T-Core, une collection d'opérateurs primitifs pour la transformation de modèles. T-Core est implémentée en Python, offrant ainsi une API disponible aux opérations primitives de transformation de modèles qui agissent sur des modèles représentés sous forme de graphes. Ceci permet à des programmeurs de « proprement » interagir avec des modèles et de les manipuler, faisant ainsi le lien entre le monde de la programmation et celui de la modélisation. Le système établi dans cette thèse modélise de manière explicite les langages de transformation de modèles et permet alors de créer des langages de transformation personnalisés. L'approche génère semi-automatiquement des langages de transformation adaptés au domaine d'application. MoTif est un autre langage de transformation de modèles construit à partir de ce système. Sa syntaxe, sa sémantique et son moteur d'exécution sont entièrement modélisés. MoTif est le résultat de la fusion entre T-Core et DEVS, un formalisme de simulation à événements discrets. MoTif permet alors d'introduire la notion de temps dans les transformations de modèles, ce qui permet de facilement modéliser des systèmes réactifs et, par conséquent, les optimiser et les calibrer. Finalement, nous explorons la notion de gestion d'exception au sein des transformations de modèles, afin de renforcer la fiabilité des logiciels bâtis à l'aide de cette technologie.

Contents

List of Figures

List of Tables

Introduction

Context

The past fifty years have seen a drastic increase in the complexity of the systems we design and use. In particular, major innovations of the last decade focused on collaboration and integration of individual systems. Despite the advances in programming languages and supporting integrated development environments, the development of complex software systems requires an enormous effort. The main reason behind the difficulty of developing complex systems is the conceptual gap between the problem to solve and the implementation using current code-centric technologies. This raises two issues we notice in current realizations: (1) the development process is not optimal and (2) there is often a mismatch between the functional needs derived from the problem and the delivered software. Some ways to tackle this complexity are through the use of abstraction [Dah02], problem decomposition [CLR00], and separation of concerns [KLM+97]. Model-driven approaches to systems development move the focus from third-generation programming language (3GL) code to *models*. The objective of model-driven development is to increase productivity and reduce time-to-market by enabling development at a higher level of abstraction and by using concepts closer to the problem domain at hand, rather than the ones offered by programming languages.

Model-Driven Engineering (MDE) [SVC06] is now considered a well-established development methodology. It attempts to solve these issues through the use of abstraction, bridging the gap between the problem and the software implementation. The MDE approach is to support systematic transformations of problem-level abstractions into their implementations. To bridge the gap between the application domain and the solution domain, MDE uses models to describe complex systems at multiple levels of abstraction and through automated support for transforming and analyzing models. MDE, and in particular, *domain-specific modelling* [GTK+07], is an approach that allows one to manipulate models at the level of abstraction of the application domain the model is intended for, rather than at the level of computing. MDE considers models and transformations as first-class entities. A model represents an abstraction of a real system, capturing some of its essential properties, to reduce accidental complexity. Models are used to specify, document, simulate, test, verify, and generate code for applications. In software language engineering terms, a model conforms to a *meta-model* [Küh06b, Küh06a]. A meta-model defines the abstract syntax and static semantics of a (possibly infinite) set of models. A model is thus typed by its meta-model that specifies its permissible syntax, often in the form of constraints. A common representation of meta-models uses the Unified Modelling Language (UML) Class Diagram notation [Obj09] with Object Constraint Language (OCL) constraints [Obj06b]. MDE allows one to manipulate these models through the use of *model transformation*. A model transformation transforms a source model into a target model, both conforming to their respective meta-models. The Object Management Group (OMG) has proposed the Model-Driven Architecture (MDA), which promotes model transformation at the heart of MDE.

The Query, Views, and Transformations (QVT) language [Obj08] is a recent addition to the OMG's set of standards.

Today's research in the field of MDE focuses on the applicability and scalability of its solutions to industrial problems. Complementary to MDE, *Multi-Paradigm Modelling* (MPM) [MV04] addresses these issues and formulates a domain-independent framework. MPM promotes modelling all parts of the system, at the most appropriate level(s) of abstraction, using the most appropriate formalism(s), to reduce accidental complexity. One key aspect of MPM is multi-abstraction. A model abstraction is a view of a system exhibiting some of its properties while hiding others. Multi-abstraction is thus the ability to express models at different levels of abstraction. MPM realizes that systems can be represented in different modelling languages or formalisms. MPM, in particular multi-abstraction and multi-formalism modelling, is enabled by the use of meta-modelling and model transformation. Instead of describing their behaviour in terms of code, MPM principles state that transformations too should be modelled explicitly. The developer can then manipulate models by means of model transformation. Transformations allow one to execute, analyse, synthesize code, optimize, compose, synchronize, and evolve models. Model transformations are at the very heart of MDE.

Problem Statement and Thesis Proposition

Despite a robust theoretical foundation, model transformation still suffers from scaling and correctness problems in an industrial context. The growing interest in model transformation has lead to a plethora of model transformation languages expressed in different paradigms, *e.g.*, template-based, rule-based, triple graph grammars, with or without explicit control flow [CH06]. They are supported by various implementations such as *AGG* [Tae04], *ATL* [JK06], *AToM*^{3} [dLV02], *GReAT* [AKK$^+$06], *MOFLON* [AKRS06], *QVT* [Obj08], *VMTS* [LLMC05], just to name a few. They provide tremendous value for developers, but in each implementation the transformation paradigm is hard-coded to be used as is [BBG$^+$06].

This thesis contributes to the engineering of model transformation languages at the foundation level, following MPM principles. This is done by modelling everything *explicitly* at the most appropriate level(s) of abstraction using the most appropriate formalism(s). In this approach, the model transformation language is modelled at the syntactic level (abstract and concrete). Moreover, the semantics of such transformation models is also modelled through the use of meta-modelling and model transformation. The aim is to increase the developer's productivity, by raising the level of abstraction at which transformations can be specified and by lowering the mismatch between model transformation languages and their application domain, *i.e.*, minimizing accidental complexity. Therefore, the work presented here provides a framework for building such model transformation languages, and illustrates its applicability by designing and implementing a new model transformation language following the MPM principles for the core algorithms, the transformation language building blocks, and the transformation formalism. The approach presented here focuses on the expressiveness of model transformation.

Since this work focuses on the foundation level of model transformation (software) development, it is important to realize there are four levels of users in this framework:

1. The *end-user* operates the system implemented. For him, the system modelled is strictly a software application that suits his need.
2. The *modeller* designs and operates models of the system. A domain-specific engineer is typically a modeller who is an expert in his domain, modelling at a level of abstraction as close as possible to his domain of expertise.
3. The *transformation engineer* is aware of the domain of expertise. He designs a framework well-suited for the domain-specific engineers. Note that in some cases, the transformation engineer may as well be the domain-specific engineer (this is analogous to a database administrator and the programmers developing the database system). The term engineer is sometimes replaced by modeller.
4. The *transformation language engineer* builds the transformation language to be used by the transformation engineer. He engineers a transformation language with the necessary and sufficient features needed for a class of application domains.

This thesis provides the necessary tools for the transformation language engineer to build transformation languages that are problem-specific.

Contributions

The goal of this thesis is to improve our understanding of model transformation, facilitate the engineering of model transformations, and increase the quality of the software produced by this technology. To achieve this goal, the focus is shifted to the languages that allow us to develop transformation models. The main outcome of this thesis is a framework and methodology to engineer model transformation languages that are tailored to the specific domain of application and the problem to solve. The contributions of this thesis are the following:

1. The ability to create **custom transformation languages** according to the problem to be solved.
2. The re-use of modelling languages to define the **scheduler** of a model transformation, instead of inventing a new one for each transformation language.
3. A framework for defining transformation languages that are **completely modelled** in a multi-paradigm modelling sense. Consequently, this allows one to cleanly specify higher-order transformations. *T-Core* encapsulates the building blocks of this framework.
4. A precise **common representation** of essential model transformation language features.
5. The introduction of **exception handling** in model transformation to ensure the dependability of the software built.
6. The integration of model transformation in a **programming** framework, as opposed to a modelling framework.
7. An **in-depth comparison** of existing model transformation languages, approaches, and paradigms.
8. The weaving of the model transformation paradigm with the discrete event simulation paradigm, ensured by **timed model transformations**. This provides a model-driven approach to modelling

and simulation-based design. *MoTif* is a novel model transformation language built using this approach.

Outline

This thesis is divided into four parts. Part I proposes an extended survey of model transformation. It comprises two chapters. Chapter 1 explains what a model transformation is and where it is applied. Chapter 2 exposes an overview of existing model transformation languages and their features.

Part II focuses on the foundations of model transformation. Chapter 3 proposes a common basis for defining model transformation languages and Chapter 4 is dedicated to the implementation of this transformation core. To complement this idea, Chapter 5 motivates and describes the need for custom-built transformation languages, focusing on the specification part rather than on the transformation engine.

Part III illustrates the proposed framework for engineering transformation languages tailored to the needs of the transformation engineer. Chapter 6 presents a formalism for modelling and simulation purposes. It is entirely modelled following MPM principles. Then, Chapter 7 shows how to define a novel transformation language in the framework described in Part II, combining the formalism presented in Chapter 6, as well as Chapters 3 and 5. Chapter 8 takes advantage of the fact that this novel transformation language is entirely engineered following MPM principles, to extend it with fault-tolerance capabilities, such as exception handling.

Part IV demonstrates applications of the transformation language developed throughout the thesis. Chapter 9 shows the application of the language to modelling and simulation-based design. Chapter 10 evaluates the expressiveness of the language and Chapter 11 evaluates its performance. A final conclusion summarizes the conclusions described at the end of each chapter and proposes an outlook for future research.

Part I

A Survey of Model Transformation

"*Rien ne se perd, rien ne se crée, tout se transforme.*"
(Nothing is lost, nothing is created, everything is transformed.)

Antoine Lavoisier

1

What is Model Transformation?

Part I of the thesis presents a thorough survey of the different applications of model transformation, of the features that model transformation languages exhibit, and a comparison of the most common model transformation languages and approaches that exist today. This chapter focuses on the first aspect. But first let us explain what a model transformation is.

1.1 Definition of Model Transformation

In MDE, models are the primary engineering artefacts. Because of the compelling need to manipulate the data stored in models, transformations play a fundamental role in model-driven development. In fact, they are so crucial that transformations are considered as "first-class citizens" in MDE. Furthermore, transformations themselves are modelled, hence the term *model transformation*.

1.1.1 Previous Definitions

The OMG defines a model transformation as "the process of converting one model to another model of the same system" [Obj03]. This definition is too restrictive as it only considers transformations that produce a different model from the initial one. In 2003, Kleppe *et al.* published a book on practical applications of the MDA. The definition they propose extends the OMG's definition with specific keywords: "a model transformation is the automatic generation of a target model from a source model, according to a transformation definition" [KWB03]. The terms *source* and *target* models refer to the functional nature of a transformation, taking a source model as input and outputting a target model. Another key aspect in this definition is the "automatic generation". This implies that the transformation is not specified directly on models but at a higher (meta-)level and the transformation is automatically generated from its definition.

Figure 1.1 illustrates this definition. A transformation is defined at the meta-model level. From the transformation definition[1], the transformation is automatically generated and is executed on the models conforming to their respective meta-models. The transformation can be interpreted or the result of a compilation. In this figure, the source and target meta-models may be different or the same, depicting *exogenous* or *endogenous* transformations respectively. Furthermore, the source and target

[1] In the literature, this term also appears as transformation specification or transformation design.

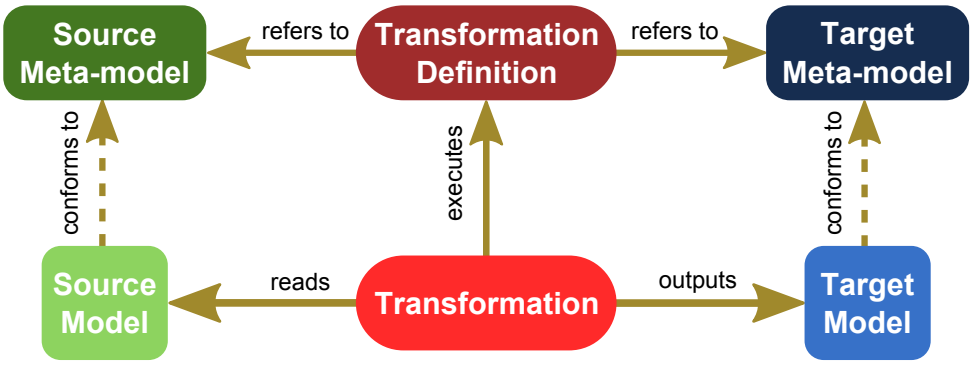


Figure 1.1: Model transformation terminology.

models may be different or the same, depicting *out-place* or *in-place* transformations respectively. These terms were first introduced by Mens and Van Gorp in [MVG06]. Moreover, the figure can be extended to allow a transformation to operate on multiple source and/or target models.

1.1.2 Proposed Definition

Since these definitions were proposed (over 8 years ago), model transformation has been applied in a much wider range of applications than expected at the time, as illustrated in the following section. Therefore, the definition of a model transformation must be revised to fit in a more general context. In this thesis, we consider a model transformation as **the automatic manipulation of a model with a specific intention**. It is automatic because, as in the definition from Kleppe *et al.*, the transformation is automatically generated from a higher level specification. A model transformation is a manipulation of a model because it encapsulates any modification or alteration of a model, which entails—at the minimum—reading, creating, and modifying model elements. Model transformation can work at different levels of abstractions, modify the syntax of a model, or even define/alter the semantics of a language. Its application may vary from simple model element modifications to defining the semantics of a language or synchronizing different views of a same model. An exhaustive list of applications of model transformation will be given in Section 1.2. The diversity in the applications thus implies that each model transformation is characterized by the intention behind its usage.

This more general definition must be placed in a multi-paradigm modelling context where *everything* is modelled. In that sense, any change in the system always happens on a model. Let M be a model that conforms to a meta-model MM. A transformation on M is an intentional change or alteration of the model, which yields a model M' conforming to a meta-model[2] MM'. Moreover, since we model everything explicitly, then a change or modification of a model must be itself modelled: we therefore have models of transformations. The meta-model of a transformation model defines all possible changes for the same intention from an instance of MM to an instance of MM'.

[2]Note that MM and MM' may be the same.

1.1.3 Program versus Model Transformation

At first glance, a model transformation, just like any other program, is a data manipulation program: take input data and produce output data. However, model transformation is a form of *meta-programming* [CI84], *i.e.,* programs that manipulate meta-data, such as programs, specifications, schemata, etc. Manipulating "semantically rich" data, such as programs, requires specialized facilities: thus the need for program transformation and compilers. However, one may argue that the difference between a model and a program resides in (1) the level of abstraction they are specified in and (2) the cognitive effort needed by a human to map the computer-based model to the system in the real world. Similarly, model transformation can work at different levels of abstraction. Its application may vary from simple model element modifications to defining the semantics of a language or synchronizing different views of the same model.

The distinction between a program and a model transformation is not clear-cut. The first distinction is biased towards tree versus graph rewriting. Traditionally, program transformation has used term rewriting—that is, tree transformations—as its underlying theory, *e.g.,* Stratego [Vis01]. A good part of the model transformation community that works on the theoretical aspects of model transformations views graph transformation as the most appropriate paradigm for model transformations. This view applies well to models such as Petri nets, which are truly graph-like. However, some models may have natural tree-like nesting (often represented as composition in their respective meta-models). Thus, viewing model transformation as transforming graphs without special attention to the tree structures is unnecessarily restricted. The bottom line is that both program and graph transformation systems transform graph structures and both can be used to transform programs or models. In practice, the choice between the two boils down more to how well the source and target languages are supported. For example, a system that can transform Java should come with a rich library of built-in program analyses, starting with the simplest query for all the interfaces a given class implements.

Another distinction is that programming languages traditionally used grammars as their syntax definition formalisms; most of the modelling world uses richer formalisms such as class models, *e.g.,* the Meta-Object Facility (MOF) [Obj06a]. The latter are more properly "concept definition" formalisms with generalization and property relations (with properties possibly divided into references and composition).

The final distinction is that models are more diverse. Although the term "model" normally refers to system abstractions above the implementation code [GTK+07], that is—artefacts such as requirements and design specifications or analysis models, model-driven environments sometimes represent programs in the same form as models, and treat them alike. Thus, we can conclude that model transformations operate on more diverse artefacts than program transformation, as these artefacts may include programs, but also other artefacts, such as specifications and schema definitions. In other words, one could view program transformation as a narrower field than model transformation.

1.2 Types and Uses of Model Transformation

In MDE, a model is a static representation of some information of a system. Model transformation allows one to manipulate models in order to modify some of their properties, answer queries, generate code, simulate, migrate, optimize, compose, or synchronize them. These different types of model transformations capture the intention behind the manipulation they perform on models. The following classification extends the taxonomy of model transformation proposed by Mens and Van Gorp [MVG06].

1.2.1 Access/Modify Operations

Strictly speaking, a model consists of a set of elements; for example if a model is represented by an instance of a UML class diagram [Obj09], then the elements correspond to objects and links. These elements are usually structured; for example in the form of an ordering or containment relationships. Properties of the model are encapsulated in elements; for example as attribute values in the objects.

The simplest operations on a model are *adding* an element to the model, *removing* an element from the model, *updating* an element's properties, *navigating* through the elements, and *accessing* the properties of an element. These primitive operations are also known as the **CRUD** operations (create, read, update, delete) as first introduced by Kilov in [Kil90]. From a pragmatic point of view, a model transformation is a sequence of CRUD operations. The effect that a specific sequence has on a model determines the nature of the transformation. The simplest of these is the simple access or modification of one or more elements of the model or their properties.

1.2.2 Query

Queries take their origins from data manipulations in databases. A query is an operation that requests for some information about a system. This operation takes as input a model M and outputs a **view** of M. A view is a projection of (a subset of) the properties of M. Therefore a query is a transformation as it is a projection, obtained by CRUD operations on the properties of M. We distinguish between two types of views: restrictive and aggregated views. A **restrictive view** reveals all, none, or some of the properties of M. For example, the query "retrieve all cycles in a Causal Block Diagram" outputs a view of the causal block diagram model represented as a cyclic graph composed of strongly connected components. Another kind of restrictive view is the output of the query "show only classes/associations of a class diagram". An **aggregated view** is a restriction of M modifying some of its properties. One example of such a view is the average of all costs per catalogue product in a relational database schema. Or, in a hierarchical model, show top-level elements only with an extra attribute denoting the number of sub-elements.

These definitions of a query and a view differ from those proposed by the Query/View/Transformations initial call for submissions [GGKH03]. The authors define a query as "an expression that is evaluated over a model" and a view as "a model which is completely derived from another model". Although stated from a more pragmatic perspective, they are nevertheless compatible with our definitions.

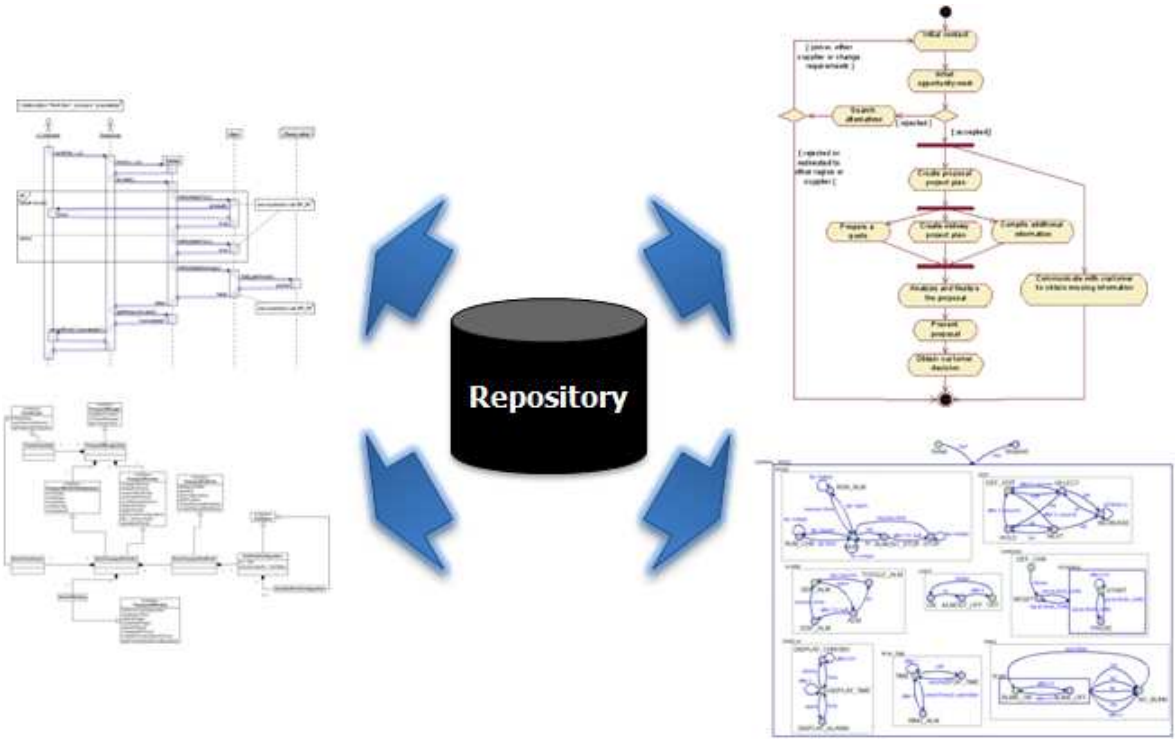
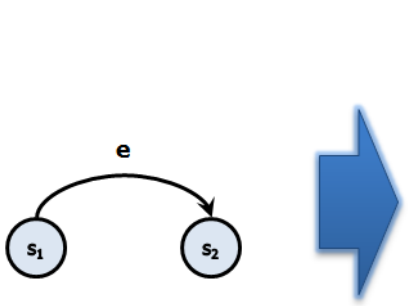


Figure 1.2: Multiple views of a single repository model.

There can be multiple views for different observers of the same model. For example in Figure 1.2, the central model, called **repository**, can be viewed as a UML class diagram to highlight the main concept elements, as a UML sequence diagram to outline the interaction between these elements, as a UML activity diagram that specifies a workflow scenario involving some of the elements, or a Statecharts model to define their behaviour. Some views are read-only and others are write-enabled. In any case, views must be kept consistent with the repository. In [GdL06], Guerra and de Lara show how multi-view consistency can be ensured by a model transformation that defines a relation between the repository and a view. For read-only views, any change is propagated from the repository to all views (or those that are affected by the change). For write-enabled views, any change is propagated from a modified view to the repository, which will propagate the changes to the other views. This principle is known as the *Model/View/Controller* in software engineering [KP88].

1.2.3 Synthesis

Synthesis is a transformation from a higher level specification to a lower level specification. We talk about **code generation** when the target specification is source code in a programming language. A typical example of code generation is when design models (such as UML class diagrams) are translated into source code (such as C#) as supported by most UML editors and CASE tools, such as Enterprise Architect [Sys00]. Another typical case of code generation is when a *domain-specific model* (DSM) or abstract model is translated into source code. For example, as illustrated in Figure 1.3, a State-




```
if e == 0:   # event "e"
    if table[1] and self.isInState(1) and self.testCondition(3):
    if (scheduler == self or scheduler == None) and table[1]:
        self.runActionCode(4)          # output action(s1)
        self.runExitActionsForStates(-1)
        self.clearEnteredStates()
        self.changeState(1, 0)
        self.runEnterActionsForStates(self.StatesEntered, 1)
        self.applyMask(DigitalWatchStatechart.OrthogonalTable[1], table)
        handled = 1
    if table[0] and self.isInState(0) and self.testCondition(4):
        if (scheduler == self or scheduler == None) and table[0]:
            self.runActionCode(5)      # output action(s2)
            self.runExitActionsForStates(-1)
            self.clearEnteredStates()
            self.changeState(0, 0)
            self.runEnterActionsForStates(self.StatesEntered, 1)
            self.applyMask(DigitalWatchStatechart.OrthogonalTable[0], table)
            handled = 1
```

Figure 1.3: Statecharts to Python compilation.

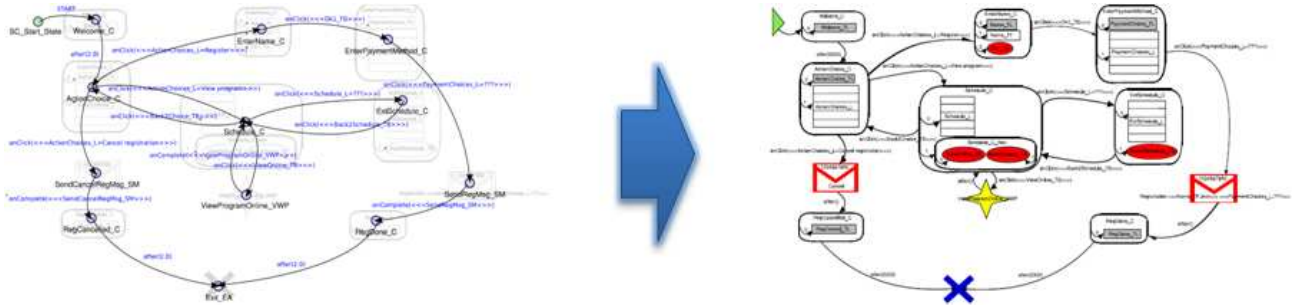


Figure 1.4: Extracting user-interface behaviour from a Statecharts model into a PhoneApps model.

charts model is compiled into Python source code in order to simulate it. This is a **model-to-code** transformation which is a special case of a **model-to-model** transformation, since source code can be modelled by its abstract syntax tree and the meta-model is the grammar of the programming language.

1.2.4 Reverse engineering

Reverse engineering is the inverse of synthesis: it extracts higher level specifications from lower-level ones. For example, Figure 1.4 shows a transformation from a Statecharts model to a DSM of the PhoneApps language for a conference registration mobile application [MV10b].

1.2.5 Translational Semantics

Harel and Rumpe [HR00] define a modelling language (or *formalism*) by syntax and semantics components. The syntax comprises the *abstract syntax* representing the essence of the concepts of the language. The abstract syntax can be mapped to several *concrete syntaxes* (textual representation,

graphical visualization, etc). These two components specify the different concepts of the language and how they are represented. The semantics of a formalism is specified by a unique *semantic mapping function* which maps each model element in the language onto an element in a *semantic domain*. For example, the meaning of a Causal Block Diagram is given by mapping it onto an Ordinary Differential Equation. For practical reasons, semantic mapping is usually applied to the abstract rather than to the concrete syntax of a model. Note that the semantic domain is a modelling language in its own right which needs to be properly modelled (and so on, recursively). In practice, the semantic mapping function maps abstract syntax onto abstract syntax.

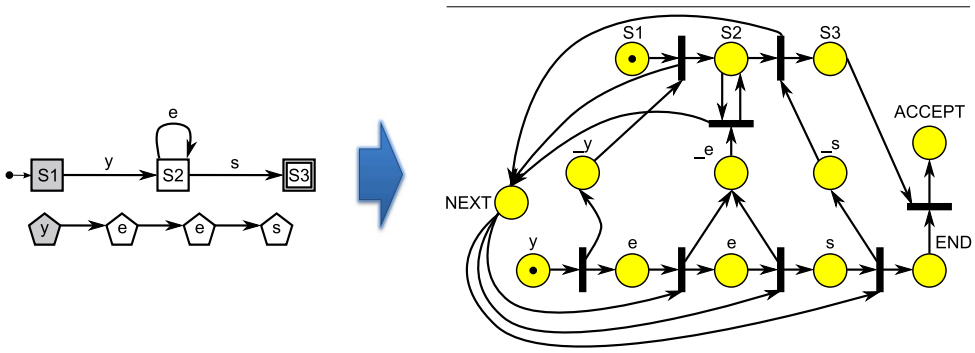


Figure 1.5: A Finite State Automata to Petri net semantic translation.

The meta-model of the formalism represents the abstract syntax and static semantics of the language. A model transformation can be used to define the dynamic semantics of the language. In the case of translational semantics, the formalism's semantic mapping function is defined by a transformation and its semantic domain is a modelling language. When defining a translational semantics, we transform a model in one formalism into a model in another formalism. Then the semantics of the source formalism is given in terms of the semantics of the target formalism. For example in [KMS+09], we define the semantics of Finite State Automata in terms of Petri nets. That is, the model transformation translates any finite state automata model into a semantically equivalent Petri net model, as depicted in Figure 1.5.

1.2.6 Simulation

A model transformation can be used to simulate models: it updates the state of the system modelled. In this case, the target model is then an "updated version" of the source model (in-place transformation). For instance, Heckel has described the behaviour of a simplistic Pacman game in [Hec06]. There, the transformation specifies the transitions that a Pacman game instance is allowed to take (pacman and ghost moving in each direction, pacman eating a pellet, ghost eating pacman). In software language engineering terms, this is called the **operational semantics** of the Pacman language. The execution of the transformation shows the trace of the model's behaviour. For example, Figure 1.6 depicts the trace of the simulation of a Finite State Automata model.

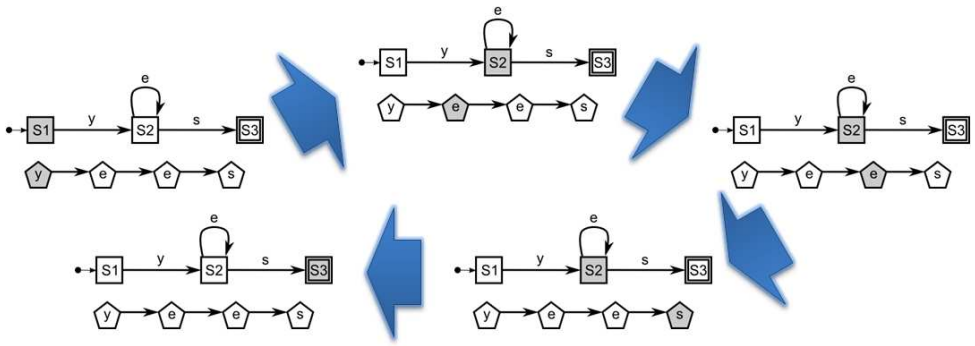


Figure 1.6: The animation trace of a Finite State Automata.

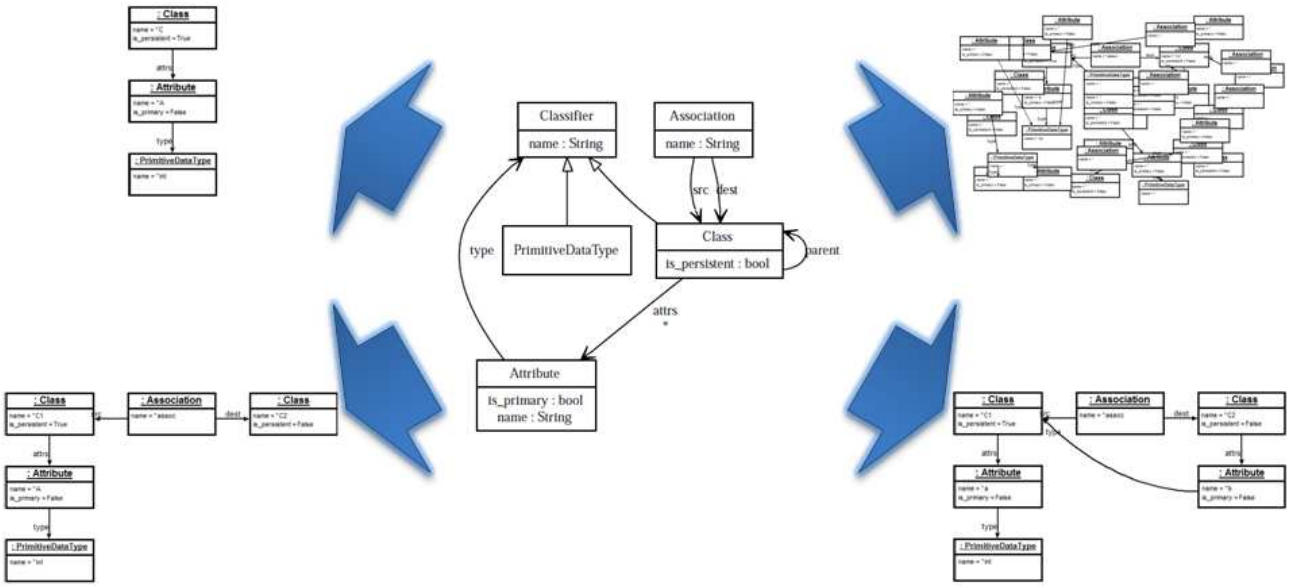


Figure 1.7: Generation of valid UML class diagrams.

1.2.7 Meta-Model Instance Generation

The meta-model of a language can be defined by a grammar such as in Extended Backus-Naur Form (EBNF). In the modelling environment DiaMeta (formerly called DiaGen [VM95]), meta-models are defined by **graph grammars**[3]. This is a model transformation that allows one to generate all possible instances of the language. For example, Figure 1.7 shows possible UML class diagrams generated for the meta-model in the middle. In [EKT09], the authors generate Statecharts models from its meta-model also via a graph grammar. This technique is very useful for model-based testing [DJK+99] of model transformations as it allows one to automatically generate input tests to verify the correctness of a transformation.

[3]Hypergraph grammars to be more precise.

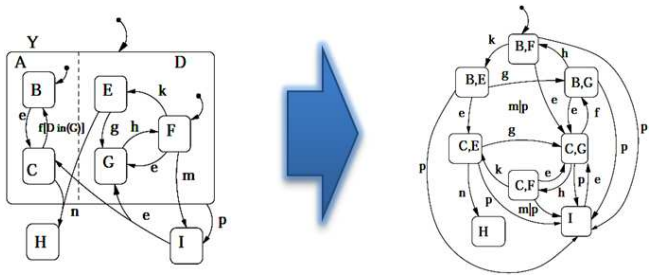


Figure 1.8: Normalization of a Statecharts model by flattening it.

1.2.8 Migration

Another use of model transformation is in model migration. In [MBP99], the authors define migration as a transformation from a software model written in one language or framework into another language, keeping the models at the same level abstraction. When a language, *e.g.*, Enterprise Java Beans 2.0 (EJB2), *evolves* to a newer version, *e.g.*, Enterprise Java Beans 3.0 (EJB3), one must migrate all models conforming to the meta-model of EJB2 so that they conform to the new meta-model of EJB3. Instead of having to migrate each model individually, an automatic process would be desired. Thanks to the fact that model transformations are defined on meta-models and operate on models, Cicchetti *et al.* [CDREP08] have proposed a model-to-model transformation to automatically migrate models.

1.2.9 Normalization

Normalization aims to decrease the syntactic complexity of models. **Desugaring** is when complex language constructs (syntactic sugar) are translated into more primitive language constructs. **Simplification** is when all uses of a language construct are transformed in a normal or canonical form. Figure 1.8 shows how a Statecharts model is normalized to its flattened form where OR- and AND-states are replaced by the appropriate states and transitions. **Parsing** the concrete syntax of a modelling language back to its abstract syntax is also considered as a normalization, which can be implemented by a model transformation involving the meta-model of the concrete syntax and the meta-meta-model of the language.

1.2.10 Optimization

Improving *operational qualities* of models is crucial for scalability. Nevertheless, optimization preserves the semantics of the model. Optimization tasks are typically done on architectural or design models. For instance in Figure 1.9, we optimize a model representing a spreadsheet with dense data converting the list representation of the cells into a two-dimensional table.

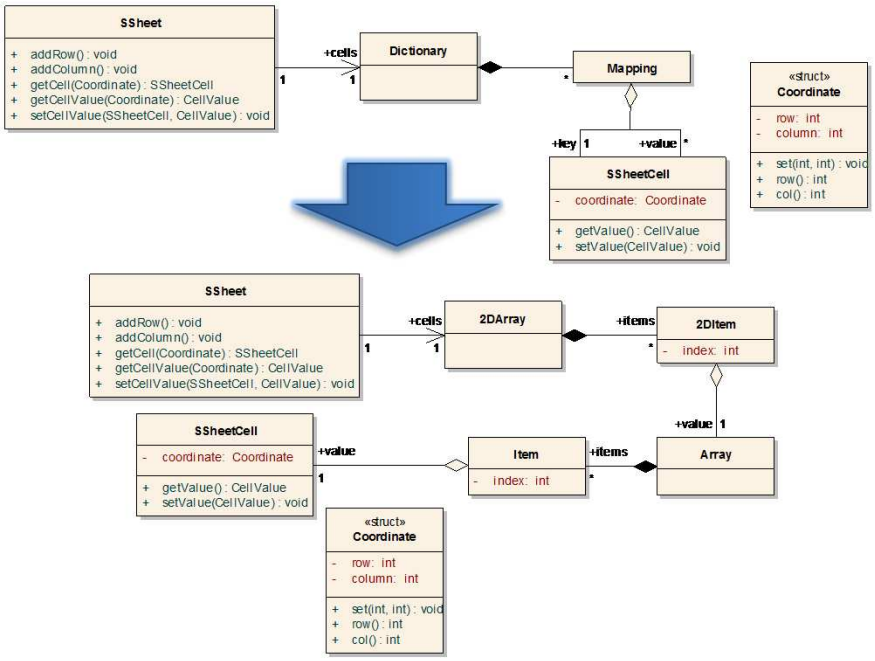


Figure 1.9: List to table optimization.

1.2.11 Restructuring

Model refactoring is a restructuring that changes the internal structure of the model to improve certain quality characteristics (such as understandability, modifiability, re-usability, modularity, adaptability) without changing its observable behaviour [Gri91]. This involves applying *refactoring* as defined by Fowler [Fow99] to models. Zhang *et al.* [ZLG05] proposed a generic model transformation engine that can be used to specify refactorings for DSMs.

1.2.12 Composition

Model composition integrates models that have been produced in isolation into a compound model. Typically, each isolated model represents a concern which may overlap. There are two techniques to compose concerns as illustrated in Figure 1.10. On the one hand, **model merging** creates a new model such that every element from the union of both models is present exactly once in the merged model. In [EPK06], the authors propose a transformation language that allows one to compute the merged models given two models conforming to the same meta-model. On the other hand, **model weaving** creates correspondence links between overlapping entities. In this case, a generic meta-model is defined for correspondences which are thus modelled.

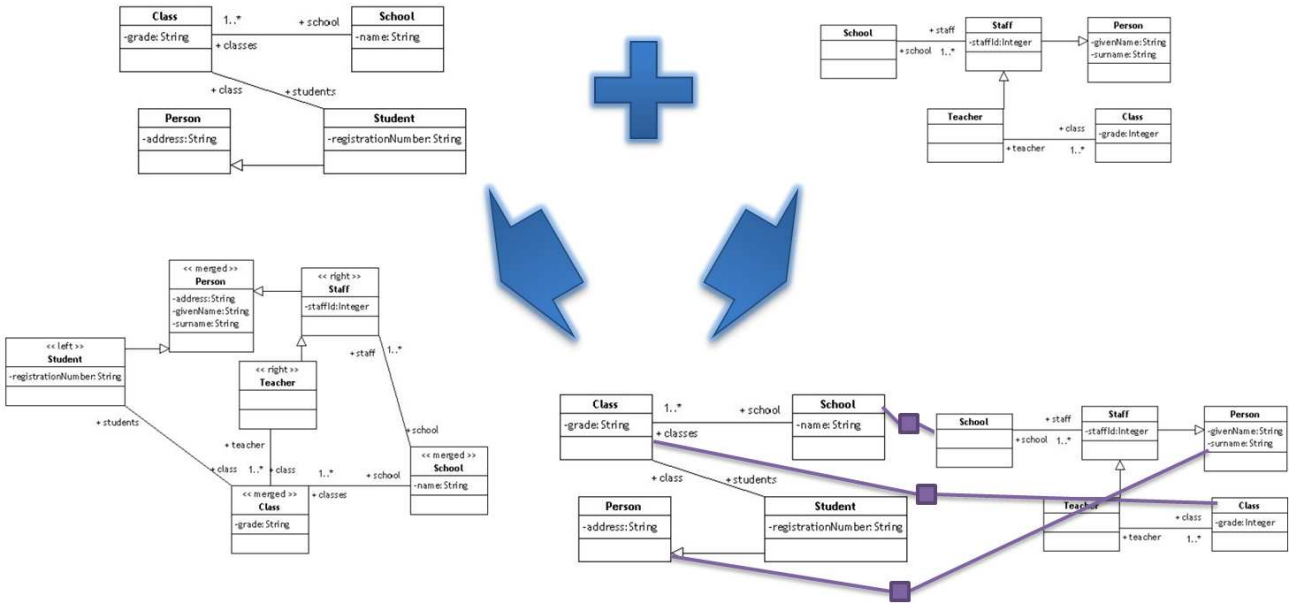


Figure 1.10: Composition of two models through model merging (on the bottom left) and model weaving (on the bottom right).

1.2.13 Synchronization

Model synchronization integrates models that have evolved in isolation but that are subject to global consistency constraints. In contrast with composition, synchronization requires that changes are propagated to the models that are being integrated. Source model changes are propagated to corresponding target model changes. Referring back to Figure 1.2, this technique is typically applied when multiple views of a repository are accessed or modified. **Incremental transformations** such as in [BÖR+08] are well-suited for this task. Furthermore, synchronization must be ensured in both directions: in this case, multi-directional transformation [Sch94, Obj08] is used to manage inconsistencies.

1.2.14 Classification of Transformation Types

Mens and Van Gorp [MVG06] have classified only some of these different types of transformations along several dimensions. Table 1.1 classifies transformation types depending on whether they are applied on the same model or not and whether they refer to the same meta-model or not. For example on the one hand, a model transformation that simulates a model modifies the source model and the resulting model still conforms to the source meta-model. On the other hand, a model transformation that synthesises a DSM to source code produces a different model that does not conform to the source meta-model. Note that an in-place transformation is more general than an out-place transformation as it can emulate the latter by first copying the model and then operating on the copy.

Composition is out-place by definition. However, it can be either endogenous or exogenous depending on the intention of the composition. For instance, if a model transformation is used to merge

	Endogenous	Exogenous	Either
In-place	Access/Modify, Simulation	X	X
Out-place	Restrictive query, Simplification	Aggregate query, Synthesis, Reverse engineering, Migration, Desugaring	Composition, Synchronization
Either	Optimization, Restructuring	X	X

Table 1.1: Model versus meta-model concerns of transformations.

	Endogenous	Exogenous	Either
Horizontal	Access/Modify, Simulation	Migration	Composition
Vertical	Restructuring, Restrictive query, Optimization, Simplification	Aggregate query, Synthesis, Reverse engineering, Desugaring	X

Table 1.2: Abstraction level of transformations.

different restrictive views of the same repository, then the merged model conforms to the same meta-model. However, if composition is used to merge an old version of a model with a newer one, then the target model would typically not conform to one of the source meta-models. The same reasoning holds for synchronization. As for optimization and restructuring operations, these transformations modify the model keeping it in the same language (*i.e.*, the target model still conforms to the source meta-model). However, depending on the specific implementation, they may produce a different model or simply modify the source model.

Finally, Table 1.2 classifies transformation types along two orthogonal dimensions: whether the source and target model occupy the same level of abstraction or not and whether they refer to the same meta-model or not. For example on the one hand, a model transformation that migrates a model to conform to a new meta-model outputs a model that is still at the same level of abstraction as the source model. On the other hand, a model transformation that performs a restrictive query on a model produces a view that conforms to a subset of the source meta-model, the view being at a different level of abstraction than the source model.

Synchronization is not present in this classification because it is orthogonal to the other types of transformation. As stated previously, it can be both endogenous or exogenous. Synchronization may also produce a model on the same level of abstraction (horizontal) as the source model(s) or not (vertical). For example, suppose two domain-specific engineers are working on the same conceptual model, each on a different version of the model. When the two models are be synchronized, the modifications applied on each model are still on the same level of abstraction. In contrast, if the domain-specific engineers are each working on different aggregated views of the same model, then the synchronization between the two views with the repository model will be on different levels of abstraction.

Syntactic	Query, Synthesis, Reverse Engineering, Simplification, Desugaring
Semantic	Access/Modify, Simulation, Migration, Optimization, Restructuring, Composition, Synchronization

Table 1.3: Syntactic and semantic transformations.

Finally, Table 1.3 distinguishes transformations that affect the syntax of a model from those affecting its semantics. A syntactic transformation solely modifies the representation of the model (semantic preserving). However in a semantic transformation (semantic modifying), the output model has a different meaning than the input model, although the representation of the latter may or may not have been modified. To illustrate the application of model transformation types, let us consider a model transformation chain (sequence of model transformation applications) to compile a DSM into executable Java code. Typically, the DSM is represented in concrete syntax. A transformation will parse the model to extract its abstract syntax. On the abstract syntax level, optimizations or refactorings may be applied to improve the quality of the artefact. From the abstract syntax of the model, a transformation synthesize it directly to Java source code. After some time, the meta-model of the DSM may have evolved. Consequently, a model transformation will migrate the abstract syntax of the DSM to conform to the new meta-model. A new model transformation is then required to synthesize source code for the new version of the model.

1.3 Conclusion

The first part of the survey presented in this chapter is based on the taxonomy of model transformation proposed by Mens and Van Gorp as well as many experiences shared in the literature. To summarize, model transformation has many purposes. Given a formalism, the meta-model defines the abstract syntax (structure) and the static semantics. Model transformation provides *dynamic semantics* (behaviour) to models of the formalism. When the transformation is endogenous (the source and target meta-models are the same), the transformation is typically a *simulation* of the formalism. In this case, a model transformation describes the operational semantics of the language in the formalism. *Refactoring* is another form of endogenous transformation, typically used for optimizing or evolving the design of models. When a transformation is exogenous (different source and target meta-models), it is typically used to *translate* models from one formalism into another. For example, a domain-specific modelling formalism may be transformed into a lower (abstraction) level formalism such as Petri nets or Statecharts. In this case, the meaning of a model is given by a translational semantics into a behaviourally equivalent model. When two models are related, they can each co-evolve and thus the initial relationship does not hold anymore. Model transformation can be used to *synchronize* models, specifying a bidirectional transformation or relation between models from different meta-models. *Code generation* is another form of exogenous model transformation, considering programs as trees and thus as models. It can also be used to allow the integration of a model in a software application and to make the model executable. Other model transformations are useful to serialize models to persistent storage.

Model transformation has applications in several industrial projects. The automotive, military, airspace, and mobile industry (*e.g.*, Porsche, BMW, Nokia, Motorola) are examples of early adopters of model-driven engineering. However, model transformation still suffers from scaling and correctness problems in an industrial context.

2

Features and Approaches

This chapter completes the survey of model transformation in Chapter 1 by first examining the different features that current model transformation approaches offer. Then a thorough comparison of some of the most relevant transformation tools and languages in use today is presented.

2.1 Overview of Model Transformation Features

From the previous chapter, we can affirm that model transformation has many applications. For example, it is used to generate platform-specific models from platform-independent models and reverse engineer them, map and synchronize among models at the same or across abstraction levels, create query-based views of a system, model evolution tasks, or transform models between different languages for integration. We will now explore the different features that a model transformation language can have. It covers more than what current tools support but this framework may change because of the very active research in the field.

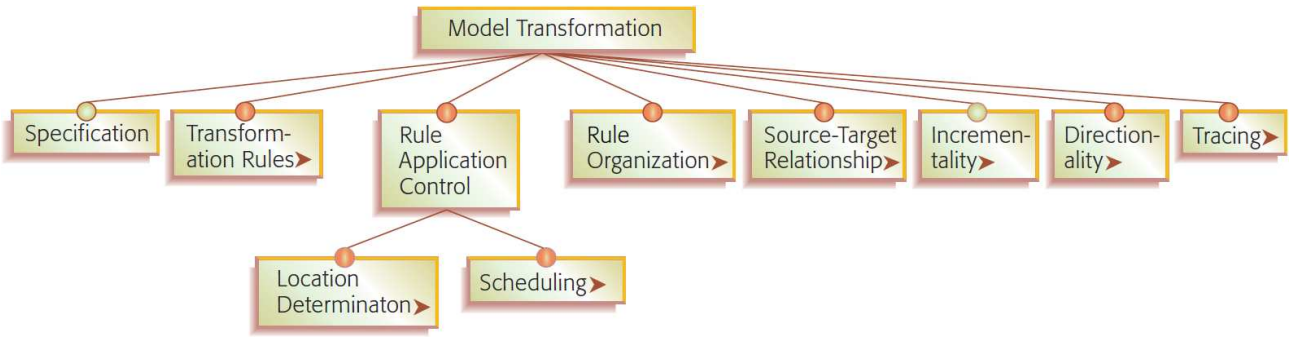


Figure 2.1: Feature diagram of model transformation languages from [CH06].

2.1.1 Transformation Language Features

In 2003 and later in 2006, Czarnecki and Helsen proposed a classification of the different features of contemporary model transformation approaches. The classification is based on a feature model [CH06]

for which the top-level features are depicted in Figure 2.1. In this section, we only highlight some of the features relevant for comparison in the remainder of this chapter.

Transformation rules are the smallest *transformation units* used to specify a transformation. In fact this feature should be renamed to transformation units so it does not restrict the classification to rule-based transformation only, although it is the most commonly used paradigm. Transformation units will be discussed in more detail in the following section.

Rule application control determines where individual rules are applied on the model and in what order the rules are executed[1]. The latter feature will be discussed in greater details in Section 2.1.3. The former feature is called **location determination**. A deterministic transformation implies that a repeated execution will always lead to the same output. For example in *Stratego*, the user defines his own traversal mechanism in a deterministic way. When several choices occur in a non-deterministic transformation, it is important to distinguish concurrent execution from one-point execution. A rule may also be applied on only one non-deterministically selected location in the model as in graph transformation approaches [EEKR97]. *AToM*^{3} [dLV02] and *VIATRA2* [VB07] offer the possibility to apply a rule on all applicable locations in the model at the same time. This may however induce conflicts in parts of a model shared by two or more applications of the rule. *AGG* [Tae04] detects such conflicts by performing a critical pair analysis on the rules and the input model. The application location of a rule may also be determined by the user interactively specifying it as in *AToM*^{3}.

Specification is the ability to specify pre- and post-conditions for the whole transformation. The specification defines a function between the source and target models. This function may be directly executable or not.

Rule organization considers the issues on the general structuring of the rules to compose them. Rules may be packaged inside modules, such as in *ATL* [JABK08], *QVT-R* [Obj08], and *VIATRA2*. When designing a transformation (typically for translational semantics) there are often redundancies in rule patterns; re-use mechanisms then come in very handy. *ATL* allows a rule to inherit from another rule: the pre-conditions and bindings of the sub-rule are computed by taking their union with those of the super-rule. The *QVT* specification presents very sophisticated re-use mechanisms, but there is no implementation available for them.

Source-target relationship refers to whether a transformation is out-place (such as in *ATL*) or in-place (such as in *AGG*). *QVT* allows one to create a new model or update an existing one.

Tracing is the runtime footprint of a transformation execution. Traceability links (or trace links) are a common form of trace information in model transformation. Traceability links connect source and target elements. They are useful for impact analysis *i.e.*, how changing a model affects another. They are also used to determine the direction of a synchronization in an N-to-M transformation (reading N source models and producing M target models where $N, M \geq 1$). Trace links can be created automatically by the transformation as in *ATL* and *QVT* to avoid a transformation unit from being applied on the same location more than once. But one might still want to have some control over their creation. In *AGG*, *AToM*^{3}, and *VIATRA2*, they are considered as any other model but have to be created

[1]The latter is called **rule scheduling** and will be covered in Section 2.1.3

manually. Tracing transformation execution is crucial for model transformation debugging. It can take the form of snapshots of the models at different steps in the transformation process. Tracing may also be expressed by having backward links from the new model to the initial (or an intermediate) model along the transformation. Traces may be automatically generated in the source or the target or in a separate storage. This is typically useful when the transformation transforms a model in one formalism into a model in another formalism. Traceable transformations are also very important when code synthesis and reverse engineering is needed. When designing a model transformation, the transformation engineer sometimes needs to relate elements conforming to different meta-models. Generic links (a generalization of trace links) can be temporarily created to fulfil this need.

An **incremental** transformation is defined as a set of relations between source and target meta-models. These relations define constraints on models to be synchronized. Change-detection and change-propagation mechanisms are then used as in [BÖR+08]. The first time it is run, the transformation creates a target model. Trace links are often automatically created. Then, if a change is detected in one of the models, it propagates this change to the other model, by adding, removing, or updating an element so that the relations are still satisfied. There are four standard scenarios in model synchronization:

- Create a target model from the source model;
- Propagate changes in the source model to the target model;
- Propagate changes in the target model to the source model;
- Verify consistency between the two models.

Directionality is a fundamental feature distinguishing unidirectional from multi-directional transformations. A **unidirectional** transformation creates (or updates) the target model only. A **multi-directional** transformation can be executed in any direction. However, it requires multi-directional rules that are conceptually defined by separate unidirectional rules, one for each direction. *Operational* rules are often unidirectional and have a functional character: given an input model, produce a target model. This entails a causality from the source to the target model. *Declarative* rules are often multi-directional: they specify a relation between both models that must be satisfied. This entails an acausal relationship between the models. For example, triple graph grammar rules are bi-directional and are specified declaratively [Sch94]. But to execute them, they are converted into seven operational rules.

- Every new element in a model has a correspondence in the other [$\times 2$]
- When an element is removed from a model, its corresponding element(s) is (are) deleted appropriately [$\times 2$]
- Enforce the consistency relations between attributes [$\times 2$]
- Create a correspondence between unmapped elements of the two models [$\times 1$]

2.1.2 Transformation Units

Transformation units are the central elements of a model transformation. A transformation unit is specified on one or more **domains** (for each meta-model). The domain expresses what paradigm the

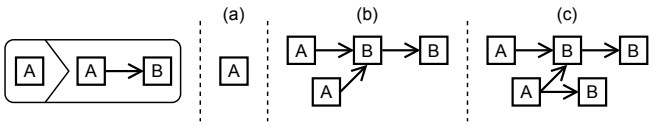


Figure 2.2: A transformation rule on the left and three models (a), (b), and (c).

transformation unit uses to operate and access elements of the models. It may be applied under certain conditions or from a given context.

They can take the form of a function, a rule, or a relation. A **function** is an imperative construct that dictates how the input model shall be modified or the target model produced, such as in *Kermeta* [FHN06]. A **rule** is a declarative construct that dictates "what" shall be transformed and not "how". It consists of a pre-condition and a post-condition. The pre-condition must be first satisfied to modify or produce model elements so that its post-condition is satisfied after its application. Figure 2.2 shows a possible graphical representation of a rule on the left. The left compartment depicts the pre-condition pattern and the right compartment depicts the post-condition pattern. A possible semantics of this rule is: if an A element is found then an A element connected to a B element shall be present. If the rule is applied to model (a), it will produce a B element and connect the A element to it. However, if it is applied to model (b), then nothing needs to be produced since there is already an A element connected to a B element. In graph transformation, the rule has a different semantics: if an A element is found then create a B element and connect A to a B. In this case, if the rule is applied to model (a) then model (b) is a possible outcome. The third type of transformation unit is a **relation**. A relation extends the notion of rule by removing the notion of direction or causality between the pre- and the post-conditions. A relation acts on domains (the meta-models involved in the transformation) and declaratively states the relationship between the elements and their properties. However, when a relation is executed, a direction must be specified to apply the transformation in one of the scenarios enumerated when discussing incremental transformations in the previous section, such as in *QVT-R*.

Transformation units consist of **patterns**. A pattern is a model fragment that can be represented as: strings for template-based transformations (*e.g.*, *Xpand* [Pro10b]), terms for tree representations of models (*e.g.*, *Stratego*), or graphs for model-to-model transformations (*e.g.*, graph transformation). Patterns can be represented using the abstract or concrete syntax of the corresponding source or target model language. The syntax can be textual or graphical. There are two notations for representing patterns in graphical syntax: the traditional and the compact notation. The former is the traditional way of representing graph transformation rules with, on the left, the pre-condition pattern known as the left-hand side (LHS) and the post-condition pattern known as the right-hand side (RHS). For instance, Figure 2.3 illustrates a rule in *AToM*^{3} on the left and its equivalent in *FUJABA* [FNTZ00] on the right. In the *AToM*^{3} rule, the LHS pattern must first be found in the source model. Then the relation labelled 3 must be removed as it is present in the LHS but not in the RHS and the relation labelled 4 must be created as it is present in the RHS but not in the LHS. The corresponding relations in the compact notation are labelled "destroy" and "create" in the compact notation of *FUJABA*. Although the compact notation is more concise and prevents the modeller from replicating pattern elements in both patterns, the traditional notation is more expressive when it comes to specifying multiple

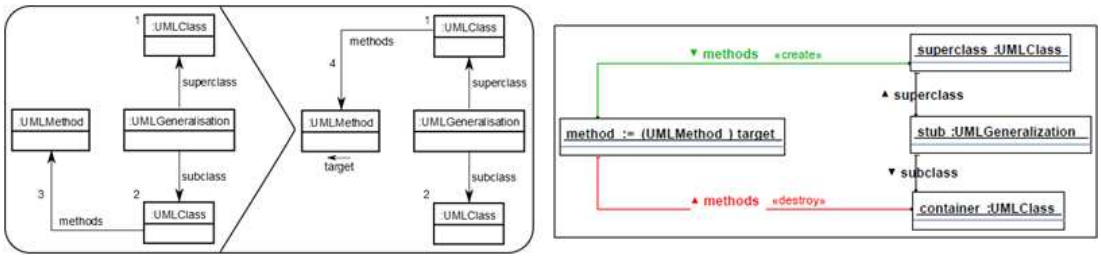


Figure 2.3: The same rule represented in *AToM*^{3} and in *FUJABA*.

```
top relation PackageToSchema {
    domain uml p:Package{name=pn}
    domain rdbms s:Schema{name=pn}
}
```

```
operation transform(source:PackageHierarchy): DataBase is do
    result := DataBase.new
    trace.initStep("uml2db")
    source.hierarchy.each{ pkg |
        var scm: Schema init Schema.new
        scm.name := String.clone(pkg.name)
        result.schema.add(scm)
        trace.addlink("uml2db", "package2schema", pkg, scm)
    }
end
```

Figure 2.4: A *QVT-R* relation on the left and the corresponding *Kermeta* function on the right.

negative-application conditions (NACs). A NAC represents a pattern that prohibits the application of the rule if encountered in the source model.

The computations and constraints over model elements in the patterns are regrouped in the **logic** of a transformation unit. The logic can be classified in two orthogonal dimensions: executable/non-executable and imperative/declarative. For example, the Java Application Programming Interface (API) for the Meta-Object Facility (MOF) [Obj06a] models is an executable, imperative logic. OCL queries [Obj06b] are executable and declarative. *QVT-R* relations are non-executable and declarative. For example, Figure 2.4 shows a *QVT-R* relation and the *Kermeta* function that corresponds to its forward check-enforce application. Note that an imperative, non-executable logic cannot exist since the imperative language will very likely have a virtual machine to execute it.

It is sometimes convenient for the modeller to specify a more **generic rule** that can be re-used. **Parametrization** of rules is supported by several transformation languages. For example in *PROGRES* [Zün94], a rule can take input/output parameters referred to by variables (such as the `diagnosis` variable in Figure 2.5). This is analogous to parameters of a function in a programming language. This allows one to bind some pattern elements to pre-defined source model elements. In *GReAT* [AKK+06], the notion of *pivot nodes* acts as parameter passing. For example, the `OrState` element is attached to the *In* icon depicting that that element was already bound to a previous rule application and will be used when executing the current rule. Parametrization reduces the size of transformations, in terms of number of rules and complexity.

Since rule-based transformation is the most common paradigm for model transformation, we will use the term rule in lieu of transformation unit for the remainder of this thesis.

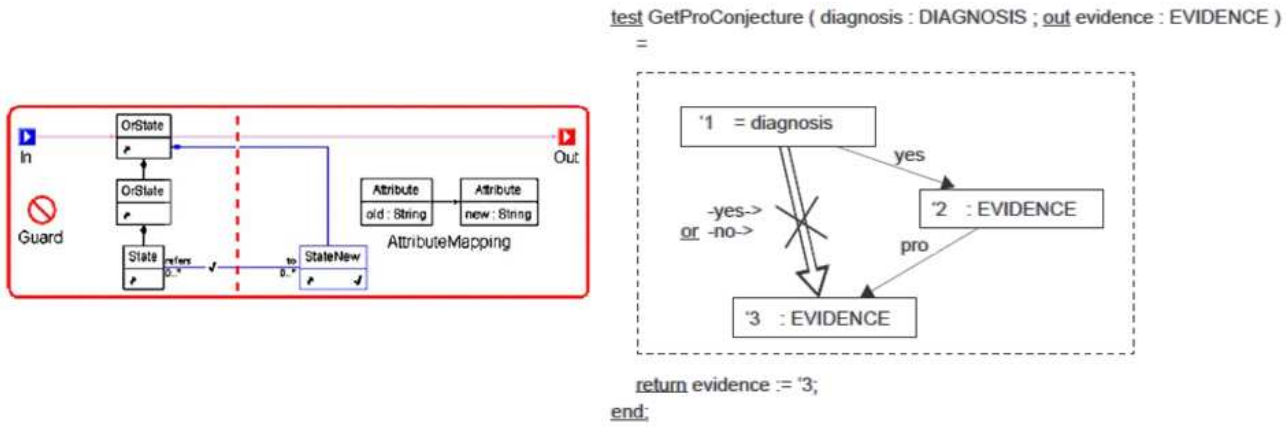


Figure 2.5: Pivot passing in *GReAT* (left) and parameter passing in *ProGReS* (right).

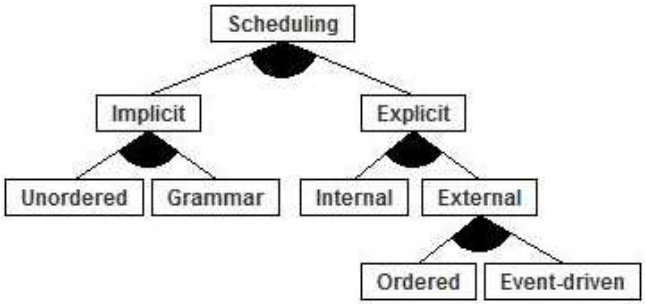


Figure 2.6: Feature diagram for rule scheduling.

2.1.3 Rule Scheduling

When designing the transformation units, it is mandatory to be aware of what kind of scheduling mechanism is used to apply the rules. Rule scheduling is part of inter-rule management. Scheduling can be achieved by explicit control structures or can be implicit due to the nature of the rule specifications. Moreover, several rules may be applicable at the same time. Similar selection mechanisms can be used as in the intra-rule case. To classify scheduling mechanisms, we combine Czarnecki and Helsen's classification with that of Blostein *et al.* [BFG96]. The feature diagram in Figure 2.6 reflects this updated classification.

One can distinguish between implicit and explicit scheduling. When the scheduling of a transformation language is **implicit**, the modeller has no direct control over the order in which the transformation units are applied. On the one hand, a transformation language can be **unordered**, *i.e.,* it simply consists of a set of rules. In this case, the order of application of the rules is entirely determined at run-time. It completely depends on the patterns specified in the rules. Applicable rules are selected non-deterministically until none apply anymore. For instance, *Groove* [Ren04] and *MOMENT2* [BÖ10] are graph transformation languages with unordered implicit scheduling.

Model transformation can also be used as a **grammar**. It is an unordered transformation (set of

rules) together with a start state and terminal states. For instance, graph grammars [EEKR97] are used for generating language elements: starting from a host graph (maybe empty), apply the rules to create graph elements until a termination state is found. A graph grammar also defines the operational semantics of a system modelled as a graph: it is the sequence of derivations from the initial graph to a final graph. Graph grammars are also used for language recognition: starting from the empty graph, find a sequence of rules that leads to the start graph. A characteristic of this approach is *state space exploration*. When no more rules are applicable because of the specific sequence of rule applications, a backtracking mechanism is used to revert the effects of applying the rules until the last non-deterministic choice was made. *DiaMeta* [Min06] uses graph grammars to define the meta-model of a language.

The scheduling of a language can be explicitly specified by the modeller. In **explicit internal** transformation languages, a rule may explicitly invoke other rules. For example in *ATL*, a matched rule (implicitly scheduled) may invoke a called rule in its imperative part. Also, a rule tagged as *lazy* will be applied only after all other rules have been applied. Another example of an explicit internal transformation language is *QVT-R*. There, the when/where clauses of a rule may have a reference to other rules: for when, the former will be applied after the latter and for where, the latter will be applied after the former.

Finally, in an **explicit external** transformation language, there is a clear separation between the rules and the scheduling logic. **Ordered** transformations specify a control mechanism that explicitly orders rule application of a set of rules. Examples are: priority-based, layered/phased, or with an explicit workflow structure. Most transformation languages are *partially ordered*, however. That is, applicable rules are chosen non-deterministically while following the control specification. Another sub-category of explicit external transformations is **event-driven** transformations, which have recently gained popularity. In these transformation systems, rule execution is triggered by external events such as in [GdL07b].

Since this thesis focuses more on the graph transformation approach, controlled (or programmed) graph rewriting is the key for scaling graph transformation to real-life industrial applications. Controlled graph transformation imposes a control structure over the transformation entities (transformation rules) to have a stricter ordering over the execution of a sequence of rules. This allows for more efficient implementations by providing search plans and pattern caching based on the given order. Initially proposed in [EEKR97] and later extended in [LLMC06] and then in [SV07], graph transformation control structure primitives may exhibit the following properties:

- *atomicity*: either all rules succeed or they all fail;
- *sequencing*: apply rules one after the other;
- *branching*: execution of a sub-structure based on a condition;
- *looping*: apply rules iteratively;
- *non-determinism*: non-deterministic ordering of rule application;
- *recursion*: ability of a control structure to call itself;
- *parallelism*: apply rules in parallel;

- *back-tracking*: explicit roll-back mechanism;
- *hierarchy*: in the sense of rule nesting.

In Table 2.1, we compare some of the relevant scalable graph transformation tools that exist today according to these properties (but is by no means an exhaustive list). A more detailed explanation is given in the following section.

Another comparative study of different graph transformation tools can be found in [TEG+05]. There, another set of tools is compared (though including *AToM*^{3} and *VMTS*) based on the solution they provide to a common case-study: the standard benchmark of Class Diagrams to Relational Database Models transformation [BRT05]. This case study will be covered in Chapter 10.

The very active field of graph transformation is not restricted to the tools appearing in Table 2.1. *DSLTrans* [BLA+10] is a layered graph transformation language. It allows one to design simple transformations that only produce a new model without affecting the original one. Unlike common model-to-model transformation languages (such as *ATL*), the transformation engineer has direct access to the tracing information that is automatically generated from previous rule applications.

GrGen.NET [GBG+06] is considered as the fastest graph transformation language [Zün08]. It is a textual language and regular expression annotations of rewrite rules provide sequencing, branching, looping, and non-determinism to the language. *GrGen.NET* does not provide an appropriate abstraction for a domain-specific modeller. Its target users however are software developers and it is meant to be used for the generation of the algorithmic core of applications processing graph structured data.

GROOVE [Ren04] offers a graph transformation language and enables the construction of a labelled transition system corresponding to all possible permutations of the rule applications. Graph patterns are combined with first-order logic predicates and thus allow for rule amalgamation (with universal and existential quantifiers). The main purpose of *GROOVE* is formal verification for graph based systems.

Henshin [ABJ+10] is the successor of *Tiger* [EETW06]. They are both integrated with the Eclipse Modelling Framework (EMF) allowing one to transform Ecore models. The execution engine relies on *AGG*, but is more expressive by adding sequence, priority, branching, and looping to schedule transformation units.

MOFLON [AKRS06] is a tool for designing triple graph grammars (TGGs) as described in Section 2.2.3. The execution engine is based on *FUJABA*'s. A *MOFLON* TGG rule is compiled to story diagram transformation rules in *FUJABA*. Instead of using a proprietary language for pattern specification, *MoTMoT* [MSVG05] (another graph transformation relying on *FUJABA*) is true standard-compliant by providing an adequate UML profile, but at the cost of defining story patterns as class diagrams.

MOLA [KBC05] merges traditional structured programming as a control structure with pattern-based transformation rules. The scheduler is a structured flowchart which allows graphical expression of statements such as rules, loops, branching, and recursive calls to sub-programs.

MOMENT2 [BÖ10] supports transformations based on rewriting logic implemented on top of the

constraint satisfaction solver Maude [CDE+07]. Thanks to its formalization based on rewrite logic, some static analysis and formal verification based on model checking are possible.

Kermeta [FHN06] is an example of a non-graph transformation language with an explicit control structure. In this language, transformations are not rule-based and do not have a formal foundation such as graph transformation. It is a textual language based on an action language which is imperative and object-oriented. *QVT-Operational Mappings* [Obj08] is another example of non-graph transformation languages with an explicit control structure. It will be described in detail in Section 2.2.4.

Property	AGG	ATOM^3	FUJABA	GREAT	PROGRES	VIATRA2	VMTS
Control Structure	Layered ordering	Priority ordering	Story diagram	Data flow	Imperative language	Abstract state machine	Activity diagram
Atomicity	Rule	Rule	Rule	Expression	`transaction`, rule	`gtrule`	`Step`
Sequencing	Implicit	Implicit	Yes	Yes	`&`	`seq`	Yes
Branching	No	No	Branch activity	`Test / Case`	`choose…else`	`if-then-else`	`Decision step`, OCL
Looping	Implicit	Implicit	For-all pattern	Yes	`loop`	`iterate`, `forall`	Self loop
Non-determinism	Within layer	Within priority layer	No	$1-n$ connection	`and`,`or`	`random`, or-pattern	No
Recursion	No	No	No	Yes	Yes	Yes	Yes
Parallelism	No	Optional	Optional	No	No	No	`Fork`, `Join`
Back-tracking	No	No	No	No	Implicit	`choose` (implicit)	No
Hierarchy	No	No	Nested state	`Block`, `ForBlock`	Modularisation	Pattern composition	`High level step`

Table 2.1: A comparison of the control structure of graph transformation tools.

2.2 Existing Transformation Languages and Approaches

Currently, there are over 30 model transformation approaches in the literature. Among these approaches, Czarnecki and Helsen distinguish between:

Visitor-based: A visitor pattern implemented in a programming language traverses the model in an object-oriented framework (*e.g.*, for pretty-printing a concrete syntax). This becomes programming rather than modelling.

Template-based: Templates are typically expressed in the concrete syntax of the target model, together with annotations of meta-code to access the source model. This approach is often used by code generators (*e.g.*, Enterprise Architect).

Direct-manipulation: Models offer an API to operate on them. The user has direct access to the API (described in the meta-meta-language) to manipulate models. This is still programming while being aware that they are models with a dedicated API.

Operational: They consist of modelled languages that allow manipulating models through, for example, declarative and/or imperative OCL. Also, meta-models are augmented with imperative constructs offering callable methods/functions in the models themselves.

Graph transformation-based: Models are represented as graphs, thus the theory of graph transformation is used to transform models. It is a declarative way of describing operations on models.

Relational: They declaratively describe mappings between source and target model, often in the form of constraints that need to be solved. They are implicitly multi-directional, but in-place transformation is harder to achieve.

Hybrid: It is a combination of two or more of the previous approaches.

Direct manipulation of models is the most used technique in software engineering. However, for the past decade, the modelling community has been promoting model manipulation techniques that are more structured, domain-specific, and declarative. A plethora of model transformation languages exist today; this section compares some of the most relevant ones. The focus is first directed to graph transformation approaches, then to relational approaches, and finally to other popular hybrid approaches. But first, let us examine the theory of graph transformation as it is the basis of the work in this thesis.

2.2.1 Foundations of Graph Transformation

Graphs are often used to model the state of a system. This allows graph transformation to model state changes of that system. Thus graph transformation systems can be applied in various fields. Graph transformation has its roots in classical approaches to rewriting, such as Chomsky grammars [Cho51] and term rewriting [BN99]. Operationally, a graph transformation from a graph G to a graph H follows these main steps. First, *choose* a rule composed of a LHS pattern and a RHS. Then, *find* an occurrence of the LHS in G satisfying the application conditions of the rule. Finally, *replace* the sub-graph

matched in G by the RHS. In fact, there are only four possible operations that a graph transformation rule can perform on the host graph: the so-called CRUD operations. Unless specified otherwise, graphs are considered typed, attributed, labelled, and directed.

There are different graph transformation approaches to apply these steps, as described in [EEKR97]. Among them is the *algebraic* approach, based on category theory with pushout constructs on the category of graphs. Algebraic graph transformation can be defined using either the *Single-Pushout* (SPO) or the *Double-Pushout* (DPO) approach. Since most of the tools adopt one or the other, we will outline DPO (the side-effect free approach) and discuss the differences with SPO.

Algebraic Graph Transformation

We consider the category **Graphs** [EEPT06] to present the major results. In this category, the *objects* are directed graphs[2] in the form $G = (V, E, s, t)$ where V is the set of vertices, E is the set of edges, and $s, t : E \to V$ are the source and target functions respectively. The *morphisms* are graph morphisms in the form of $f : G \to H = (f_V : V_G \to V_H, f_E : E_G \to E_H)$ where the mapping from (nodes and edges of) G_1 to (nodes and edges of) G_2 is total, *i.e.*, $\forall e \in E_G, f_V(s(e)) = s(f_E(e)) \wedge f_V(t(e)) = t(f_E(e))$. The *composition operator* and *identity morphism* of **Graphs** lead to component-wise graph morphisms on nodes and edges.

A *pushout* over morphisms $k : K \to D$ and $r : K \to R$ is defined by a pushout object H and morphisms $n : R \to H$ and $g : D \to H$ such that diagram (2) in Figure 2.7(a) commutes.

A *graph transformation rule* $p : (L \xleftarrow{l} K \xrightarrow{r} R)$, called *production*, is composed of a pair of injective total graph morphisms $l : L \leftarrow K$ and $r : K \to R$ where L, K, and R are respectively the LHS, interface, and RHS of p. In this case, K represents what sub-graph to preserve, being the common part of L and R. We can now formally define a graph transformation.

Let $p : (L \xleftarrow{l} K \xrightarrow{r} R)$ be a graph production, D a context graph (which includes K), and $m : L \to G$ a total graph morphism called *match*. Then a *DPO graph transformation* $G \overset{p,m}{\Rightarrow} H$ from G to H is given by the DPO diagram of Figure 2.7(a), where (1) and (2) are pushouts in the category **Graphs**. This is also known as *direct derivation*.

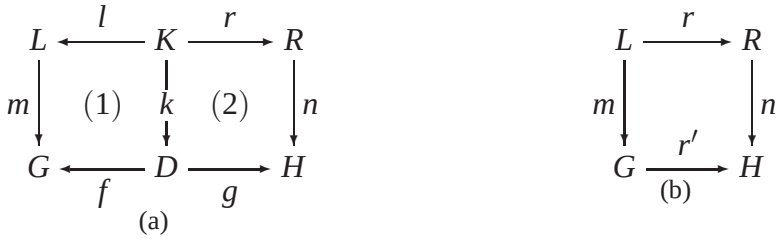


Figure 2.7: DPO (a) and SPO (b) constructions.

In the DPO approach, a transformation rule can thus be applied in the following steps:

[2] All the results of this section can be generalized to the category of typed, attributed, labelled, and directed graphs.

1. Find a match $M = m(L)$ in G.
2. Remove $L - K$ (the elements to be deleted) from M such that the *gluing condition* $(G - M) \cup k(K) = D$ still holds.
3. *Glue* $R - K$ (the elements to be created) to D in order to obtain H.

To build the context graph D, the gluing condition comprising the *identification condition* and the *dangling condition* has to be satisfied. In general, L does not need to be isomorphic to M. This is a problem since elements from L cannot be unambiguously identified. The identification condition requires that all elements in $m(R-L)$ (to be deleted) have only one pre-image in L. Another problematic situation is the presence of dangling edges, where the production deletes the source or the target of an edge outside the scope of the LHS. Therefore the dangling condition requires that when p specifies the deletion of a node, it should also delete all its incident edges.

Concurrency

Assume a graph transformation system with a set of productions $P = \left\{ p_i : (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i) \right\}$. Given a host graph G, the rule $p_i \in P$ is applicable if the context graph D_i of pushout (1) in Figure 2.7(a) exists (i.e., the gluing condition is satisfied). Suppose rules $p_1, p_2 \in P$ are both applicable. Applying p_1 and p_2 in a parallel system allows the two transformations to take place simultaneously. On a sequential system however, their atomic CRUD operations have to be interleaved arbitrarily. Meanwhile, under what conditions can p_1 and p_2 be applied concurrently?

To answer this question, the literature defines the notion of direct derivation independence (see [EEKR97]). Let $d_1 = (G \overset{p_1,m_1}{\Rightarrow} H_1)$ and $d_2 = (G \overset{p_2,m_2}{\Rightarrow} H_2)$ be two direct derivations. d_1 and d_2 are *parallel independent* if they do not conflict: p_2 can still be applied after the application of p_1 and vice-versa. Using DPO this can be formulated as $m_1(L_1) \cap m_2(L_2) \subseteq m_1(l_1(K_1)) \cap m_2(l_2(K_2))$. Therefore, neither of them can delete elements matched by the other. Thus, d_1 and d_2 may only overlap on the elements preserved by both derivations.

On the other hand, two consecutive direct derivations d_1 and d_2 are *sequential independent* if they are not causally dependent: applying first p_1 on G followed by p_2 or p_2 then p_1 leads to the same result. Using DPO this can be formulated as $n_1(R_1) \cap m_2(L_2) \subseteq n_1(r_1(K_1)) \cap m_2(l_2(K_2))$. Therefore, d_2 may not delete elements preserved by d_1 and cannot use any element created by d_1.

The conditions for interleaving d_1 and d_2 is formulated by the *Local Church-Rosser theorem*. Referring to Figure 2.8(a), the theorem states the following two conditions:

- If $G \overset{p_1,m_1}{\Rightarrow} H_1$ and $G \overset{p_2,m_2}{\Rightarrow} H_2$ are parallel independent, then there exists a graph X and two direct derivations $H_1 \overset{p_2,m_2'}{\Rightarrow} X$ and $H_2 \overset{p_1,m_1'}{\Rightarrow} X$ such that the pair $G \overset{p_1,m_1}{\Rightarrow} H_1 \overset{p_2,m_2'}{\Rightarrow} X$ and the pair $G \overset{p_2,m_2}{\Rightarrow} H_2 \overset{p_1,m_1'}{\Rightarrow} X$ are each sequential independent.
- If two direct derivations $G \overset{p_1,m_1}{\Rightarrow} H_1 \overset{p_2,m_2'}{\Rightarrow} X$ are sequential independent, then there exists a graph H_2 and two direct derivations $G \overset{p_2,m_2}{\Rightarrow} H_2 \overset{p_1,m_1'}{\Rightarrow} X$ such that $G \overset{p_1,m_1}{\Rightarrow} H_1$ and $G \overset{p_2,m_2}{\Rightarrow} H_2$ are parallel

independent.

The condition for parallelizing d_1 and d_2, is formulated by the *Parallelism theorem*. Referring to Figure 2.8(b), the theorem states that, given two productions p_1 and p_2, a parallel derivation $G \overset{p_1+p_2,m}{\Rightarrow} X$ exists if and only if there exists sequential independent direct derivations $G \overset{p_1,m_1}{\Rightarrow} H_1 \overset{p_2,m_2'}{\Rightarrow} X$.

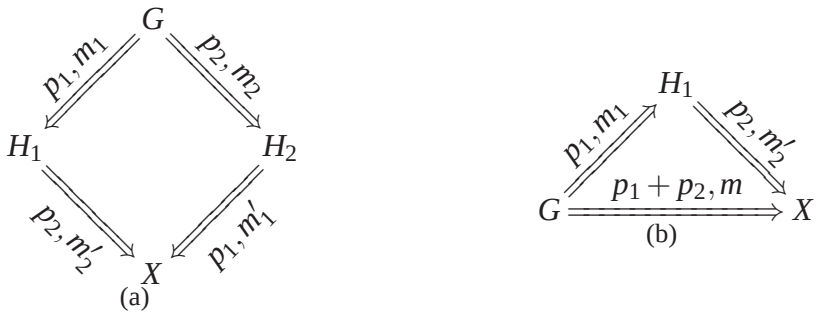


Figure 2.8: Derivations of (a) Local Church-Rosser and (b) Parallelism theorems.

These two theorems answer the question on the conditions for applying rules concurrently. Two graph transformations can be applied in an arbitrary order provided they are parallel independent. In this case, they can be applied in parallel via a parallel graph transformation. If two rules are parallel dependent they form a *critical pair*.

Algebraic Graph Transformation using SPO

In SPO, a production $r : L \to R$ is an injective partial graph morphism r. A partial graph morphism from a graph A to a graph B is a total graph morphism from a sub-graph of A to B. L and R denote respectively the LHS and RHS of p. Given the match $m : L \to G$ as a total graph morphism, a *SPO graph transformation* $G \overset{r,m}{\Rightarrow} H$ from G to H is given by the pushout diagram of Figure 2.7(b) in the category of graphs with partial morphisms.

The main difference between the SPO and the DPO approach is how they handle the identification and dangling problems. DPO prevents the rule application in both situations, whereas SPO implicitly deletes the problematic nodes. For this reason, DPO rules are invertible and SPO rules are not.

Similar concurrency properties hold for SPO. In fact, a SPO production can be translated to a DPO production defining K as a sub-graph of L. However, the reverse translation is not always possible.

Application Conditions

Graph transformation defines the transformation of models at some level of abstraction. The LHS is also called the positive application condition (PAC) since it determines the pattern to be *found* in the host model. Nevertheless, in lots of applications, it is often convenient to specify what pattern should *not* be found. This is referred to as negative application condition (NAC) [HHT96].

A graph transformation rule is then extended with the notion of *application condition*. A production with application condition $\hat{p} = (L \xrightarrow{p} R, A(p))$ consists of a graph transformation rule p and an application condition of $A(p)$. The authors of [HHT96] distinguish the PACs from the NACs in $A(p)$ as sets of total graph morphisms representing positive and negative constraints. In practice, the LHS implicitly contains the PACs, but NACs should be explicitly specified. The production $\hat{p}$ is said to be *applicable* if, given a match $m : L \to G$, m satisfies all positive and negative constraints of $A(p)$. Assume $A(p)$ only consists of a NAC with negative constraint $\bar{p} : L \to \bar{L}$. $\hat{p}$ is applicable if there is no total graph morphism $\bar{m} : \bar{L} \to G$ such that the composition $\bar{m} \circ \bar{p} = m$ holds as depicted in Figure 2.9.

Extensions of the DPO approach with NAC have been proposed in [HHT96] and parallel and sequential independence have been adapted accordingly.

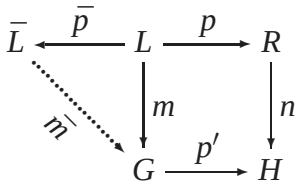


Figure 2.9: Application of a production with NAC.

Negative application conditions are very useful in transformation languages. A graph transformation rule consists of a LHS, a RHS and optionally a NAC. The LHS represents a pre-condition pattern to be found in the host graph along with conditions on attributes. The RHS represents the post-condition pattern after the rule has been applied on the matched sub-graph by the LHS. The NAC represents what pattern condition shall not be found in the host graph, inhibiting the application of the rule. NACs therefore increase the expressiveness of transformation rules. This makes the individual rules and the transformation process more understandable. Also, allowing negative expressions reduces the number of rules for a given transformation (often by a factor of two since, on top of the rules necessary for the transformation, additional rules must be specified to prevent the application of some of them). NACs may become very handy to prevent a sequence of derivations to process the graph. For example, when the transformation traverses the graph (or parts of it), making use of NACs can prevent infinite loops.

Hierarchical Graph Transformation

As mentioned previously, there are other approaches to graph transformation than the algebraic one. For example, in *hyperedge replacement* graph transformations, the LHS and RHS are hypergraphs[3]. The transformation is applied on a hyperedge, which is replaced by an arbitrary hypergraph with designated attachment nodes specified from the LHS.

Drewes *et al.* have extended the DPO approach for hierarchical graphs using hyperedge replacement transformations [DHP02]. This approach transforms hypergraphs where some hyperedges (called

[3] A graph where the edges may have multiple source and target vertices.

frames) can *contain* hypergraphs or variables. They denote transformations on these nested hypergraphs as hierarchical graph transformations. A hierarchical graph $G = \langle \hat{G}, F_G, cts_G \rangle$ consists of the graph $\hat{G}$ at the root of the hierarchy, the set of frames $F_G \subseteq E_G$ (subset of the hyperedges of G), and a content function cts_G assigning to each frame $f \in F_G$ either a (hierarchical) hypergraph or a variable (symbol). They extend the notion of graph morphism to hierarchical graph morphism from G to H with mappings on root graphs, frames, variables, and contents. The category **H** of hierarchical graphs without variables is thus defined. A pushout with injective hierarchical morphisms in **H** is defined in the same way as in **Graphs** with injective matches on $\hat{G}$ and on the hierarchical hypergraphs in F_G recursively. Because the match morphisms are injective, only the dangling condition is necessary to glue (the identification problem is handled like in SPO). In order to handle hierarchical graph transformation with variables, assignments on variables of the LHS are treated as a morphism satisfying the dangling condition.

The hierarchical graph transformation defined above does not completely abide by DPO because of the injective morphisms $L \leftarrow K$ and $K \rightarrow R$. As described in [DHP02], hierarchical graph transformation allows encapsulation of local transformation of graphs. This brings the graph transformation paradigm closer to a programming paradigm having the possibility of "storing" patterns inside variables.

In the *node replacement* approach [VJ04], hierarchy is added to graph transformation by letting nodes hold directed graphs.

A hierarchical graph is a graph where edges or nodes can contain other graphs nested in them. External edges from an inner graph to an ancestor (in the hierarchy tree) are not allowed. Hierarchical graph transformations are useful when the host model is very large: they allow *locality*. A single rule may be focused on parts of the graph. Rules thus gain in expressiveness in the sense that they allow transformations at different levels of the graph hierarchy (abstraction levels). For example, a modeller having designed a sub-model of the whole model can then specify a transformation only for the part he is interested in. From an implementation point of view, when hierarchical graphs are transformed by hierarchical graph transformation rules, the rule matching may be performed more efficiently. That is, the search space of the matching process can be drastically reduced, given that the rule is applied in a given context (the parent node).

2.2.2 Graph Transformation Languages

The theory of graph transformation is used as a basis for many model transformation tools. This subsection presents some of the most popular graph transformation languages[4] available today.

AGG

The Attributed Graph Grammar system (*AGG*) [Tae04] is still considered as the closest language implementing the theory of algebraic graph transformation. A type graph (the equivalent of a meta-

[4] In fact, they are controlled graph transformation languages where an explicit external scheduler organizes the individual graph transformation rules.

model in graph transformation) is specified by nodes and edges. Nodes are distinguishable by their type and may hold attributes[5].

AGG allows us to define typed attributed graph grammars. The graph rules composing a graph grammar support LHS, RHS, and NAC specification in terms of elements of the type graph. The application of an *AGG* graph rule follows the SPO approach with the option of allowing injective or non-injective matches. Graph rules can be organized in layers. That is, all rules in a layer will be applied as long as at least one of them finds a match before moving to the next layer. When no more rules in the last layer can match, the transformation terminates. It is also possible to restart from the first layer until no more rules across all layers can match anymore.

AToM^3

AToM^3 is a tool for meta-modelling, multi-formalism modelling, and model transformation [dLV02]. Model transformation can be performed on models conforming to a cross product of meta-models[6]. Since models are represented as abstract syntax graphs (ASGs), model transformation is performed through graph transformation. It was the first tool to provide a meta-modelling layer in graph transformations.

The control mechanism is limited to a priority-based transformation flow. The transformation system is a graph grammar consisting of graph transformation rules that can be assigned priorities. The rules are applied following the priority ordering: if a rule with higher priority fails, then the rule with the next lower priority is tried. If a rule succeeds, the transformation process starts back at the highest priority rule. These iterations go on until no more rules are applicable. When more than one rule with the same priority is applicable, one of them is chosen randomly, or the user chooses one interactively, or they are applied in parallel. For the latter option, *AToM*^{3} does not support conflict detection of overlapping rules. It is also possible to divide transformations into layers by sequencing graph grammars, without priorities.

ProGReS

The Programmed Graph Rewriting System (*ProGReS*) was the first fully implemented environment to allow programming through graph transformations [BS99, Zün94, SWZ95]. The control mechanism is a textual imperative language. A rule in *ProGReS* has a boolean behaviour indicating whether it succeeded or not. Among the imperative control structures it provides, rules can be conjuncted using the `&` operator. This allows for applying a sequence of rules in order. Branching is supported by the `choose` construct, which applies the first applicable rule following the specified order. *ProGReS* allows non-deterministic execution of transformation rules. `and` and `or` are the non-deterministic duals of `&` and `choose` respectively by selecting in a random order the rule to be applied. With the `loop` construct, it is possible to loop over sequences of (one or more) rules as long as it succeeds.

A sequence of rules can be encapsulated in a `transaction` following the usual atomicity, isola-

[5] In *AGG*, the type of the attributes can be any Java primitive or user-defined type.

[6] Cross meta-modelling is commonly referred to as multi-formalism modelling.

tion, durability, and consistency (ACID) properties. The underlying database system where the models are stored is responsible for ensuring the first three properties. An implicit back-tracking mechanism ensures consistency however. Hence, *ProGReS* offers two kinds of back-tracking: data back-tracking (with undo operations) and control flow back-tracking [Zün92]. When a rule r' fails in a sequence in the context of a transaction, the control flow will back-track to the previously applied rule r. The data back-tracking mechanism undoes the changes performed by the transformation of r. If r is applicable on another match, it applies the transformation on it and the process continues with the next rule (possibly r'). If r has no further matches, two cases arise. If r was chosen non-deterministically from a set of applicable rules, a non-previously applied rule is selected from this set. Otherwise, the process back-tracks recursively to the rule applied before r. Sequences and transactions can be named allowing recursive calls. The module concept provides a two-level hierarchy in the control flow structure by encapsulating a sequence of transactions.

FUJABA

Insights gained through the development of *ProGReS* have led to *FUJABA* (From UML to Java and Back Again) [NNZ00], a completely redesigned graph transformation environment based on Java and UML. *FUJABA*'s programmed graph rewriting system is based on Story Charts, a combination of Story Diagrams [FNTZ00] and Statecharts. An activity in such a diagram contains either graph rewrite rules, which adopt Collaboration Diagram-like representation, or pure Java code. The graph schemes for graph rewriting rules exploit UML class diagrams. With the expressiveness of Story Charts, graph transformation rules can be *sequenced* (using success and failure guards on the linking edges) along with activities containing code. *Branching* is ensured by the condition blocks which act like an if-else construct. An activity can be a *for-all* story pattern, which acts like a while loop on a transformation rule.

FUJABA's approach is implementation-oriented. Classes define method signatures and method content is described by Story Chart diagrams. All models are compiled to Java code. There is no notion of time.

GReAT

GReAT (for Graph Rewriting And Transformation language) is the model transformation language for the domain-specific modelling tool GME [AKK+06]. *GReAT*'s control structure language uses a proprietary asynchronous dataflow diagram notation where a production is represented by a "block" (called *Expression* in [AKK+06]). Expressions have input and output interfaces (*inports* and *outports*). They exchange packets: node binding information. The in-place transformation of the host graph thus requires only packets to flow through the transformation execution. Upon receiving a packet, if a match is found, the (new) packet will be sent to the output interface. Inport to outport connections depict sequencing of expressions in that order.

Two types of hierarchical rules are supported. A *Block* forwards all the incoming packets of its inport to the target(s) of that port connection (*i.e.*, the first inner expression(s) of the *Block*). On the other hand, a *ForBlock* sends one packet at a time to its first inner expression(s). When the *ForBlock*

has completely processed the packet, the next packet is sent iteratively. Branching is achieved using *Test* expressions. *Test* is a special composite expression holding *Case* expressions internally. A *Case* is given in the form of a rule with only a LHS and a boolean condition on attributes. An incoming packet is tested on each *Case* and every time the *Case* succeeds, it is sent to the corresponding outport. If a *Case* has its *cut* behaviour enabled, the input will not be tried with the subsequent *Cases*. When an outport is connected to more than one inport or if multiple *Cases* succeed in a *Test* (also one-to-many connection), the order of execution of the expressions that follow is non-deterministic. To achieve recursion, a composite expression (*Block*, *ForBlock*, or *Test/Case*) can have an internal connection to a parent or ancestor expression (in terms of the hierarchy tree).

VMTS

The controlled graph rewriting system of *VMTS* is provided by the VMTS Control Flow Language (*VCFL*) [LLMC06], a stereotyped UML Activity Diagram. In this abstract statemachine a transformation rule is encapsulated in an activity, called *step*. Sequencing is achieved by linking steps; self loops are allowed. Branching in *VCFL* is a *decision step* conditioned by an OCL expression. Chains of *steps* can thus be connected to the *decision*. However at most one of the branches may execute. The *steps* connected to the *decision* should then be non-overlapping (this is checked at compile-time). A branch can also be used to provide conditional loops and thus support iteration.

Steps can be nested in a *high level step*. A primitive step ends with success when the terminating state is reached and with failure when a match fails. However, in hierarchical steps, when a decision cannot be found at the level of primitive steps, the control flow is sent to the parent state or else the transformation fails. As in *GReAT*, recursive calls to *high level steps* is possible. A *fork* connected to a *step* allows for parallelism and a *join* synchronizes the parallel branches. Semantically, parallelism is possible in *VMTS* but it is not yet implemented [LLMC06].

VIATRA2

Transformations of the Visual Automated model Transformations framework (*VIATRA2*) are specified in the Viatra Textual Command Language (*VTCL*), incorporating graph transformation techniques driven by abstract state machines [VB07]. In *VTCL*, a rule has a pre-condition and a post-condition. These conditions are composed of *patterns* (similar to a LHS), *negative patterns* (similar to a NAC) or *OR-patterns* (allowing to specify a disjunction of patterns). In *VTCL*, rules can be parameterized by attribute values declared globally, or model elements bound to the application of a rule (this is similar to pivots in *GReAT*). They can be applied in sequence using the `seq` keyword. One rule can be applied non-deterministically from a set of rules using the `random` construct. `If-then-else` is used for branching. `Try A else B` attempts to apply rule `A` and, if no matches were found, then rule `B` is applied. `Iterate` applies a rule as long as possible, whereas `forall` first finds all matches and then rewrites them one by one. In *VIATRA2*, the notion of rule hierarchy is obtained by composing sub-patterns into more complex ones (such as `or`-patterns). This way, patterns can be re-used in multiple rules. A rule may be called recursively as long as it does not involve a negative pattern.

2.2.3 Graph-based Model-To-Model Relations

The previous subsections dealt with graph transformations as "operations": given a host model and a set of transformation rules, produce a *new* model by applying the rules on the host model. In the following two subsections, graph transformation is raised to the abstraction level of relations: given two models, specify a relation between them. Because they do not specify any causality, these model-to-model transformations (or relations) are inherently bidirectional. Thus, in a single specification, they combine source-to-target and target-to-source transformations. Such declarative graph transformations can then be used for model (co-)evolution, model synchronization, and incremental change propagation between the two models.

Triple Graph Grammars

Originally, Triple Graph Grammars (TGGs) were inspired by Pair Graph Grammars (PGGs). In 1971, Pratt proposed PGGs [Pra71] to examine string-to-graph translations as a one-to-one context-free mapping. In 1994, Schürr introduced TGGs as graph-to-graph translations and data integration [Sch94]. In contrast with an operational graph grammar, a TGG is not intended to model the editing processes on related graphs (by inserting, deleting, or modifying graph elements), but rather provides a generative description of graph languages and their relationships. A TGG consists of context-sensitive triple productions allowing complex LHS and RHS graphs, as well as a separate correspondence graph for modelling many-to-many relationships. In addition, the correspondence links between the correspondence graph and the LHS and RHS play the role of traceability links that map elements of one graph to elements of the other graph and vice-versa. Furthermore, each correspondence link may carry additional information about the transformation itself.

In a TGG, the graph transformation rules are monotonic: they are non-deleting rules. Following the notation in the commuting diagram of Figure 2.10(a), a monotonic graph transformation rule $p : L \to R$ is a graph transformation rule such that $L \subseteq R$ and n consists of all the mappings of m as well as additional mappings restricted from $R - L$ to $H - G$ only. A *monotonic production* is then given by the pushout of Figure 2.10(a) in **Graphs**.

TGGs act on *graph triples* of the form $(G_L \xleftarrow{l_G} G_C \xrightarrow{r_G} G_R)$ where L_G, R_G, and C_G are the LHS, RHS, and correspondence graphs respectively. l_G and r_G are graph morphisms allowing m-to-n relationships between L_G and R_G such that every pair of related elements in a subset of $G_L \times G_R$ has a pre-image in G_C.

A *production triple* $p = (p_l \xleftarrow{l} p_c \xrightarrow{r} p_r)$ consists of three monotonic productions: $p_l : (L_L \to R_L)$, $p_r : (L_R \to R_R)$, and $p_c : (L_C \to R_C)$ (the left, right, and correspondence pushouts in perspective in Figure 2.10(b)). Furthermore, $l : (R_C \to R_L)$ and $r : (R_C \to R_R)$ are graph morphisms such that the two diagrams at the back of Figure 2.10(b) are pushouts in **Graphs**. A TGG production is therefore a graph partitioned into three (left, right, and correspondence) graphs. Viewed from another angle, a TGG production contains three graph productions: one operates on a left graph, one on the right graph and one on a correspondence graph. It is this combination of three graph rewriting rules which has to be applied simultaneously that we call "triple graph grammar rule".

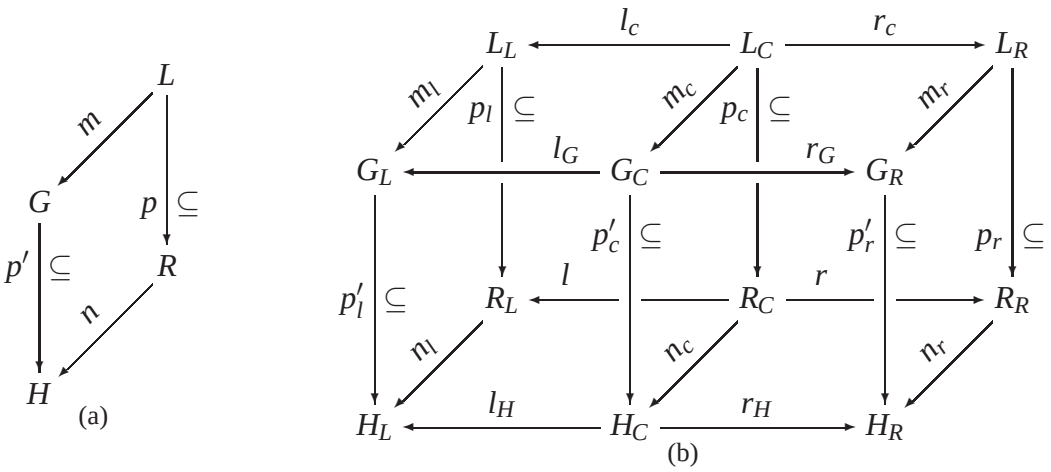


Figure 2.10: (a) A monotonic production and (b) a TGG production applied on a triple graph. The pushouts should not be confused with the SPO and DPO notation of Section 2.2.1.

Moreover, since a TGG rule is context-sensitive, it is applied on an *axiom* graph triple $G = (G_L \xleftarrow{l_G} G_C \xrightarrow{r_G} G_R)$ and produce the result graph $H = (H_L \xleftarrow{l_H} H_C \xrightarrow{r_H} H_R)$, as depicted in Figure 2.10(b).

The presented TGG rules above define a declarative bidirectional transformation from a left graph to a right graph. In a model-driven engineering context, a TGG rule defines a bidirectional relation between two meta-models. The operational semantics of a TGG rule (how the transformation is performed) is described by three kinds of *operational graph transformation rules* (in the sense of Section 2.2.1) where the LHS and the RHS are triple graphs: creation, deletion, and consistency rules. The former ensures that every new element of one model has a correspondence in the other model. The second makes sure that when an element is removed from one model, its corresponding element(s) is (are) deleted appropriately. The latter enforces the consistency relation between two elements (at the attribute level) by updating the corresponding element. Forward and backwards versions of these rules are generated with an additional traceability rule that creates the correspondence link between unmapped consistent elements of the two models. The (semi-)automatic derivation of some of these "lower-level" transformations is given in [KS06b]. In total seven operational graph transformation rules are generated for each model element, for every TGG rule in the grammar. The operational rules are then given priorities to ensure correct application.

TGGs had a great impact in the graph transformation community allowing declarative and bidirectional graph transformations. For example, in [GdL07b] the authors have extended TGGs to handle meta-models with inheritance and parameterized by *events*. Event-driven grammars have been introduced in the context of expressing user interface behaviour. Formalized in the DPO approach, a TGG was used to relate a model's concrete syntax (its visual representation to the user) to its abstract syntax (the graph model). Triple graph grammars have also been extended to multi-graph grammars [KS06a] where an arbitrary number of models can be related. MOFLON [AKRS06] is the main model transformation tool that supports TGGs.

Pattern-based Transformation

As recently pointed out in [SK08], TGGs do not support NACs. Also, it is not clear how arbitrary attribute manipulation is handled in TGGs. Furthermore, a TGG is context-sensitive and assumes an axiom triple graph as context. This induces causality between rules, thus TGG rules are not *purely* declarative. For these reasons, pattern-based transformation (PBT) was proposed [dLG08]. PBT is highly inspired by TGG, but their intentions differ.

In PBT, a (model-to-model) specification is a conjunction of *triple patterns* acting as constraints over triples graphs (equivalent concept to graph triples in a TGG). A triple graph *TrG* relates the two models (graphs) by an intermediate correspondence graph. A triple pattern defines a constraint on the graph triple by specifying positive and negative information (similar to PAC and NAC). There are three types of patterns. The simple pattern (S-Pattern) consists of a negative pre-condition $\overleftarrow{N}$, a positive graph *Q*, and a negative post-condition $\overrightarrow{N}$. An S-Pattern thus states that *Q* should be found in *TrG* whenever $\overleftarrow{N}$ is not; and once *Q* is found, $\overrightarrow{N}$ should not occur in *TrG*. The composite pattern (C-Pattern) is an S-Pattern with an additional positive pre-condition $\overleftarrow{P}$. The negative pattern (N-Pattern) simply consists of a negative post-condition.

Given a specification, the patterns are compiled into operational TGG rules. The compilation process of the patterns is divided into two phases. First, *deduction rules* are produced. This generates new patterns which take inter-pattern dependencies into account. The N-Patterns are transformed into post-conditions for the S- and C-patterns. The S- and C-patterns are enriched with further pre- and post-conditions according to their dependencies. From there, forward and backwards operational TGG rules are derived.

There are some limitations of this approach: the derivation process does not allow patterns with both positive and negative post-conditions. Although different from TGG, PBT does not handle complex attribute relationship either. No practical application implements PBTs yet. Bidirectional relational transformation is a very active topic of research in the graph transformation community. Other non-graph-based declarative model-to-model transformation approaches exist, such as: QVT-relational [Obj08] and Tefkat [LS06].

2.2.4 Hybrid Model Transformation Approaches

Model transformation approaches are not restricted to graph transformation. First we describe how relational database systems can resolve model transformation using concepts similar to that of graph transformation. Then we describe a hybrid approach (mixing declarative and imperative aspects) provided by one of today's most used model transformation tool. Finally, we elaborate the transformation language proposed by the OMG as a standard.

Model Transformation in Relational Databases

Graph transformation as described in the previous sections is performed in memory. This approach scales up to some point as long as both models and transformation process fit in memory. However,

for very large models (of the order of 10^6 or more elements) it is preferable to store them in a database. For that reason, Varró *et al.* propose in [VFV06] a model transformation approach performed in a relational database management system (RDBMS). Once models are stored appropriately in an RDBMS, the transformation specification consists of views and query statements.

Here, we assume that meta-models are initially specified in a subset of UML class diagrams and models in UML Communication diagrams. The transformation, however, requires the models to be represented in a RDBMS in the following way. From the meta-model, one table per class is generated with a column for a unique identifier. Additionally, one column is created per attribute and per many-to-one association. Many-to-many associations are represented as tables on their own with a column for the source and another for the target. Foreign keys ensure the constraint dependencies for association ends and inheritance. Models are stored as rows filling these tables.

The transformation rules follow the SPO graph transformation approach. A rule is divided into two parts: the *matching phase* and the *modification phase*. For the matching phase, the pre-condition LHS $\uplus$ NAC (weaving overlapping elements) of the rule is considered. The LHS is stored as a single view, *LHS-view*, in the RDBMS. An inner join is added for every object (node) and every association instance (edge) in the LHS. They are filtered according to the edge constraints of the structure of the pattern. Additional filters are used for specifying the exact matching conditions (total injective graph morphism). Finally, the selection projects only the joined columns. Similarly, *NAC-views* are created for each NAC pattern of the rule. LHS $\uplus$ NAC is stored as a separate view. A left outer join of each NAC-view is performed on the LHS-view and the join condition depicts the overlapping elements. To prevent the NAC from being positively matched, the filters of the view force a null value on the columns of the join conditions. Finally, the selected columns are those of the LHS-view.

The modification phase of a transformation rule is encapsulated in a transaction consisting of a sequence of INSERT, DELETE, and UPDATE statements. This phase starts by deleting edges if LHS $-$ RHS $\neq \emptyset$. An UPDATE statement removes the foreign key of the source of a many-to-one association. A DELETE statement removes a many-to-many association as well as any node. Additional DELETE and UPDATE statements are required to ensure the deletion of dangling edges. Then insertions come into place if RHS $-$ LHS $\neq \emptyset$. An INSERT statement creates a many-to-many association as well as a new node object. An UPDATE statement creates a many-to-one association. In the RDBMS approach, a model element can have an attribute as a one-to-one association between them. This is why there is no UPDATE statement that modifies the value of an attribute.

An advantage of this approach is that a single rule may be applied in parallel on all its matches. This is achieved by applying the modification phase on all the rows returned by the pre-condition view of the rule. Both matching and modification phases can be optimized with the underlying database system used. For example, to perform SPO-like deletion, it may suffice to allow cascading deletes on associations, if they are represented accordingly in the database. Although applying a transformation in a RDBMS is less efficient than in memory, an optimization in time can be gained by properly creating indices on columns where a matching occurs.

ATL

The ATLAS Transformation Language (ATL) is a hybrid model transformation language combining declarative and imperative constructs [JK06, Jou06]. It is a programming language with its own compiler and virtual machine. An ATL transformation is defined from (possibly several) read-only source meta-models to one write-only target meta-model.

The transformation specification consists of a set of rules and possibly helpers and external modules. The helpers are similar to OCL helpers: they serve as wrappers in the context of source models elements (since the target model is not navigable). *Operation helpers*, taking input parameters, act as functions. *Attribute helpers* decorate the source model by enriching it with a derived subset of its structure.

A declarative rule is called a *matched rule*, since it is transparent from the internal matching and scheduling algorithms of ATL. A matched rule is composed of a source and a target pattern. The source pattern specifies a set of pairs (t,g) where t is a type from the source meta-model and g is an OCL boolean guard. The target pattern is a set of pairs (t',b) where t' is a type from the target meta-model and b is a binding initializing the attributes or references of t'. (t',b) can be replaced by an *action block* where ATL imperative statements are used to build the target model elements. A rule may refer to other rules. *Standard* rules are applied once for every match, *lazy* rules are applied as many times as they are referred to, and *unique lazy* rules are lazy rules but re-use the target elements they created when applied multiple times. Declarative rules support inheritance as means of re-use and polymorphism. A subrule may only match a subset of the match of its parent, but can extend the creation of target elements. A *called rule* is an imperative procedure which can be invoked from a rule (matched or called) and is implemented either using the ATL imperative language constructs or any other language (but the latter has limited support).

Although declarative rules resemble graph transformation rules with a LHS and a RHS, the procedural semantics of an ATL transformation is quite different from the execution of a graph transformation system on a source model. The transformation starts with a first pass through all the guards to evaluate the helpers. The transformation is executed in the second pass. First, a called rule marked as *entry point* is applied if present, which may trigger subsequent rule applications. Then all the matches from all the standard matched rules are computed. Afterwards, for every match, the target elements are created without evaluating the bindings. At the same time, a traceability link between the rule, its matched source elements, and the new target elements is established internally. Secondly, all initializations (including bindings) are resolved following the *ATL resolve algorithm*. If referenced, lazy rules are applied too. Then action blocks evaluations follow. The algorithm ends by invoking the called rule marked as *end point*, if present. The order of execution of the standard rules is non-deterministic. Nevertheless, determinism and termination of the algorithm is ensured, provided that no lazy or called rules are used.

The Eclipse Modelling Framework (EMF) has adopted ATL as its language and tool support for model transformation. However, ATL lacks a formal foundation, unlike graph-based transformation.

QVT

The Meta-Object Facility (MOF) 2.0 Query, View, and Transformation (*QVT*) framework [Obj08] is a recent addition to the OMG's set of standards. The *QVT* specification defines three transformation languages that collectively form a hybrid transformation language. *QVT-Relations* (*QVT-R*) and *QVT-Core* (*QVT-C*) are declarative transformation languages at different levels of abstraction. *QVT-Operational Mappings* (*QVT-OM*) is an imperative transformation language that extends both *QVT-R* and *QVT-C*. *Black-Box* implementations are also imperative extensions of *QVT-R* and *QVT-C* allowing one to plug-in external code.

Transformations specified in *QVT-R* consist of declarations of *relations* specifying constraints that must be satisfied by the input models. A relation consists of at least two domains as well as pre- and post-conditions. A domain specifies the type (meta-model) of the involved model instance along with a pattern defining the template that a model must satisfy. A `when` clause specifies a pre-condition required by the relation. A `where` clause specifies a post-condition that must hold if the current relation holds. Both clauses may refer to other relations. There are two kinds of relations in *QVT-R*. All *top-level* relations are required to hold after the execution of the transformation, while non-top-level relations must only hold if invoked directly or transitively by a `where` clause. The execution of a *QVT-R* transformation follows the *check-enforce* semantics. On the one hand, if a transformation is executed in the direction of a `checkonly` domain, then the transformation simply checks whether there is a valid match in the target model that satisfies all relationships. On the other hand, if a transformation is executed in the direction of an `enforce` domain, then the transformation checks whether the relations holds. When a relation fails to find a valid match, the appropriate model elements are created, deleted, or modified according to the pattern of the target domain. Hence, when only one domain is `enforced` and the others `checkonly`, the transformation is uni-directional. When at least two domains are `enforced`, the transformation is multi-directional. Finally, when all domains are `checkonly`, the transformation is a synchronization verifying if the models are consistent with respect to the relations. Additionally, declaring a meta-model element as `key` ensures that the *QVT-R* transformation does not create duplicate elements if they already exist. Patterns can be matched against existing model elements, instantiated to model elements in new models, and may be used to apply changes to existing models. Nevertheless, the language handles the manipulation of traceability links automatically and hides the related details from the developer. Furthermore, scheduling of relations is implicitly determined by the `when` and `where` clauses.

Transformations specified in *QVT-C* consist of declarations of *mappings*. A mapping supports pattern matching over a flat set of variables by evaluating conditions over those variables against a set of input models. In contrast with *QVT-R*, a mapping defines explicit traceability links between domains. Similar *checking* and *enforcement* semantics are also available in *QVT-C*. Every *QVT-R* transformation can be translated to a semantically equivalent *QVT-C* transformation.

Transformations specified in *QVT-OM* are similar to classes, in that they can be instantiated. They consist of unidirectional *mapping operations* specified with imperative constructs. A mapping operation is a standard UML operation that, given one or more source model elements, specifies how target model elements are constructed. A mapping operation may specify pre- and post-conditions in `when`

and `where` clauses respectively. The body of an operation is expressed by an imperative extension of OCL. It consists of optional initialization (`init`) and termination (`finalize`) sections. They respectively allow computations prior to and after the instantiation of the outputs. The `population` section specifies how the output objects must be constructed. The target object output by a mapping operation is referred to by the `result` variable. A *QVT-OM* transformation starts executing from the `main` operation. Mapping operations may be invoked at any time just like standard operations using the `map` keyword. Branching is ensured by `if-elif-else` constructs. Executing operations in loops is provided by `forEach` and `while` statements. When a mapping operation *inherits* from another one, the initialization section of the latter is invoked after the former's. Sequencing operations is done by the `merge` of mapping operations. `Disjunction` of mapping operations allows the execution of at most one operation whose `when` pre-condition is satisfied. Moreover, `parallelTransform` and `wait` allow running operations in parallel. In *QVT-OM*, mappings between source and target model elements are implicitly created like in *QVT-R*. `ResolveIn` allows one to refer to the source element mapped on the created object. Also, lazy instantiation of output objects is possible using the `late resolve` construct.

The black-box mechanism allows complex algorithms to be implemented in any programming language and enables re-use of already existing libraries. This makes some parts of the transformation opaque, which brings a potential danger since their functionality is arbitrary and is not controlled by the transformation engine.

2.3 Conclusion

This chapter concludes the survey of model transformation that began in Chapter 1. This survey established that model transformation has various applications, such as: to generate platform-specific models from platform-independent models and reverse engineer them, to map and synchronize among models at the same or across abstraction levels, to create query-based views of a system, to model evolution tasks, or to transform models between different languages for integration. After a high level overview of the different types and uses of model transformation, the survey elaborated on the features that contemporary model transformation languages offer to attempt solving these problems. Focusing on first controlled graph transformation languages and then on others, the survey compared twelve of the most relevant languages that are used today.

An interesting interpretation of the feature model presented by Czarnecki and Helsen is that every vaild combination of the leaves of the feature diagram leads to a specific transformation language equipped with these features. This allows us to talk about a *family of model transformation languages*.

Part II

A Basis for Model Transformation

“Diversity is the one true thing we all have in common. Celebrate it every day.”

Winston Churchill

3

A Minimal Transformation Core

Since the applications are very different in nature, it is not optimal to have a single model transformation language to perform all the tasks previously enumerated. Instead, it is more appropriate to have dedicated transformation languages tailored to specific transformation problems.

The diversity of today's model transformation languages makes it hard to compare their expressiveness and provide a framework for interoperability. De-constructing and then re-constructing model transformation languages by means of a unique set of most primitive constructs facilitates both. Thus this chapter introduces *T-Core*, a collection of primitives for model transformation. Combining *T-Core* with a (programming or modelling) language enables the design of model transformation formalisms. We show how basic and more advanced features from existing model transformation languages can be re-constructed using *T-Core* primitives.

3.1 Introduction

A plethora of different rule-based model transformation languages and supporting tools exists today. They cover all (or a subset of) the well-known essential features of model transformation [SV09]: *atomicity*, *sequencing*, *branching*, *looping*, *non-determinism*, *recursion*, *parallelism*, *back-tracking*, *hierarchy*, and *time*. For such languages, the semantics (and hence implementation) of a transformation rule consists of the appropriate combination of building blocks implementing primitive operations such as matching, rewriting, and often a validation of consistent application of the rule. The above-mentioned essential features of transformation languages are achieved by implicitly or explicitly specifying "rule scheduling". Languages such as *ATL* [JK06], *FUJABA* [FNTZ00], *GReAT* [AKK+06], *MoTif* [SV10], *VIATRA* [VB07], and *VMTS* [LLMC06] include constructs to specify the order in which rules are applied. This often takes the form of a control flow language. Without loss of generality, we consider transformation languages where models are encoded as typed, attributed graphs.

The diversity of transformation languages makes it hard, on the one hand, to compare their expressiveness and, on the other hand, to provide a framework for interoperability (*i.e.*, meaningfully combining transformation units specified in different transformation languages). One approach is to express model transformation at the level of primitive building blocks. De-constructing and then re-constructing model transformation languages by means of a small set of most primitive constructs offers a common basis to compare the expressiveness of transformation languages. It may also help

in the discovery of novel, possibly domain-specific, model transformation constructs by combining the building blocks in new ways. Furthermore, it allows implementers to focus on maximizing the efficiency of the primitives in isolation, leading to more efficient transformations overall. Lastly, once re-constructed, different transformation languages can seamlessly interoperate as they are built on the same primitives. This use of common primitives in turn allows for global as well as inter-rule optimization.

In this chapter we introduce *T-Core*, a collection of transformation language primitives for model transformation in Section 3.2. Section 3.3 motivates the choice of its primitives. Then, Section 3.4 shows how transformation entities, common as well as more esoteric, can be re-constructed. Finally, Section 3.6 describes related work.

3.2 De-constructing Transformation Languages

Model transformation language primitives can be defined at different levels of granularity. The decomposition process is similar to what is found in object-oriented languages as depicted in Table 3.1. At the highest level, the transformation can be decomposed into sub-transformations[1], each dedicated to a specific task in order to accomplish a single goal (simulation, code generation, synchronization, etc). Following the analogy, a transformation corresponds to a package in object-oriented languages. Defining model transformation language primitives at this level means that transformations are treated as black-boxes, which is not the intention. Thus at a lower level, a (sub-)transformation can be decomposed into individual rules. Rules are the units of a transformation like a class is to a package. However, setting the rules as primitives would not consider other model transformation paradigms such as relational or functional. At a coarser level of abstraction, a transformation encapsulates CRUD operations performed on a model. However, we believe that these operations should be defined at the virtual machine level, rather than having a transformation language engineer combine them, *i.e.*, this is not the optimal level of abstraction. Hence rule primitives reside somewhere between rule definitions and CRUD operations. They dictate how a rule operates. As methods define the behaviour of a class and operations on objects, rule primitives define the behaviour of a rule operating on the model. At a more fine-grained level, a rule primitive encapsulates CRUD operations performed on the model. This is similar to how methods encapsulate operations that can be performed on variables (assignment, navigation, iteration, etc).

The proposed decomposition of model transformation languages therefore focuses on the rule primitives level. After the comparison of the features of model transformation languages in Chapter 2, one can synthesize the common essential features of model transformation as follows:

Pre- and post-condition patterns that allow one to declaratively specify a rule;

Matching rule pre-condition patterns in the host model to bind model elements (a match) that will be modified by the application of a rule;

[1] A sub-transformation can be considered as a transformation on its own. But when designed modularly, composing these transformations can lead to a more complex transformation.

Model Transformation Paradigm	Object-Oriented Paradigm
Transformation	Package
Rule	Class
Rule Primitive	Method
CRUD operation	Operation on variables

Table 3.1: Analogy of the abstraction hierarchy in model transformation and object-oriented paradigms.

Rewriting the host model to satisfy the post-condition of a rule;

Validation of consistent rule applications to detect conflicts and resolve them;

Manipulation of matches to **iterate** through them and **roll-back** to previous match states;

Control of the flow of rule applications by offering **choices** and **concurrency**;

Composition mechanisms to provide structure, re-use, and encapsulation.

Based on the previous observations, we propose here a collection of model transformation primitives. The class diagram in Figure 3.1 presents the module *T-Core* (which stands for *Transformation Core*) encapsulating model transformation primitives. *T-Core* consists of eight primitive constructs (Primitive objects): a Matcher, Iterator, Rewriter, Resolver, Rollbacker, Composer, Selector, and Synchronizer. The first five are RulePrimitive elements and represent the building blocks of a single transformation unit. *T-Core* is not restricted to any form of specification of a transformation unit. In fact, we consider only PreConditionPatterns and PostConditionPatterns. For example, in rule-based model transformation, the transformation unit is a *rule*. The PreConditionPattern determines its applicability: it is usually described with a LHS and optional NACs. It also consists of a PostConditionPattern which imposes a pattern to be found after the rule was applied: it is usually described with a RHS. RulePrimitives are to be distinguished from the ControlPrimitives, which are used in the design of the rule scheduling part of the transformation language. A meaningful composition of all these different constructs in a Composer object allows modular encapsulation of and communication between Primitive objects.

Primitives exchange three different types of messages: Packet, Cancel, and Exception. A packet π represents the host model together with sufficient information for inter- and intra-rule processing of the matches. π thus holds the current model (graph in our case) `graph`, the `matchSet`, and a reference to the `current` PreConditionPattern identifying a MatchSet. A MatchSet refers to a `condition` pattern and contains the actual matches as well as a reference to the `matchToRewrite`. Note that each MatchSet of a packet has a unique condition, used for identifying the set of `matches`. A Match consists of a sub-graph of the `graph` in π where each element is bound to an element in `graph`. Some elements (Nodes) of the match may be labelled as `pivots`, which allows certain elements of the model to be identified and passed between rules. A cancel message φ is meant to cancel the activity of

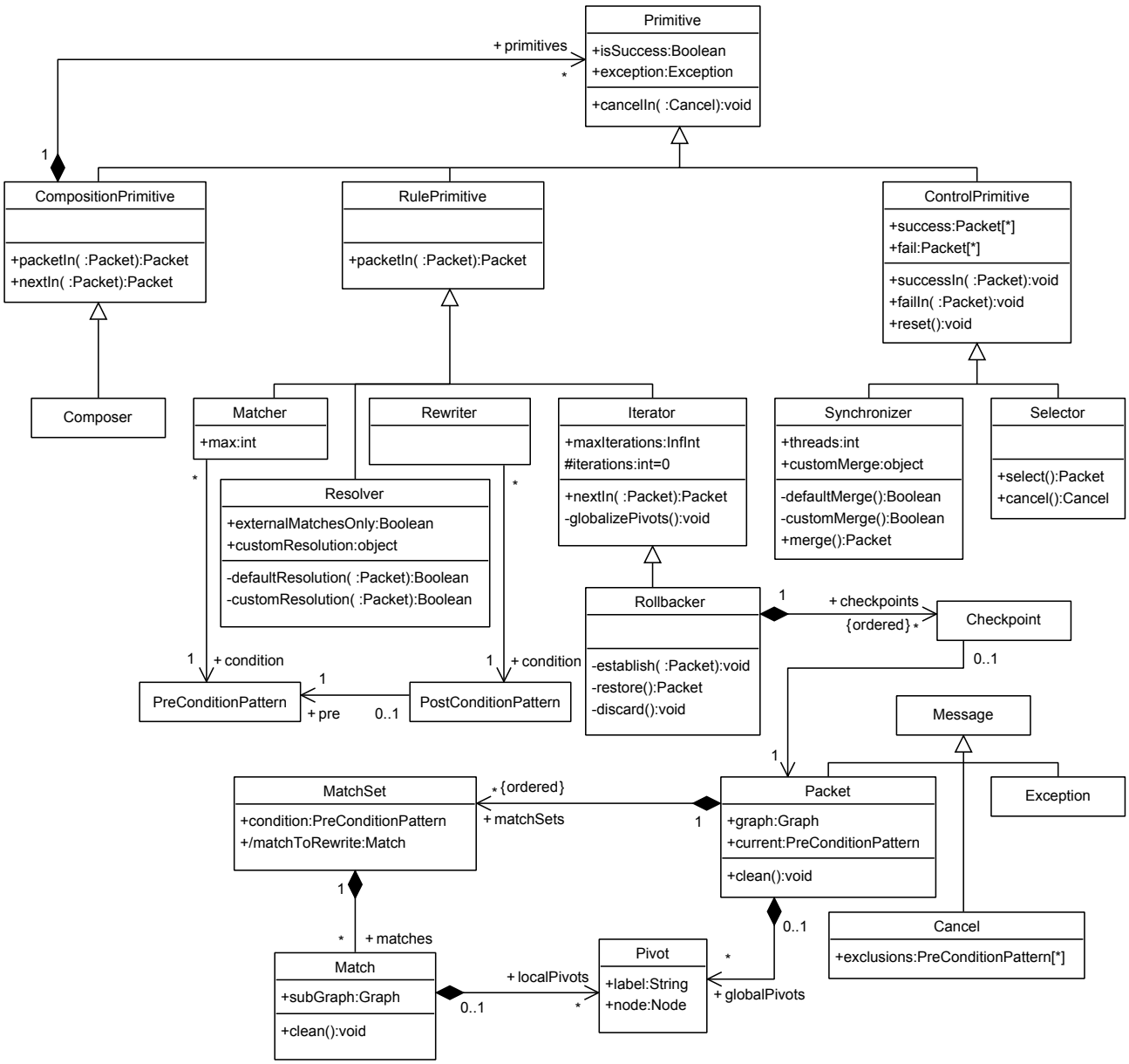


Figure 3.1: The *T-Core* module.

an active primitive element (especially used in the presence of a Selector). Finally, specific exceptions χ can be explicitly raised, carrying along the currently processed packet π.

All the primitive constructs can receive packets by invoking either their `packetIn`, `nextIn`, `successIn`, or `failIn` methods. The result of calling one of these methods sets the primitive in success or failure mode as recorded by the `isSuccess` attribute. Cancel messages are received from the `cancelIn` method. Next, we describe in detail the behaviour of the different methods supported by each of the eight primitive elements.

3.2.1 Matcher

Algorithm 1 Matcher.*packetIn*(π)

$M \leftarrow$ (max) matches of *condition* found in π.*graph*
if $\exists \langle$*condition*$, M' \rangle \in \pi$.*matchSets* **then**
  $M' \leftarrow M' \cup M$
else
  add $\langle$*condition*$, M \rangle$ to π.*matchSets*
end if
π.*current* $\leftarrow$ *condition*
isSuccess $\leftarrow M \neq \emptyset$
return π

The Matcher looks for an occurrence of its pre-condition pattern *condition* in the graph of the input packet π. The transformation modeller may optimize the matching by setting the *max* attribute to finding one, all, or a maximum number of matches when he knows a priori that this many matches of the matcher will be processed in the overall transformation. The matching also considers the pivot mapping[2] (if present) of the current match of π. After matching the graph, the Matcher stores the different matches in the packet as described in Algorithm 1. In this notation, *MS* is a MatchSet object structure, M is the set of Match instances it holds and m is a single Match object. Some implementations may, for example, parametrize the Matcher by the condition pattern or embed it directly in the Matcher. The transformation units (*e.g.*, rules) may be compiled in pre/post-condition patterns or interpreted, but this is a tool implementation issue which is not discussed here.

3.2.2 Rewriter

As described in Algorithm 2, the Rewriter applies the required transformation according to the post-condition pattern *condition* on the match specified in the packet it receives from its *packetIn* method. That match is consumed by the Rewriter: no other operation can be further applied on it. Some validations are made in the Rewriter to verify, for example, that π.*current*.*condition* = *condition*.*pre* or that no error occurred during the transformation. In our approach, a modification (update or delete) of an element in $\{m \in M | \langle$condition.*pre*$, M \rangle \in \pi$.*matchSets*$\}$ is automatically propagated to all the other matches, when applicable.

3.2.3 Iterator

The Iterator chooses a match among the set of matches of the *current* condition of the packet it receives from its *packetIn* method, as described in Algorithm 3. The match is chosen randomly in a Monte-Carlo sense, repeatable using sampling from a uniform distribution to provide a reproducible, fair sampling. When its *nextIn* method is called, the Iterator chooses another match as long as the

[2] The bound pivot nodes are stored in *globalPivots*. But the matching may also assign pivots (useful for nested rules, as discussed later) and stores them in *localPivots*.

Algorithm 2 Rewriter.*packetIn*(π)

if π is invalid **then**
  isSuccess $\leftarrow$ **false**
  exception $\leftarrow \chi(\pi)$
  return π
end if
$MS \leftarrow \langle$*condition.pre*$, M\rangle \in \pi$.*matchSets*
apply transformation on MS.*matchToRewrite*
if transformation failed **then**
  isSuccess $\leftarrow$ **false**
  exception $\leftarrow \chi(\pi)$
  return π
end if
set all modified nodes in MS.*matchToRewrite* to *dirty*
remove MS.*matchToRewrite* from MS.*matches*
isSuccess $\leftarrow$ **true**
return π

maximum number of iterations *maxIterations* (possibly infinite) is not yet reached, as described in Algorithm 4. In the case of multiple occurrences of a MatchSet identified by π.*current*, the Iterator selects the last MatchSet.

Algorithm 3 Iterator.*packetIn*(π)

if $\langle \pi$.*current*$, M\rangle \in \pi$.*matchSets* **then**
  $MS \leftarrow \langle \pi$.*current*$, M\rangle$
  choose $m \in MS$.*matches*
  MS.*matchToRewrite* $\leftarrow m$
  iterations $\leftarrow 1$
  isSuccess $\leftarrow$ **true**
  return π
else
  isSuccess $\leftarrow$ **false**
  return π
end if

Algorithm 4 Iterator.*nextIn*(π)

if $\langle \pi$.*current*$, M\rangle \in \pi$.*matchSets* **and** *iterations* $<$ *maxIterations* **then**
  $MS \leftarrow \langle \pi$.*current*$, M\rangle$
  choose $m \in MS$.*matches*
  MS.*matchToRewrite* $\leftarrow m$
  iterations $\leftarrow$ *iterations* $+ 1$
  isSuccess $\leftarrow$ **true**
  return π
else
  isSuccess $\leftarrow$ **false**
  return π
end if

3.2.4 Resolver

The Resolver resolves a potential conflict between matches and rewritings as described in Algorithm 5. For the moment, the Resolver detects conflicts in a simple conservative way: it prohibits any change to other matches in the packet (check for *dirty* nodes). However, it does not verify if a modified match is still valid with respect to its pre-condition pattern. The *externalMatchesOnly* attribute

specifies whether the conflict detection should also consider matches from its match set identified by $\pi.current$ or not. In the case of conflict, a default resolution function is provided but the user may also override it. Although the conflict detection is conservative, the *customResolution* function may discard the conflict if, for example, NACs are not enabled in other matches. That is, the Resolver will detect trivial conflicts, but the transformation engineer is empowered to define the conflicts that may occur in his application domain.

Algorithm 5 Resolver.*packetIn*(π)

```
for all condition c ∈ {c| ⟨c,M⟩ ∈ π.matchSets} do
  if externalMatchesOnly and c = π.current then
    continue
  end if
  for all match m ∈ M do
    if m has a dirty node then
      if not customResolution(π) then
        if not defaultResolution(π) then
          isSuccess ← false
          exception ← χ(π)
          return π
        end if
      end if
    end if
  end for
end for
isSuccess ← true
return π
```

3.2.5 Rollbacker

The Rollbacker provides transactional behaviour with back-tracking capabilities. Consequently, it is used as a recovery point that allows backward recovery of packets, *e.g.*, by means of checkpointing as described in Algorithms 6 and 7. The *packetIn* method establishes a checkpoint of the received packet. This is done by making a copy $\hat{\pi}$ of the input packet π and pushing it on a temporary stack. It also sets the maximum number of iterations to the total number of matches found for the current condition. The *nextIn* method restores the last checkpoint to roll-back the packet to its previous state $\hat{\pi}$. If there are no more matches left in M, it also removes the previous checkpoint established.

Algorithm 6 Rollbacker.*packetIn*(π)

```
establish(π)
if ⟨π.current, M⟩ ∈ π.matchSets then
  maxIterations ← |M|
else
  maxIterations ← max
end if
iterations ← 1
isSuccess ← true
return π
```

Algorithm 7 Rollbacker.*nextIn*(π)

```
π̂ ← restore()
iterations ← iterations + 1
if iterations < maxIterations then
  isSuccess ← true
else
  discard()
  isSuccess ← false
end if
return π̂
```

3.2.6 Selector

The Selector is used when a choice needs to be made between multiple packets processed concurrently by different constructs. It allows exactly one of them to be processed further. When its *successIn* (or *failIn*) method is called, the received packet is stored in its *success* (or *fail*) collection, respectively. Note that, unlike the previously described methods, it is only when the *select* method in Algorithm 8 is called that a packet is returned, chosen from *success*. The selection is random in the same way as in the Iterator. However, if *success* is empty, the returned packet is randomly chosen from *fail*. Note that if both *success* and *fail* are empty, *select* throws an exception with an empty packet π_ϕ. When the *cancel* method is invoked, the two collections are cleared and a cancel message φ is returned where the *exclusions* set consists of the singleton π.*current* (meaning that further operations of the chosen *condition* should not be cancelled).

3.2.7 Synchronizer

The Synchronizer is used when multiple packets processed in parallel need to be synchronized. It is parametrized by the number of *threads* to synchronize. This number is known at design-time. Its *successIn* and *failIn* methods behave exactly like those of the Selector. The Synchronizer is in success mode only if all threads have terminated by never invoking *failIn*. The *merge* method "merges" the packets in *success*, as described in Algorithm 9. A trivial default merge function is provided by unifying and "gluing" the set of packets. Nevertheless, it first conservatively verifies the validity of the received packets by prohibiting overlapping matches between them. If it fails, the user can specify a custom merge function. This avoids the need for static parallel independence detection. Instead it is done at run-time and the transformation modeller must explicitly describe the handler. One pragmatic use of that solution is, for instance, to let the transformation run once to detect the possible conflicts and then the transformation modeller may handle these cases by modifying the transformation model.

Algorithm 8 Selector.*select()*

if *success* $\neq \emptyset$ **then**
  $\hat{\pi} \leftarrow$ choose from *success*
  isSuccess $\leftarrow$ **true**
else if *fail* $\neq \emptyset$ **then**
  $\hat{\pi} \leftarrow$ choose from *fail*
  isSuccess $\leftarrow$ **false**
else
  $\hat{\pi} \leftarrow \pi_\phi$
  isSuccess $\leftarrow$ **false**
  exception $\leftarrow \chi(\pi_\phi)$
end if
success $\leftarrow \emptyset$
fail $\leftarrow \emptyset$
return $\hat{\pi}$

Algorithm 9 Synchronizer.*merge()*

if $|$*success*$| =$ *threads* **then**
  if *customMerge()* **then**
    $\hat{\pi} \leftarrow$ the merged packet in *success*
    isSuccess $\leftarrow$ **true**
    success $\leftarrow \emptyset$
    fail $\leftarrow \emptyset$
    return $\hat{\pi}$
  else if *defaultMerge()* **then**
    $\hat{\pi} \leftarrow$ the merged packet in *success*
    isSuccess $\leftarrow$ **true**
    success $\leftarrow \emptyset$
    fail $\leftarrow \emptyset$
    return $\hat{\pi}$
  else
    isSuccess $\leftarrow$ **false**
    exception $\leftarrow \chi(\pi_\phi)$
    return π_ϕ
  end if
else if $|$*success*$| + |$*fail*$| =$ *threads* **then**
  $\hat{\pi} \leftarrow$ choose from *fail*
  isSuccess $\leftarrow$ **false**
  return $\hat{\pi}$
else
  isSuccess $\leftarrow$ **false**
  exception $\leftarrow \chi(\pi_\phi)$
  return π_ϕ
end if

3.2.8 Composer

The Composer serves as a modular encapsulation interface of the elements in its *primitives* list. When one of its *packetIn* or *nextIn* methods is invoked, it is up to the user to manage subsequent method invocations to its primitives. Nevertheless, when the *cancelIn* method is called, the *Composer* invokes the *cancelIn* method of all its sub-primitives. This cancels the current action of the primitive object by resetting its state to its initial state. Cancelling happens only if a primitive is actively processing a packet π such that the current condition of π is not in φ.*exclusions*, where φ is the received cancel message. In the case of a Matcher, since the current condition of the packet may not already be set, the *cancelIn* also verifies that the condition of the *Matcher* is not in the exclusions list. The interruption of activity can, for instance, be implemented as a pre-emptive asynchronous method call of *cancelIn*. Furthermore, resetting the dirty flag of modified nodes is done in the Composer by calling the *clean* method of a packet. Also, resetting the *success* and *fail*

collections of the control primitives should be done by calling their `reset` method at the appropriate time.

3.3 T-Core: a minimal collection of transformation primitives

In the de-construction process of transformation languages into a collection of primitives, questions like "up to what level?" or "what to include and what to exclude?" arise. The proposed *T-Core* module answers these questions in the following way.

3.3.1 Rationale

In a model transformation language, the smallest transformation unit is traditionally the *rule*. A rule is a complex structure with a declarative part and an operational part. The declarative part of a rule consists of the specification of the rule (*e.g.*, LHS/RHS and optionally NAC in graph transformation rules). However, *T-Core* is not restricted to any form of specification be it rule-based, constraint-based, or function-based. In fact, some languages require units with only a pre-condition to satisfy, while others with a pre- and a post-condition. Some even allow arbitrary permutations of repetitions of the two. In *T-Core*, either a PreConditionPattern or both a Pre- and a PostConditionPattern must be specified. For example, a graph transformation rule can be represented in *T-Core* as a pair of a pre- and a post-condition pattern, where the latter has a reference to the former to satisfy the semantics of the interface K (in the $L \leftarrow K \rightarrow R$ algebraic graph transformation rules) and to be able to perform the transformation. Transformation languages where rules are expressed bidirectionally or as functions are supported in *T-Core* as long as they can be represented as pre- and post-condition patterns.

The operational part of a rule describes how it executes. This operation is often encapsulated in the form of an algorithm (with possibly local optimizations). Nevertheless, it always consists of a *matching phase*, *i.e.*, finding instances of the model that satisfy the pre-condition and of a *transformation phase*, *i.e.*, applying the rule such that the resulting model satisfies the post-condition. *T-Core* distinguishes these two phases by offering a Matcher and a Rewriter as primitives. Consequently, the Matcher's condition only consists of a pre-condition pattern and the Rewriter then needs a post-condition pattern that can access the pre-condition pattern to perform the rewrite. Combinations of Matchers and Rewriters in sequence can then represent a sequence of simple graph transformation rules: *match-rewrite-match-rewrite*. Moreover, because of the separation of these two phases, more general and complex transformation units may be built, such as: *match-match-match* or *match-match-rewrite-rewrite*. The former is a query where each Matcher filters the conditions of the query. The latter is a nesting of transformation rules. In this case, however, overlapping matches between different Matchers and then rewrites on the overlapping elements may lead to inconsistent transformations or even nonsense. This is why a Resolver can be used from *T-Core* to safely allow *match-rewrite* combinations.

The data structure exchanged between *T-Core* RulePrimitives in the form of packets contains sufficient information for each primitive to process it as described in the various algorithms in Section 3.2. The Match allows one to refer to all model elements that satisfy a pre-condition pattern. The pivot

mappings allow elements of certain matches to be bound to elements of previously matched elements. The pivot mapping is equivalent to passing parameters between rules as it will be shown in the example in Section 3.4.1. The MatchSet allows delaying the rewriting phase instead of having to rewrite directly after matching.

Packets conceptually carry the complete model (optimized implementation may relax this) which allows concurrent execution of transformations. The Selector and the Synchronizer both permit one to join branches or threads of concurrent transformations. Also, having separated the matching from the rewriting enables one to manage the matches and the results of a rewrite by further operators. Advanced features such as iteration over multiple matches or back-tracking to a previous state in the transformation are also supported in *T-Core*. If the Rollbacker is used in combination with the Iterator, then the overall behaviour can handle back-tracking for cases where multiple matches are found.

Since *T-Core* is a low-level collection of model transformation primitives, combining its primitives to achieve relevant and useful transformations may involve a large number of these primitive operators. Therefore, it is necessary to provide a "grouping" mechanism. The Composer allows one to modularly organize *T-Core* primitives. It serves as an interface to the primitives it encapsulates. This then enables scaling of transformations built on *T-Core* to large and complex model transformation designs.

T-Core is presented here as an open module which can be extended, through inheritance for example. One could add other primitive model transformation building blocks. For instance, a conformance check operator may be useful to verify if the resulting transformed model still conforms to its meta-model. It can be interleaved between sequences of rewrites or used at the end of the overall transformation as suggested in [KMS+09]. We believe however that such new constructs should either be part of the (programming or modelling) language or the tool in which *T-Core* is integrated, to keep *T-Core* as primitive as possible.

3.3.2 Usage of T-Core

The API of *T-Core* presented in the previous section offers a common interface to all primitive transformation operators. Furthermore, the CompositionPrimitive can be used to encapsulate the execution of other primitives in order to provide abstraction. The `packetIn` method is the entry point of a *T-Core* transformation. Figure 3.2 illustrates a typical interaction with a transformation operator. When a CompositionPrimitive gets initially created, it is responsible of recursively created the instances of its sub-primitives following the composite design pattern [GHJV94]. Its `packetIn` is invoked with a packet that had previously been initialized with the input graph of the transformation. Because the operators support asynchronous execution, the `packetIn` method returns the resulting packet after being processed by the corresponding RulePrimitive *r*. To know whether *r* has been successfully applied or not, one should query the `isSuccess` property of *r*. Similarly, if an exception occurred, the `exception` property of *r* will refer to the corresponding detailed error. Therefore it is important to not forget to set the `isSuccess` property of a custom Composer in case of a successful execution so that the invoking context of the transformation can be aware of that status, as well as any exception that may have occurred. A similar pattern can be used for the `nextIn` method.

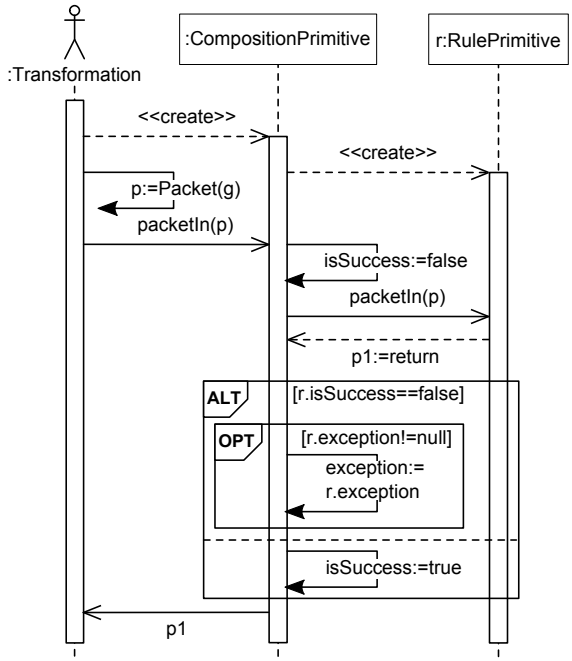


Figure 3.2: Sequence diagram for using a RulePrimitive.

Figure 3.3 illustrates a typical interaction with a Selector. Recall that control primitives accumulate packets and can then produce a single packet: they serve as join points in the transformation. Packets are stored by invoking the *successIn* or *failIn* methods. At the appropriate time, the corresponding join function (*select* for the Selector and merge for the Synchronizer) can be invoked to retrieve a single packet from that operator. In the case of a Selector, a cancel event may be requested to invoke the cancelIn method of the other primitives if the suspension of their activity is desired. The *reset* method shall be invoked afterwards to clear the lists of packets. As in the previous case, verifying for success and errors needs to be integrated as well.

T-Core is designed in such a way that the transformation primitives can be executed independently from one another. To produce a meaningful result (transformation, query, state exploration, etc.), certain operators should preferably be applied before others. For example, Figure 3.4 illustrates the interaction between *T-Core* operators to execute a transformation rule. In the following, we outline good practices for using the *T-Core* primitive operators:

- A Matcher should always be preceded by an Iterator in order to select a match found.
- A Rewriter should not be applied before executing the Matcher whose condition is the pre-condition of the Rewriter's post-condition. Otherwise, the rewriting phase will not affect the input graph and an exception will be reported.
- A Roll-backer will typically receive a *packetIn* message as soon as the enclosing Composer receives a packet. That way, the original graph will be checkpointed and may be restored at a later time via the *nextIn* method.

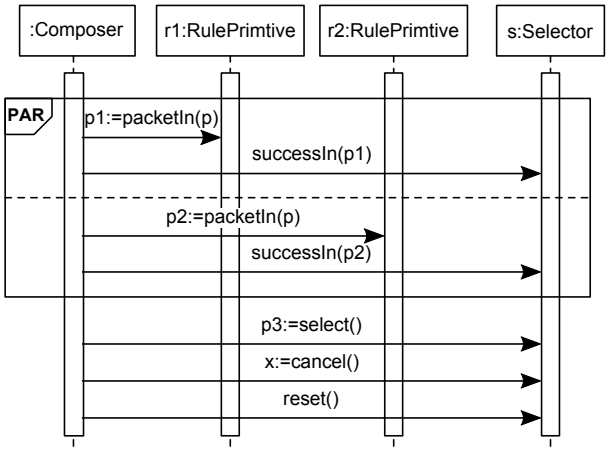


Figure 3.3: Sequence diagram for using a ControlPrimitive.

- A Resolver should be executed after at least one application of a Rewriter. That is because conflicts are detected in an optimistic way, *i.e.*, after a modification of the graph, since the invariant part of the rule is not stored.

3.4 Re-constructing Transformation Languages

De-constructing model transformation languages in a collection of model transformation primitives makes it easier to reason about transformation languages. In fact, properly combining *T-Core* primitives with an existing well-formed programming or modelling language allows us to re-construct some already existing transformation languages and even construct new ones[3]. Figure 3.5 shows some examples of combinations of *T-Core* with other languages. Figure 3.5(a) and Figure 3.5(b) combine a subset of *T-Core* with a simple (programming) language which offers *sequencing*, *branching*, and *looping* mechanisms (as proposed in Böhm-Jacopini's *structured program theorem* [BJ66]). We will refer to such a language as an *SBL language*. The first combination only involves the Matcher and its PreConditionPattern, Packet messages to exchange, and the Composer to organize the primitives. These *T-Core* primitives integrated in an SBL language lead to a *query language*. Since only matching operations can be performed on the model, they represent queries where the resulting packet holds the set of all elements (sub-graph) of the model (graph) that satisfy the desired pre-conditions. Including other *T-Core* primitives such as the Rewriter promotes the query language to a transformation language. Figure 3.5(b) enumerates the *T-Core* primitives combined with an SBL language necessary to design a complete sequential model transformation language. Replacing the SBL language by another one, such as UML Activity Diagrams in Figure 3.5(c), allows us to re-construct Story Diagrams [FNTZ00], for example, since they are defined as a combination of UML Activity and Collaboration Diagrams with graph transformation features. Other combinations involving the whole *T-Core* module may lead to novel transformation languages with exception handling and the notion of timed model transformations when combined with a discrete-event modelling language (c.f., Part III of this thesis).

[3]This is the subject of Part III of this thesis.

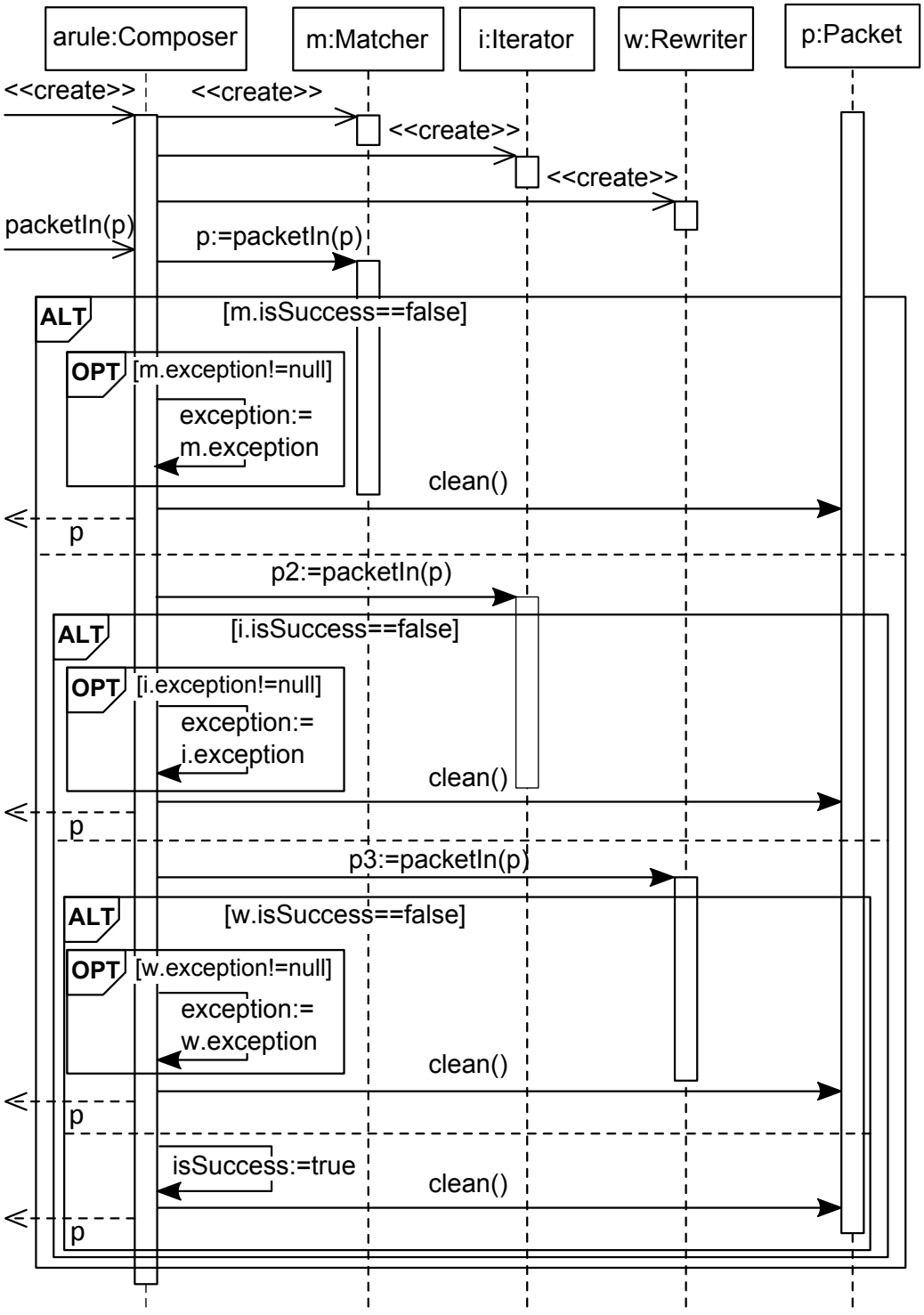


Figure 3.4: Sequence diagram of a simple rule execution.

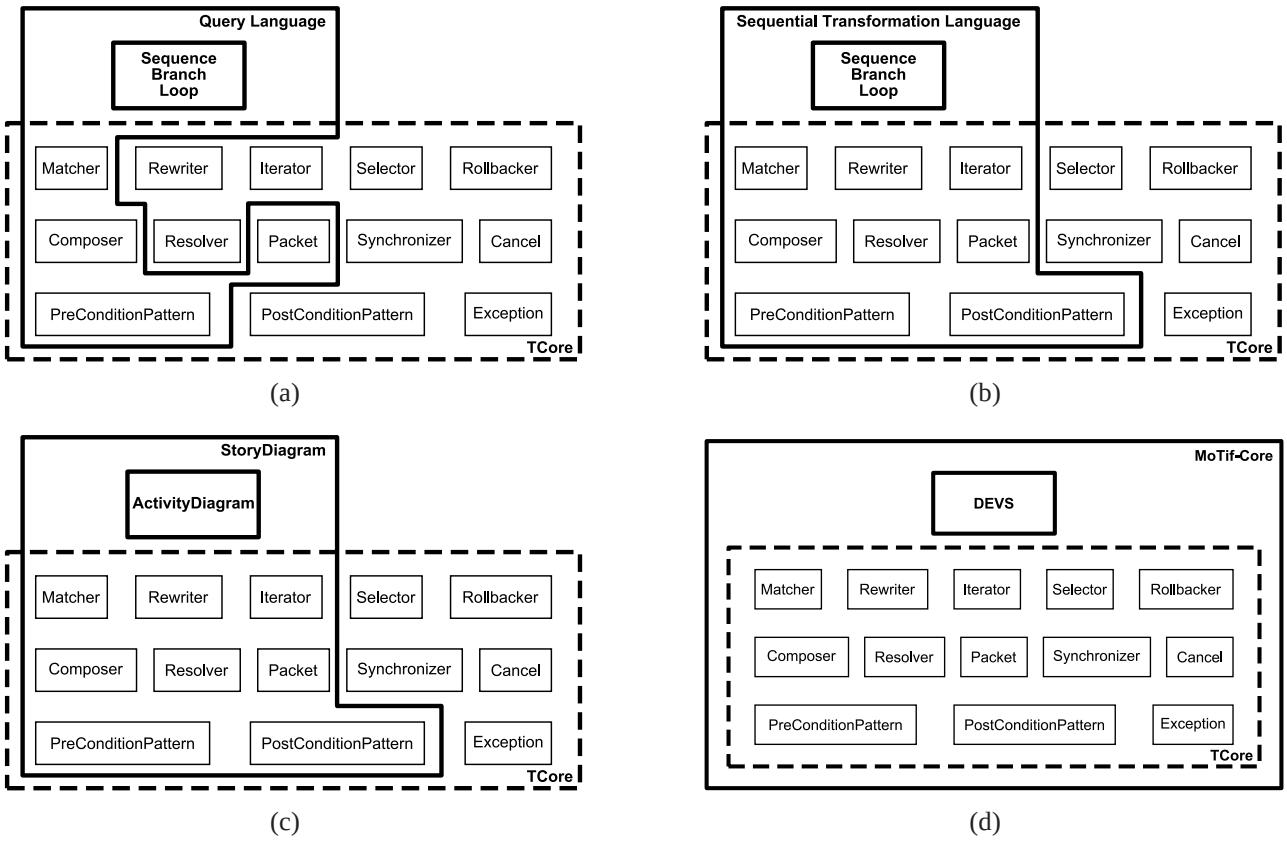


Figure 3.5: Combining *T-Core* with other languages allows one to re-construct existing and new languages.

We now present the re-construction of two transformation features using the combination of an SBL language with *T-Core* as in Figure 3.5(b).

3.4.1 Re-constructing Story Diagrams

In the context of object-oriented reverse-engineering, the *FUJABA* tool allows the user to specify the content of a class method by means of Story Diagrams, an extension of UML Activity Diagrams. A Story Diagram organizes the behaviour of a method with activities and transitions. An activity can be a Story Pattern or a statement activity. The former consists of a graph transformation rule and the latter is Java code. Figure 3.6 shows such a story diagram taken from the `doDemo` method example in [FNTZ00]. This snippet represents an elevator loading people on a given floor of a house who wish to go to another level. The rule in the pattern is specified in a UML Collaboration Diagram-like notation [Obj09] with objects and associations. Objects with implicit types (*e.g.*, `this`, `l2`, and `e1`) are *bound* objects from previous patterns or variables in the context of the current method. The Story Pattern 6 is a for-all Pattern. Its rule is applied on all matches found looping over the unbound objects (*e.g.*, `p4`, and `l4`). The outgoing transition labelled `each time` applies statement 7 after each iteration of the for-all

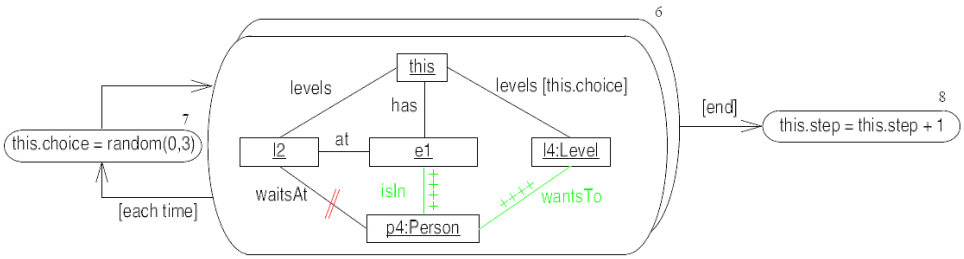


Figure 3.6: The *FUJABA* doSubDemo transformation showing a for-all Pattern and two statement activities.

Pattern. This activity allows the pattern to simulate random choices of levels for different people. When all iterations have been completed, the flow proceeds with statement 8 reached by the transition labelled *end*, which simulates the elevator going one level up.

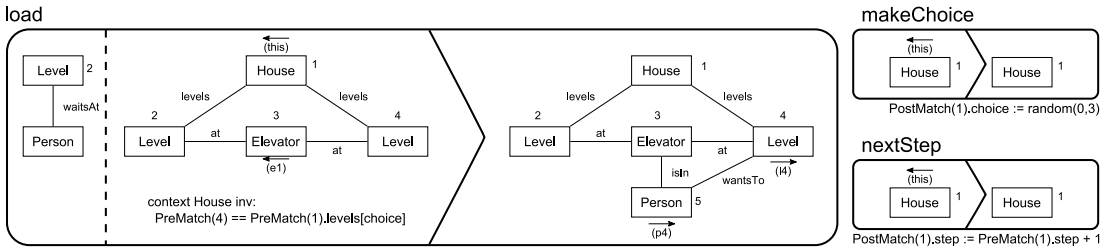


Figure 3.7: The three *MoTif* rules for the doSubDemo transformation.

We now show how to re-construct this non-trivial story diagram transformation from an SLB language combined with *T-Core*. An instance of that combination is called a *T-Core model*. First, we design the rules needed for the conditions of rule primitives. Figure 3.7 describes the three necessary rules corresponding to the three Story Diagram activities. We use the visual concrete syntax of *MoTif* [SV10] where the central compartment is the LHS, the compartment on the right of the arrow head is the RHS and the compartment(s) on the left of dashed lines are the NAC(s). The concrete syntax for representing the pattern was chosen to be intuitively close enough to the *FUJABA* graphical representation. Numeric labels are used to uniquely identify different elements across compartments. Elements with an alpha-numeric label between parentheses denote pivot elements. A right-directed arrow on top of the label depicts that the model element matched for this pattern element is assigned to a pivot (*e.g.*, `p4` and `l4`). A left-directed arrow on top of the label depicts that the model element matched for this pattern element is bound to the specified pivot (*e.g.*, `this` and `e1`).

The *T-Core* model equivalent to the original doSubDemo transformation consists of a Composer doSubDemoC. The hierarchy of its sub-primitives is illustrated in the Collaboration Diagram in Figure 3.8. It is composed of two Composers loadC and nextStepC each containing a Matcher, an Iterator, a Rewriter, and a Resolver. The `packetIn` method of doSubDemoC first calls the corresponding method of loadC and then feeds the returned packet to the `packetIn` method of nextStepC. This ensures that the output packet of the overall transformation is the result of first loading all the Person

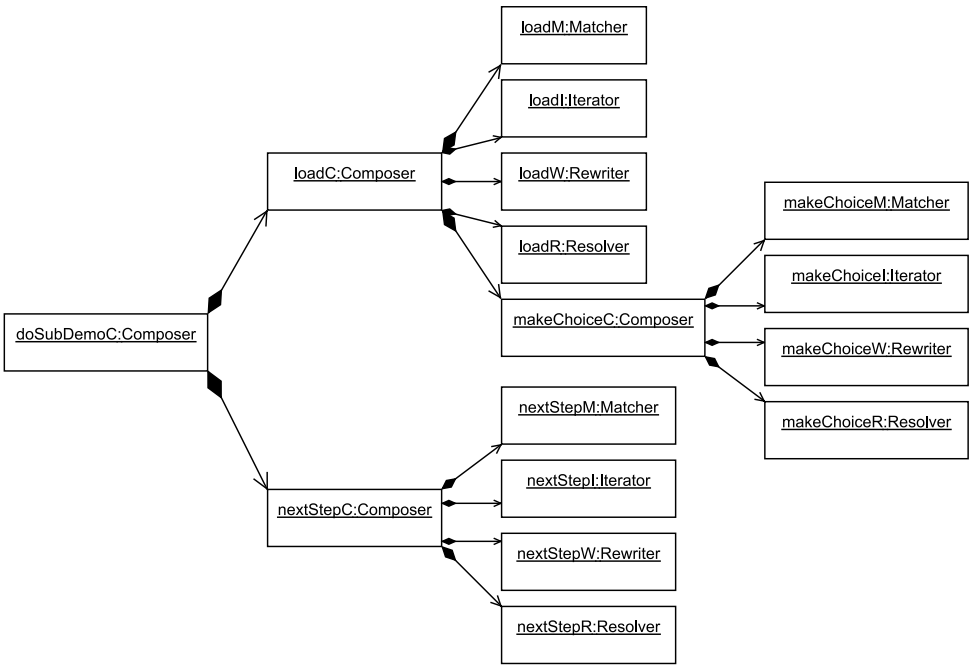


Figure 3.8: The object hierarchy of the doSubDemo composer.

objects and then moving the elevator by one *step*. Algorithm 10 describes this behaviour.

Algorithm 10 doSubDemoC.*packetIn*(π)

```
π ← loadC.packetIn(π)
π ← nextStepC.packetIn(π)
isSuccess ← true
return π
```

makeChoiceC and nextStepC behave as simple transformation rules. Their *packetIn* method behaves as specified in Algorithm 11. First, the matcher is tried on the input packet. Note that the conditions of the matchers makeChoiceM and nextStepM are the LHSs of rules `makeChoice` and `nextStep`, respectively. If the matcher fails, the composer goes into failure mode and the packet is returned. Then, the iterator chooses a match. Subsequently, the rewriter attempts to transform this match. Note that the conditions of the rewriters makeChoiceW and nextStepW are the RHSs of rules `makeChoice` and `nextStep`, respectively. If the rewriter fails, an exception is thrown and the transformation stops. Otherwise, the resolver verifies the application of this pattern with respect to other matches in the transformed packet. The behaviour of the resolution function will be elaborated on later. Finally, on a successful resolution, the resulting packet is output and the composer is put in success mode.

loadC is the composer that emulates the for-all Pattern of the example. Algorithm 12 specifies that behaviour. After finding all matches with loadM (whose condition is the LHS and the NAC of rule

load), the packet is forwarded to the iterator loadI to choose a match. The iteration is emulated by a loop with the failure mode of loadI as the breaking condition. Inside the loop, loadW rewrites the chosen match and loadR resolves possible conflicts. Then, the resulting packet is sent to makeChoiceC to fulfil the each time transition of the story digram. After that, the *nextIn* method of loadI is invoked with the new packet to choose a new match and proceed in the loop.

Algorithm 11 makeChoiceC.*packetIn(π)*

```
isSuccess ← false
π ← makeChoiceM.packetIn(π)
if not makeChoiceM.isSuccess then
  return π
end if
π ← makeChoiceI.packetIn(π)
if not makeChoiceI.isSuccess then
  return π
end if
π ← makeChoiceW.packetIn(π)
if not makeChoiceW.isSuccess then
  return π
end if
π ← makeChoiceR.packetIn(π)
if not makeChoiceW.isSuccess then
  return π
end if
isSuccess ← true
return π
```

Algorithm 12 loadC.*packetIn(π)*

```
isSuccess ← false
π ← loadM.packetIn(π)
if not loadM.isSuccess then
  return π
end if
π ← loadI.packetIn(π)
if not loadI.isSuccess then
  return π
end if
while true do
  π ← loadW.packetIn(π)
  if not loadW.isSuccess then
    return π
  end if
  π ← loadR.packetIn(π)
  if not loadR.isSuccess then
    return π
  end if
  π ← makeChoiceC.packetIn(π)
  π ← loadI.nextIn(π)
  if not loadI.isSuccess then
    isSuccess ← true
    return π
  end if
end while
```

Having seen the overall *T-Core* transformation model, let us examine how the different Resolvers should behave in order to provide a correct and complete transformation. The first rewriter called is loadW and the first time it receives a packet is when a transformation is applied on one of the matches of the matcher loadM. Therefore each match consists of the same House (since it is a bound node), two Levels, an Elevator, and the associations between them. On the other hand, loadW only adds a Person and links it to a Level. Therefore the default resolution function of the resolver loadR applies successfully, since no matched element is modified nor is the NAC violated in any other match. The next resolver is makeChoiceR which is in the same loop as loadR. There, the House is conflicting

with all the matches in the packet according to the conservative default resolution function. Note that makeChoiceM finds at most one match (the bound House element). However, makeChoiceW does not really conflict with matches found in loadM. We therefore specify a custom resolution function for makeChoiceR that always succeeds. The same applies for nextStepR.

3.4.2 Re-constructing amalgamated rules

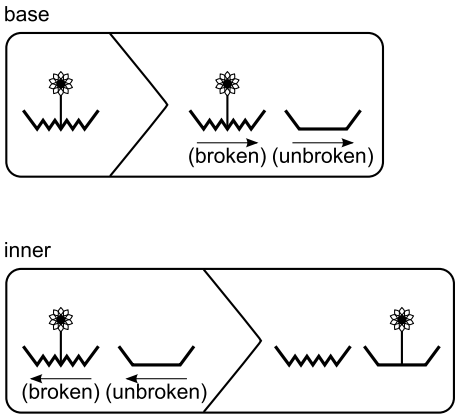


Figure 3.9: The transformation rules for the *Repotting Geraniums* example

Algorithm 13 baseC.*packetIn(π)*

```
isSuccess ← false
π ← baseM.packetIn(π)
if not baseM.isSuccess then
    return π
end if
while true do
    π ← baseI.packetIn(π)
    if baseI.isSuccess then
        π ← baseW.packetIn(π)
        if not baseW.isSuccess then
            return π
        end if
        π ← baseR.packetIn(π)
        if not baseR.isSuccess then
            return π
        end if
        π ← innerC.packetIn(π)
    end if
    π ← baseM.packetIn(π)
    if not baseM.isSuccess then
        isSuccess ← true
        return π
    end if
end while
```

In a recent paper, Rensink *et al.* claim that the *Repotting the Geraniums* example is inexpressible in most transformation formalisms [RK09]. The authors propose a transformation language that uses an amalgamation scheme for nested graph transformation rules. As we have seen in the previous example, nesting transformation rules is possible in *T-Core* and will be used to solve the problem. It consists of *repotting all flowering geraniums whose pots have cracked*. Figure 3.9 illustrates the two nested graph transformation rules involved and Algorithm 13 demonstrates the composition of primitive *T-Core* elements encoding these rules. baseM (with, as condition, the LHS of rule base) finds all broken pots containing a flowering geranium, given the input packet containing the input graph. The set of matches resulting in the packet are the combination of all flowering geraniums

and their pot container. From then on starts the loop. First, baseI chooses a match. If one is chosen, baseW transforms this match and baseR resolves any conflicts. In this case, baseW only creates a new unbroken pot and assigns pivots. Therefore, baseR's resolution function always succeeds. In fact, the resolver is not needed here, but we include it for consistency. The innerC composer encodes the `inner` rule which finds the two bound pots and moves a flourishing flower in the broken pot to the unbroken one. In order to iterate over all the flowers in the broken pot, the innerC.*`packetIn`* method has the exact same behaviour as loadC.*`packetIn`* in Algorithm 12, with the exception of not calling a sub-composer (like makeChoiceC). Note that an always successful custom resolution function for innerR is required. After the Resolver successfully outputs the packet, the `inner` rule is applied. Then (and also if baseI had failed) baseM.*`packetIn`* is called again with the resulting packet. The loop ends when the baseM.*`packetIn`* method call inside the loop fails, which entails baseC returning the final packet in success mode.

3.5 Transformation Language Product Line

There is a wide variety of transformation languages and tools that exist today. Also, they are very powerful in solving the problems they were initially intended for. For example, *FUJABA* [NNZ00] is primarily meant to provide reverse-engineering capability, *AToM*^{3} [dLV04] and *GReAT* for defining translational semantics and simulation of formalisms, the new version of *VIATRA2* [BÖR$^+$08] to provide means for model synchronization, etc. However, most of them have a tendency to provide a generic tool for solving any kind of model transformation problem. This is especially true with the arrival of *QVT* [Obj08] and most applications of *ATL* [Pro10a]. This genericity requires transformation languages to be very expressive, which makes analysis of transformation models built using these *general purpose* transformation languages very hard. In fact, some approaches have realized this problem and propose Turing-incomplete transformation languages, such as *DSLTrans* [BLA$^+$10].

The solution proposed here is to use a sub-set of *T-Core* primitives to restrict a transformation language for one specific purpose or intention. To some extent, one can redefine a transformation language as consisting of the following features:

1. **Primitive transformation operators**, for example taken from (a sub-set of) the *T-Core* module;
2. Combined with a **scheduling language**, which can be programmed (*e.g.*, Java [EETW06]) or modelled (*e.g.*, UML Activity diagrams [LLMC06], Coloured Petri nets [WKS$^+$09]).

In fact, the scheduling language may be a domain-specific language dedicated for defining transformation schedulers. The combination of both provides a product line of **problem-specific** transformation languages. This restricts the transformation engineer to focus entirely on designing transformation models without added complexity that is irrelevant for the purpose of the transformation. Also, the transformation language has no more expressiveness than is needed and this may allow for better analysis of the transformation models. Nevertheless, the expressiveness of the transformation language then depends on the glue language (*i.e.*, the scheduler) used and the primitive operators chosen.

Py-T-Core

Currently *T-Core* is implemented in Python and is available at the website [Syr10]. It is a direct implementation of the class diagram of Figure 3.1. Therefore, the combination of *T-Core* primitives with Python as a scheduling language seems adequate. This results in a new transformation language, called *Py-T-Core*[4].

For example, a query is defined as in Listing 3.1: given a packet, if a match is found it is selected and the resulting packet is output. The packet then consists of a single match set containing a single match. This match describes the sub-graph that satisfies the pre-condition pattern *i.e.*, the query.

Listing 3.1: A query in *Py-T-Core*.

```
from t_core.composer import Composer
from t_core.matcher import Matcher
from t_core.iterator import Iterator

class Query(Composer):
    def __init__(self, LHS):
        super(Query, self).__init__()
        self.M = Matcher(condition=LHS, max=1)
        self.I = Iterator(max_iterations=1)

    def packet_in(self, packet):
        self.exception = None
        self.is_success = False
        # Match
        packet = self.M.packet_in(packet)
        if not self.M.is_success:
            self.exception = self.M.exception
            return packet
        # Choose the only match
        packet = self.I.packet_in(packet)
        if not self.I.is_success:
            self.exception = self.I.exception
            return packet
        # Output success packet
        self.is_success = True
        return packet
```

Listing 3.2 illustrates how a simple rule is defined, such as in Algorithm 11.

Listing 3.2: A simple rule in *Py-T-Core*.

```
from t_core.composer import Composer
from t_core.matcher import Matcher
from t_core.iterator import Iterator
from t_core.rewriter import Rewriter
from t_core.resolver import Resolver
```

[4]Similarly, an implementation in C would be called *C-T-Core* or in Java would be called *J-T-Core*.

```
class ARule(Composer):
    def __init__(self, LHS, RHS, ignore_resolver=False,
                 external_matches_only=False,
                 custom_resolution=lambda packet: False):
        super(ARule, self).__init__()
        self.ignore_resolver = ignore_resolver
        self.M = Matcher(condition=LHS, max=1)
        self.I = Iterator(max_iterations=1)
        self.W = Rewriter(condition=RHS)
        self.R = Resolver(external_matches_only, custom_resolution)

    def packet_in(self, packet):
        self.exception = None
        self.is_success = False
        # Match
        packet = self.M.packet_in(packet)
        if not self.M.is_success:
            self.exception = self.M.exception
            return packet
        # Choose the only match
        packet = self.I.packet_in(packet)
        if not self.I.is_success:
            self.exception = self.I.exception
            return packet
        # Rewrite
        packet = self.W.packet_in(packet)
        if not self.W.is_success:
            self.exception = self.W.exception
            return packet
        if not self.ignore_resolver:
            # Resolve any conflicts if necessary
            packet = self.R.packet_in(packet)
            if not self.R.is_success:
                self.exception = self.R.exception
                return packet
        # Output success packet
        self.is_success = True
        return packet
```

Listing 3.3 shows how an iterative rule applied on multiple matches is defined. This is similar to what was described in Algorithm 12, with the exception that the latter had a nested rule applied at each iteration.

Listing 3.3: A rule applied on all matches at once in *Py-T-Core*.

```
from util.infinity import INFINITY
from arule import ARule

class FRule(ARule):
    def __init__(self, LHS, RHS, ignore_resolver=False,
```

```
                external_matches_only=False,
                custom_resolution=lambda packet: False,
                max_iterations=INFINITY):
        super(FRule, self).__init__(LHS, RHS, ignore_resolver,
                                    external_matches_only, custom_resolution)
        # Matcher needs to find many matches
        self.M.max = max_iterations
        self.I.max_iterations = max_iterations

    def packet_in(self, packet):
        self.exception = None
        self.is_success = False
        # Match
        packet = self.M.packet_in(packet)
        if not self.M.is_success:
            self.exception = self.M.exception
            return packet
        # Choose the first match
        packet = self.I.packet_in(packet)
        if not self.I.is_success:
            self.exception = self.I.exception
            return packet
        while True:
            # Rewrite
            packet = self.W.packet_in(packet)
            if not self.W.is_success:
                self.exception = self.W.exception
                return packet
            if not self.ignore_resolver:
                # Resolve any conflicts if necessary
                packet = self.R.packet_in(packet)
                if not self.R.is_success:
                    self.exception = self.R.exception
                    return packet
            # Choose another match
            packet = self.I.next_in(packet)
            # No more iterations are left
            if not self.I.is_success:
                if self.I.exception:
                    self.exception = self.I.exception
                else:
                    # Output success packet
                    self.is_success = True
                return packet
```

Listing 3.4 depicts the definition of a rule to be applied as long as there are matches. This is similar to what was described in Algorithm 13, with the difference that the latter also had a nested rule applied inside the loop.

Listing 3.4: A rule applied as long as possible in *Py-T-Core*.

```
from util.infinity import INFINITY
from arule import ARule

class SRule(ARule):
    def __init__(self, LHS, RHS, ignore_resolver=False,
                 external_matches_only=False,
                 custom_resolution=lambda packet: False,
                 max_iterations=INFINITY):
        super(SRule, self).__init__(LHS, RHS, ignore_resolver,
                                    external_matches_only, custom_resolution)
        self.I.max_iterations = max_iterations

    def packet_in(self, packet):
        self.exception = None
        self.is_success = False
        # Match
        packet = self.M.packet_in(packet)
        if not self.M.is_success:
            self.exception = self.M.exception
            return packet
        return self.transform(packet)

    def transform(self, packet):
        # Choose the first match
        packet = self.I.packet_in(packet)
        if not self.I.is_success:
            self.exception = self.I.exception
            return packet
        while True:
            # Rewrite
            packet = self.W.packet_in(packet)
            if not self.W.is_success:
                self.exception = self.W.exception
                return packet
            if not self.ignore_resolver:
                # Resolve any conflicts if necessary
                packet = self.R.packet_in(packet)
                if not self.R.is_success:
                    self.exception = self.R.exception
                    return packet
            # Rule has been applied once, so it's a success anyway
            self.is_success = True
            if self.I.iterations == self.I.max_iterations:
                return packet
            # Re-Match
            packet = self.M.packet_in(packet)
            if not self.M.is_success:
                self.exception = self.M.exception
                return packet
```

```
            # Choose another match
            packet = self.I.next_in(packet)
            # No more iterations are left
            if not self.I.is_success:
                if self.I.exception:
                    self.exception = self.I.exception
                return packet
```

Listing 3.5 shows an example of how to combine rules and primitives with the procedural constructs of Python. This describes the solution of the *Distributed Mutual Exclusion Algorithm* benchmark presented in [VSV05].

Listing 3.5: The composition of different rules in *Py-T-Core*.

```
class ShortTransformationSequence(Composer):
    def __init__(self, N, debug_folder=''):
        super(ShortTransformationSequence, self).__init__()
        self.N = N
        self.NewRule = SRule(HNewRuleLHS(), HNewRuleRHS(),
                             max_iterations=N-2, ignore_resolver=True)
        self.MountRule = ARule(HMountRuleLHS(), HMountRuleRHS(),
                               ignore_resolver=True)
        self.RequestRule = FRule(HRequestRuleLHS(), HRequestRuleRHS(),
                                 max_iterations=N, ignore_resolver=True)
        self.TakeRule = ARule(HTakeRuleLHS(), HTakeRuleRHS(),
                              ignore_resolver=True)
        self.ReleaseRule = ARule(HReleaseRuleLHS(), HReleaseRuleRHS(),
                                 ignore_resolver=True)
        self.GiveRule = ARule(HGiveRuleLHS(), HGiveRuleRHS(),
                              ignore_resolver=True)

    def packet_in(self, packet):
        # New Processes
        packet = self.NewRule.packet_in(packet)
        packet.clean()
        if not self.NewRule.is_success:
            if self.NewRule.exception is not None:
                self.exception = self.NewRule.exception
                return packet
        # Mount
        packet = self.MountRule.packet_in(packet)
        packet.clean()
        if not self.MountRule.is_success:
            if self.MountRule.exception is not None:
                self.exception = self.MountRule.exception
                return packet
        # Request
        packet = self.RequestRule.packet_in(packet)
        packet.clean()
        if not self.RequestRule.is_success:
            if self.RequestRule.exception is not None:
```

```
            self.exception = self.RequestRule.exception
            return          # Pass it around
        for _ in range(self.N):
            # Take
            packet = self.TakeRule.packet_in(packet)
            packet.clean()
            if not self.TakeRule.is_success:
                if self.TakeRule.exception is not None:
                    self.exception = self.TakeRule.exception
                    return packet
            # Release
            packet = self.ReleaseRule.packet_in(packet)
            packet.clean()
            if not self.ReleaseRule.is_success:
                if self.ReleaseRule.exception is not None:
                    self.exception = self.ReleaseRule.exception
                    return packet
            # Give
            packet = self.GiveRule.packet_in(packet)
            packet.clean()
            if not self.GiveRule.is_success:
                if self.GiveRule.exception is not None:
                    self.exception = self.GiveRule.exception
                    return packet
        self.is_success = True
        return packet
```

Py-T-Core allows a programmed[5] software to integrate with model transformation solutions thanks to the *T-Core* API. This is a pragmatic solution to bridge the gap between software developers (who program large-scale systems) and domain experts (who describe the behaviours of their model through transformation).

3.6 Related work

The closest work to the one presented here is [VJBB09]. In the context of global model management, the authors define a type system offering a set of primitives for model transformation. The advantage of our approach is that *T-Core* is described here as a module and is thus directly implementable. Also, the approach described in [VJBB09], does not deal with exceptions at all unlike *T-Core*. Nevertheless, their framework is able to achieve higher-order transformations (HOTs), *i.e.*, transformations that operate on model transformations. The implementation of *T-Core* is currently available in Python. Since this is an object-oriented language, the *T-Core* primitive operators are implemented as classes. Thus, at run-time, the operators are objects which can be directly manipulations and thus emulate HOTs. However, as it will be seen in Chapter 7, *T-Core* can be combined with a modelling language. Thus, HOTs can be easily specified in such a completely modelled transformation language.

[5] As opposed to a modelled software where no artefacts are hard-coded.

The *GP* graph transformation language [MP08] also offers transformation primitives. The authors however focus more on the scheduling of the rules then on the rules themselves. Their scheduling (control) language is an extension of an SBL language. Our approach is more general since much more complex scheduling languages (*e.g.*, allowing concurrent and timed transformation execution) can be integrated with *T-Core*. Although it performs very efficiently, the application area of *GP* is more limited, as it can not deal with arbitrary domain-specific models.

Other graph transformation tools, such as *VIATRA* [VB07] and *GReAT* [AKK+06], have their own virtual machine used as an API. In our approach, since the primitive operations are modelled, they are completely compatible with other existing model transformation frameworks.

T-Core does cover a significant amount of variation in pattern-based model transformation. For example, we showed how to solve the amalgamated rule problem where pattern elements are combined with universal and existing quantifiers. This was done by wisely "nesting" pre-condition patterns with the use of pivots. Other pattern compositions include disjunctive constructs such as in [BV06]. That is, a LHS pattern can consist of sub-patterns that can be conjuncted and disjuncted. Chapter 10 will show how this can be accomplished with *T-Core* primitives. When the LHS consists of two disjuncted patterns, we first split each disjunctive case in separate pre-condition patterns. Then, the `packetIn` method of the Matcher of each pattern is called. Each resulting packet is output to a Selector which finally selects one of the packets.

3.7 Conclusion

This chapter motivated the need for providing model transformation language primitives. *T-Core* was defined by precisely describing each of these primitive constructs. The de-construction process of model transformation languages enabled us to re-construct existing simple model transformation features as well as more complex ones by combining *T-Core* with, for example, an SBL language. This allowed us to compare different model transformation languages using a common basis. Furthermore, *T-Core* is combined with a programming language which allows non-MDE users to integrate with MDE solutions. This integration is transparent for programmers since *Py-T-Core* and *T-Core* offer a complete API.

T-Core was presented as a minimal collection of model transformation primitives, defined at the optimal level of granularity. *T-Core* is not restricted to any form of specification of transformation units, be it rule-based, constraint-based, or function-based. It can also represent bidirectional and functional transformations as well as queries. *T-Core* modularly encapsulates the combination of these primitives through composition, re-use, and a common interface. It is an executable module that is easily integrable with a programming or modelling language.

It is impossible to prove that *T-Core* is a collection of the most primitive transformation operators, because of the complexity and diversity of the expressiveness of most model transformation languages. However, our experience showed that it can be used to reproduce most of the languages described in Chapter 2. Note that, for example, declarative transformations defined as relations, such as in *QVT-R* or *TGG*, cannot be directly expressed using *T-Core* primitives. That is because their transfor-

mation units specify relations between the involved meta-models as opposed to the operational nature of transformation rules. However, if these relations can be compiled into operational rules (such as in *TGG* c.f. Chapter 2.2.3), then *T-Core* primitives can be used to mimic the corresponding behaviour of the relations.

The detection of conflicts in the Resolver and the Synchronizer is conservative. A possible extension would be to incorporate more advanced detection mechanisms, such as through critical pair analysis [LEO08]. However, this technique assumes that the transformation units are traditional graph transformation rules with a single *match-rewrite* combination, which is not always the case in *T-Core*.

T-Core can serve as a basis for inter-operating model transformations expressed in different formalisms. That is, by mapping each and every construct of the languages to an appropriate combination of *T-Core* operators. In [HKA10], the authors define a language for composing heterogeneous transformations defined in different formalisms (*e.g.*, *ATL* and *QVT-OM*). Their approach is to wrap each transformation model in *components* and communicate between each other via in/out-port connections, treating the transformation models as black-boxes. This is the opposite of opening the languages and mapping them to a common denominator: *T-Core*. The disadvantage of their approach is that port connection consistency is validated through simple type checking. Also, their current implementation is restricted to models only represented in Ecore.

Now that the primitives are well-defined, efficiently implementing each of them will certainly lead to more efficient model transformation languages.

4

Implementation of Himesis

One of the key aspects to address industrial-scale model-driven engineering problems is the ability to execute model transformations on large models efficiently. Therefore a first step is to provide an efficient implementation of the model transformation primitives described in Chapter 3. In this chapter, we first design and implement efficient data structures to represent models to-be-transformed as well as transformation models. We describe the implementation of the *T-Core* transformation primitives in the form of Himesis, a kernel for graph-based model representation and manipulation. We then implement the transformation primitives described in the previous chapter. The performance of our implementation is thoroughly analysed.

4.1 Introduction

Model-based development (MBD) is increasingly adopted in industry. However, the models industry deals with are very large, with up to 10^6 elements. Modern (MBD) tools, such as AToMPM (our successor of *AToM*^{3} [dLV02]), must allow the modeller to work with such industrial-scale models. These tools should be able to handle common tasks such as loading, saving, visually representing, and transforming large models.

Graphs, as opposed to meta-model/instance, are commonly used to represent models (because they are often truly graph-like, such as Petri nets) and this is also the choice made in AToMPM. The goal of this chapter is to develop and analyse the performance of the data structures used internally to represent typed, attributed, directed graphs. This graph kernel is called Himesis[1]. In order to make the kernel of AToMPM as efficient as possible, Himesis must allow one to efficiently manipulate graphs. In AToMPM, models are represented as graphs. Moreover, the tool is implemented in the Python language. We therefore restrain our investigation to compare two potential candidate graph representation/manipulation software libraries implemented in Python, namely *IGraph* [CN06] and *NetworkX* [HSS08]. A thorough performance analysis allows us to choose the most efficient one for our purposes. Furthermore, we extend it to efficiently manipulate models, more specifically to perform model transformation. We describe the different algorithms that are used and compare their performance to other model transformation tools by means of a standard graph transformation benchmark.

[1]This name was first introduced in [Pro05]. It is derived from “genesis” for origin and “mimesis” for representation. The syllable “hi” stands for hierarchical.

Chapter 3 abstracted away details such as the data structure representing models as graphs, the matching procedure of the Matcher, and the transformation procedure of the Rewriter. This chapter focuses on the implementation of these aspects of *T-Core*. In Section 4.2, we first compare the performance of two promising software libraries with Python APIs for graph creation and manipulation. Section 4.3 examines the performance of the most efficient one, focusing on the tasks common in model-based tools. One critical task in particular is to find sub-graphs in a given graph, respecting some constraint conditions (meta-model type, multiplicities, OCL constraints, etc). Since graph transformation relies heavily on the matching algorithm, Section 4.4 describes the algorithms implemented in Himesis for (1) computing sub-graph isomorphisms and (2) pattern matching as used in model transformation. The performance of Himesis is analysed in Section 4.5 by means of a standard graph transformation benchmark.

4.2 Making the Right Choice

Our search for existing libraries that efficiently manipulate graphs resulted in two potential candidates: *IGraph* [CN06] and *NetworkX* [HSS08].

4.2.1 IGraph and NetworkX

Both IGraph [igr09] and NetworkX [net10] are open source software packages for creating and manipulating graphs. IGraph is implemented in ANSI C and it offers a Python API. NetworkX is entirely implemented in Python. They both allow creating directed multi-graphs, *i.e.,* graphs whose edges have a source to target orientation and there can be more than one edge between any two (not necessarily distinct) nodes. Attributes can be assigned to nodes, edges, or to the graph itself[2]. The values of attributes can be of any type, including graphs, thus supporting hierarchical graphs [DHP02]. Although both libraries exhibit very similar features, they differ in the way data is stored internally.

In IGraph, nodes are not explicitly stored. Instead, the internal structure only keeps track of the total number of nodes in the graph. Nodes and edges are each identified by a non-negative integer ID. Node and edge ID numbering is always continuous which may require re-numbering when a node is deleted. Consequently, the attribute values of a node are not stored in the node itself. Instead, a vector is assigned globally to the graph. The drawback is that if an attribute is only meaningful for a small subset of nodes, the required memory space will be assigned for all nodes, as if they all had this attribute defined. Attributes are conveniently accessible by lookup/reference tables.

In NetworkX, a graph is stored by its adjacency list implemented in a Python dictionary of dictionaries. The outer dictionary is indexed by nodes and their values are themselves dictionaries thus encoding edges adjacent to the indexed node. The inner dictionary is indexed by neighbouring nodes and their values are edge attributes associated with that edge. Nodes can take the form of any hashable Python object. For non-hashable objects, NetworkX allows one to represent the node as a unique identifier and assign the data as a node attribute. This is similar to how IGraph allows arbitrary objects to be stored in a node. Unlike with IGraph, with NetworkX the burden is on the developer to guarantee

[2]We only considered attributed nodes for the experiments in this section.

the uniqueness of the identifiers.

4.2.2 IGraph vs. NetworkX

We will now examine how each library performs for model manipulation tasks. The tasks that concern us are: creation, deletion, and modification of nodes and edges, as well as the traversal of all the elements in the graph (the well known CRUD operations).

Experimental Conditions

Figure 4.1 shows a heat graph, represented as a table, assessing for which case one library is more optimal than the other. For each operation we vary two parameters. The first one is the number of times n an operation is applied. In this comparison, we generate an initial Erdõs-Rényi random graph $G(50, 0.5)$. It is a graph with 50 nodes such that an edge is created between any two nodes with probability $p = 0.5$, the randomness being sampled from a uniform distribution function. The probability chosen generates a dense graph, given that directed multiple edges and self loops are allowed. To ensure non-biased experiments, the same initial graph G is used for both libraries, by exporting the adjacency list of the generated graph reconstructing it in IGraph and in NetworkX. The second parameter is how data is stored in the graph. For this experiment, we evaluate the case when no data is stored in the graph (depicted by the "No Attributes" label in Figure 4.1) and when nodes hold attribute values (depicted by the "Attributed" label in Figure 4.1)

The table in Figure 4.1 represents the results of the experiments along three dimensions: whether data is stored, the operation under study, and the number of times the operation is performed. It shows the ratios $r \in [0, +\infty[$ of the computation time between the two libraries[3] *i.e.*,

$$r = \frac{t_{\text{NetworkX}}}{t_{\text{IGraph}}}$$

When $0 \leq r < 1$, NetworkX is faster and when $r > 1$, IGraph is faster. The boundary case of $r = 1$ simply depicts that they were as fast for performing the same operation. The ratio depends on three dimensions:

- $op \in \{AN, AE, UN, T, DE, DN\}$ is the operation of interest: *Add nodes*, *Add edges*, *Update nodes*, *Traverse*, *Delete edges*, and *Delete nodes*, respectively.
- $d \in \{NA, AT\}$ indicates whether data is stored: *No attributes* or *Attributed*, the latter stores data at the node level using the library's node attribute mechanism. For the attributed case, the size of the data stored at each node is 4,118 bytes, which is considered as a light-weight attribute in Python.
- $n \in \mathbb{N}$ is the number of times an operation has been applied in sequence.

In this experiment, each operation is applied n times in the order defined above. For the case where $d = NA$, operation AN is first applied n times on the initial graph G: this creates n new nodes. Then, AE is

[3] All numerical results of the experiments presented in this paper have an error margin of $\pm 1.000 \times 10^{-1}$ seconds because of the resolution of the timers.

Data	Operation	10	20	50	70	100	200	500	700	1,000	2,000	5,000	7,000	10,000	20,000	50,000	70,000	100,000	120,000	150,000	170,000	200,000
No attributes	Add nodes	8.9	14.0	35.0	37.1	73.7	90.2	286.7	242.1	296.5	657.3	1467.2	1741.4	1002.5	1425.1	761.7	1161.6	1046.4	1170.6	982.4	1270.2	1347.9
No attributes	Add edges	0.9	1.8	4.1	5.2	6.7	10.6	15.2	15.6	16.2	18.6	20.0	20.8	20.3	20.7	28.9	27.0	27.1	28.3	28.6	29.7	30.2
No attributes	Update nodes	2.5	4.5	11.0	14.0	19.6	33.0	68.4	67.1	77.6	118.5	209.6	202.1	153.5	161.2	137.3	137.7	121.7	132.2	135.3	136.9	146.1
No attributes	Traverse	1.0	1.0	1.5	1.8	3.3	2.5	1.7	1.6	1.7	2.0	1.6	2.0	1.8	1.6	2.6	2.5	2.3	2.4	2.4	2.5	2.7
No attributes	Delete edges	0.2	0.4	1.0	1.3	1.9	3.6	7.9	10.3	13.5	23.5	42.2	52.5	58.5	56.1	56.4	49.3	46.5	45.8	43.7	43.8	44.9
No attributes	Delete nodes	2.2	4.2	9.5	12.1	15.0	25.8	40.3	45.3	50.7	63.3	75.8	79.3	83.2	74.2	68.5	59.9	53.3	54.0	54.3	51.6	51.5
Attributed	Add nodes	6.0	3.5	2.0	1.7	1.5	1.2	1.1	1.1	1.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Attributed	Add edges	1.0	1.8	3.7	4.8	6.2	9.6	14.3	14.6	16.0	18.1	20.7	20.5	21.0	19.5	33.1	31.3	42.8	40.4	59.3	55.5	52.8
Attributed	Update nodes	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Attributed	Traverse	0.9	0.9	1.3	2.0	3.4	2.7	1.7	1.7	1.6	2.2	1.7	2.1	1.9	1.9	3.3	4.1	3.6	4.3	4.0	3.9	4.1
Attributed	Delete edges	0.3	0.4	0.9	1.4	1.8	3.4	7.8	10.2	13.0	23.1	42.7	50.4	57.9	54.0	55.2	47.0	43.8	43.9	43.2	42.8	41.2
Attributed	Delete nodes	1.8	3.1	5.8	7.1	7.6	8.7	7.3	6.9	6.5	5.7	5.0	4.7	4.5	4.1	4.2	4.3	4.3	4.2	4.2	4.2	4.2

Figure 4.1: Relative performance of IGraph and NetworkX for CRUD operations. The darker the colour, the better IGraph performs.

applied n times, which adds n new edges in the new graph. UN updates the attribute values of n nodes in the graph and T traverses the whole graph in a breadth-first fashion. After these four operations are applied, n edges are deleted from the graph. Finally, DN deletes n nodes from the resulting graph. We chose to apply the operations in this order to avoid dependencies and deletion which make no sense, and thus do not bias the computation times. Similar experiments were performed for the case where $d = AT$. The system running these experiments is composed of 32 nodes equipped with an Intel Core 2 Duo processor of 2.66 GHz, 8 GB of memory with a 667 MHz DDR2, and two times 4MB of L2 cache.

Analysis of the Comparison

At a first glance, Figure 4.1 shows many more dark cells then light ones, indicating that overall, IGraph performs better than NetworkX. For a graph with no attributes (when $d = NA$), since IGraph stores nodes very efficiently as explained previously, it clearly outperforms NetworkX with respect to the creation of elements: 1.3×10^3 times faster for the creation of 2×10^5 nodes and 30 times faster for the creation of the same number of edges. NetworkX is up to 4 times faster for deleting a small number of edges (less than 50), while IGraph is up to 45 times faster for larger values of n. As for the deletion of nodes, IGraph is up to 80 times faster for mid-sized graphs (10^4 edges) and around 50 times faster for larger graphs. This ratio of computation time r is significantly smaller than for the creation of nodes because of the re-numbering required to ensure a continuous numbering of nodes in IGraph. Traversing all nodes in the graph is twice as fast on average in IGraph for any size of the graph. The update operation ($op = UN$) in this case is simply the sum of adding and removing the same number of nodes n, since no attributes are stored in the nodes. On average this operation is 100 times faster for IGraph.

Now that we know IGraph is significantly more efficient than NetworkX for non-attributed graphs, we will examine whether this is still the case when attribute data is added to the graph. When $d = AT$, the creation of nodes is about 3 times faster in IGraph than in NetworkX for $n \leq 100$. In both libraries, creating a larger number of nodes is as fast as the initialization of attributes becomes an overhead on the actual creation of a node. This is confirmed by the fact that the update operation is equally fast in

Operation	No Attributes		Attributed	
	Average	Standard deviation	Average	Standard deviation
Add nodes	719.9	583.3	1.5	1.2
Add edges	17.9	9.8	23.2	18.0
Update nodes	99.5	65.5	1.0	0.0
Traverse	2.0	0.6	2.5	1.2
Delete edges	28.7	22.8	27.8	22.1
Delete nodes	46.4	25.5	5.2	1.7

Table 4.1: Average (first column) and standard deviation (second column) over all values of n of the performance ratios of IGraph over NetworkX.

both libraries. The ratio for edge creation is the same as for the non-attributed case for $n \leq 2 \times 10^4$. This is predictable as the presence of node attributes does not influence edge creation. However, when creating more edges, IGraph is slightly even more efficient: the ratio is 1.5 times higher than for the non-attributed case. Edge deletion for the attributed case performs as well as its non-attributed counterpart. Node deletion becomes only 4 times faster with IGraph when attributes are present. NetworkX is slower traversing the graph with attributed nodes (up to 4 times slower for graphs with 2×10^4 nodes). Table 4.1 summarizes the overall comparison. The averages were computed based on the data captured in each row of Figure 4.1.

4.3 Optimal Representation of Models

Given the results of the previous experiment, IGraph is overall significantly more time-efficient. It is also more space-efficient since the machines running the NetworkX library ran out of memory at $n = 3 \times 10^5$, while no thrashing was observed when IGraph was dealing with graph sizes of up to 10^6 elements. In this section, we investigate the optimal representation of typed attributed graphs. We analyse the performance and relative cost of the CRUD operations.

4.3.1 Representing Models as Directed Simple Graphs

Models are abstractions of pertinent aspects of a system. An important class of models is those where *entities* represent the concepts and data of the model and *relations* describe how these concepts are related. Moreover, a relation may itself hold data. When such models are realized as directed graphs, representing entities as nodes and relations as edges seems obvious at a first glance. IGraph supports attribute assignment on both elements. In specific cases, a relation may itself be related to another relation or entity[4]. But then the graph representation would require one to consider such relations as nodes. Therefore, to uniformly represent entities and relations of a model, we propose that they be represented as nodes. Thus, a graph edge represents the link between an entity and a relation, a

[4]For example, in UML class diagrams [Obj09], an association class can relate two classes and also be part of an inheritance relationship with another association class.

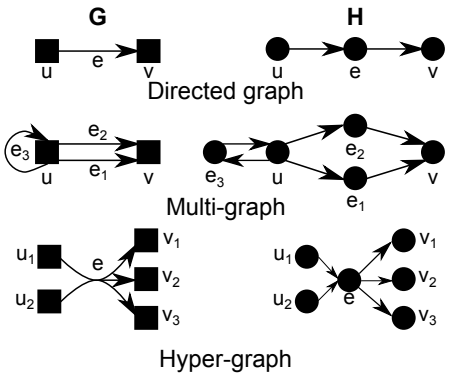


Figure 4.2: Different types of graphs G and their representation as Himesis graphs H.

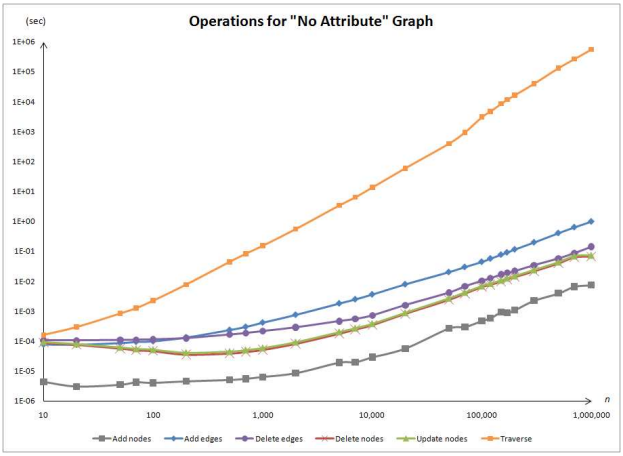
relation and an entity, or a relation and a relation. Hence, attributes need only be stored on nodes. Another advantage of this uniform representation of models becomes apparent when a model has a multi-graph or hyper-graph topology. Figure 4.2 describes how the relations are represented in each case. However, our representation does not consider multi-hyper-graphs, which are not common in MDE.

Now let us examine the cost of this uniform representation of a model M. Let $G = (V(G), E(G))$ denote a directed graph representing the entities of M as nodes $V(G)$ and its relations as edges $E(G)$. Let H denote the graph representing the entities and relations of M as nodes $V(H)$ and the links as edges $E(H)$. Examples of G and H are depicted in Figure 4.2. We then have that $|V(H)| = |V(G)| + |E(G)|$ and $|E(H)| = 2 \times |E(G)|$. Therefore there is only a constant difference $|E(G)|$ between the size of H and G. This is also the case when G is a multi-graph. When G is a hyper-graph, $|V(H)|$ is as before but now $|E(H)| = |Src(E(G))| + |Tar(E(G))|$, where *Src* and *Tar* represent respectively the source nodes and target nodes of all edges.

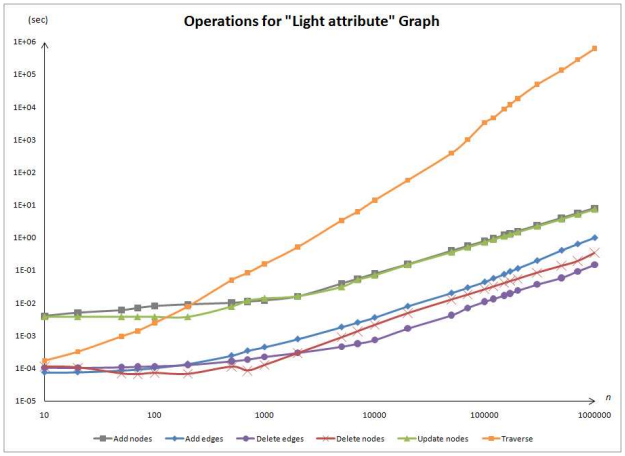
4.3.2 Performance Evaluation of CRUD Operations

To investigate the optimal representation of data, we need to modify the domain of d in the condition tuple (d, op, n) such that: $d \in \{NA, LA, HA, LO, HO\}$, respectively *No Attribute*, *Light Attribute*, *Heavy Attribute*, *Light Object*, and *Heavy Object*. They span two dimensions: data representation and the size of the data. The "attribute" label indicates that, for each node, data is stored as a separate node attribute. The "object" label indicates that all the data is wrapped in a single object and only that object is stored as a node attribute. In our experiments, a light attribute is 139 bytes, whereas a heavy attribute is 4,330 bytes. The first corresponds to the size of two integers and three characters in Python. The second corresponds to the size of two integers, two 50-character-long strings and another string of 4,094 characters long.

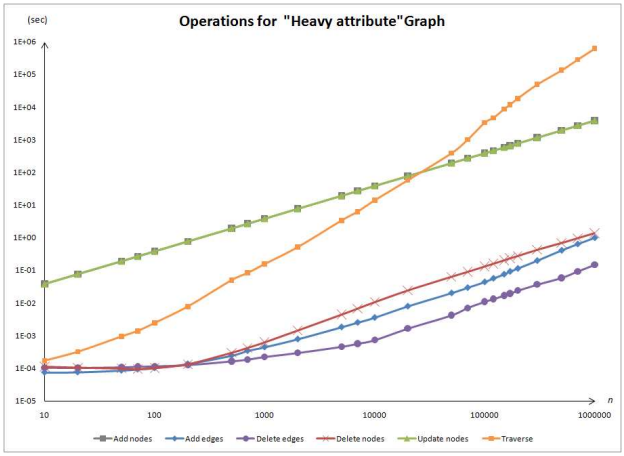
Figure 4.3 represents the time performance of IGraph for each value of d. When no data is stored in the graph, *i.e.*, $d = NA$ (Figure 4.3(a)), node creation is the least costly operation with less than 10 milliseconds for adding 10^6 nodes. Node deletion is also very efficient with 100 milliseconds for the same amount of nodes. As mentioned before, for the case where $d = NA$, the update node operation



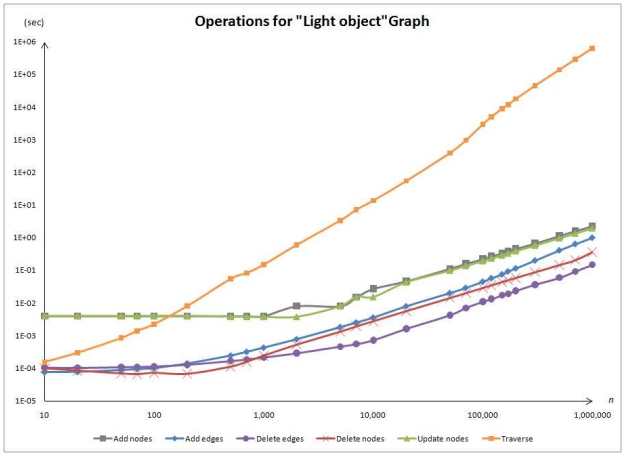
(a)



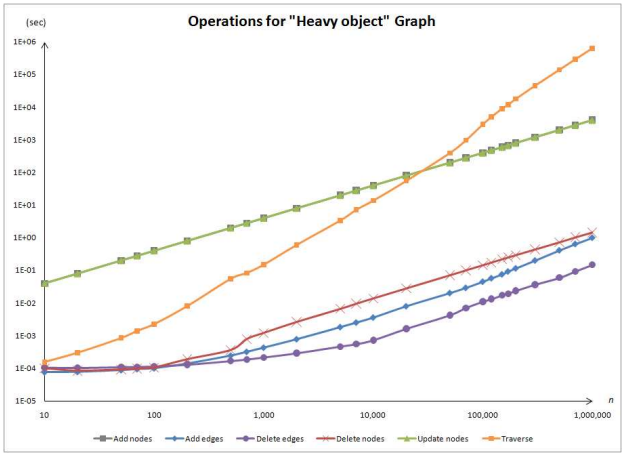
(b)



(c)



(d)



(e)

Figure 4.3: The effect of data representation. The graphs are plotted on a log-log scale.

Operation	No attribute			Light attribute			Light object			Heavy attribute			Heavy object		
	Small $[0,10^3[$	Medium $[10^3,10^5[$	Large $[10^5,10^6]$	Small $[0,10^3[$	Medium $[10^3,10^5[$	Large $[10^5,10^6]$	Small $[0,10^3[$	Medium $[10^3,10^5[$	Large $[10^5,10^6]$	Small $[0,10^3[$	Medium $[10^3,10^5[$	Large $[10^5,10^6]$	Small $[0,10^3[$	Medium $[10^3,10^5[$	Large $[10^5,10^6]$
Add nodes	5.E-06	2.E-04	5.E-03	2.E-03	2.E-01	3.E+00	7.E-04	7.E-02	8.E-01	6.E-03	2.E-01	3.E+00	5.E-03	7.E-02	8.E-01
Add edges	2.E-04	2.E-02	3.E-01	2.E-04	2.E-02	5.E-01	2.E-04	2.E-02	5.E-01	2.E-04	2.E-02	3.E-01	3.E-04	3.E-02	3.E-01
Update nodes	1.E-04	2.E-02	2.E-01	2.E-03	2.E-01	2.E+00	5.E-04	5.E-02	7.E-01	2.E-03	2.E-01	3.E+00	5.E-04	6.E-02	7.E-01
Traverse	2.E-02	4.E+02	2.E+03	5.E-02	6.E+02	3.E+05	5.E-02	5.E+02	3.E+05	3.E-02	4.E+02	2.E+03	9.E-03	6.E+02	2.E+03
Delete edges	2.E-04	4.E-03	3.E-02	2.E-04	5.E-03	8.E-02	2.E-04	5.E-03	8.E-02	2.E-04	5.E-03	3.E-02	2.E-04	5.E-03	3.E-02
Delete nodes	4.E-05	3.E-03	4.E-02	3.E-05	2.E-03	2.E-02	3.E-05	1.E-03	1.E-02	1.E-04	8.E-03	1.E-01	1.E-04	5.E-03	6.E-02

Figure 4.4: Average times in seconds for executing CRUD operations.

is evaluated as first deleting then adding a new node to replace it. This is why it takes about the same time as the delete node operation. Edge deletion seems to perform faster than edge creation for larger graphs with respectively 100 milliseconds versus 1 second. Node traversal is undoubtedly the most costly operation. IGraph can traverse 2×10^4 nodes within a minute, but it takes almost 6.5 days to traverse 10^6 nodes in breadth-first search! This poor performance is due to the internal implementation of the IGraph version used. However, in model transformation it is rarely the case that the whole graph must be traversed, since it is often a single match that is requested. Nevertheless, requiring all matches may in the worst case lead to multiple traversals of the graph. We plan to solve this issue in the future.

The plots for $d = LA$ and $d = LO$ are very similar to each other. This indicates that the relative performance of the operations is the same when light data is stored in the graph (Figure 4.3(b) and 4.3(d) respectively). Edge operations become the fastest. Nevertheless, from $n = 2 \times 10^4$ onwards, node deletion performs better than edge creation but worse than edge deletion, both by a factor of 2. Node creation is now slower by a factor 10^3 in the case of $d = LA$ and by a factor of 5×10^2 in the case of $d = LO$. Moreover, node update takes about the same time as node creation, which confirms that the setting of attribute values is an overhead for node creation. Traversal is still the most costly operation.

Finally, the plots for $d = HA$ and $d = HO$ are also very similar. The exceptions are that node deletion is now more expensive than edge creation with 1.7 seconds for $n = 10^6$. Also, node creation (and thus the update operation) is more costly than the traversal operation up to $n = 2 \times 10^4$ nodes.

To better illustrate the described results, the table in Figure 4.4 presents the average performance time of each operation for different sizes of the graph. The graphs are grouped in three categories. Small graphs (less than 10^3 nodes) are typically used for small examples or debugging purposes. Medium graphs (between 10^3 and 10^5 nodes) are considered as large graphs for academics but average size for industrial projects. Large graphs (more than 10^5 nodes) are typically used in large industrial applications such as mobile networking. Furthermore, the table in Figure 4.5 summarizes the impact of choosing the “attribute” or the “object” representation for data in the graph. It clearly shows that the “object” approach is more efficient than the “attribute” approach.

The plots in Figure 4.6 compare the time performance of the CRUD operations for each representation of data.

Add Nodes. From Figure 4.6(a), node creation is polynomial with a powers of 10^{-8}, 10^{-5}, and 10^{-3} for the case where there are no attributes, for light attributes, and for heavy attributes respectively.

Operation	LO/LA	HO/HA	HA/LA	HO/LO
Add nodes	3.E-01	5.E-01	2.E+00	3.E+00
Add edges	1.E+00	1.E+00	9.E-01	1.E+00
Update nodes	3.E-01	3.E-01	1.E+00	1.E+00
Traverse	1.E+00	1.E+00	4.E-01	4.E-01
Delete edges	1.E+00	1.E+00	8.E-01	9.E-01
Delete nodes	7.E-01	7.E-01	4.E+00	4.E+00

Figure 4.5: Effect of using IGraph's node-level attribute mechanism for each node attribute individually compared to wrapping all attributes in one object stored using IGraph's node-level attribute mechanism.

Add/Delete Edges. From Figure 4.6(b), edge creation is independent of the data representation and size. The log-log relation is in fact quadratic in n. Edge deletion is also independent of the data representation as shown in Figure 4.6(c).

Delete Nodes. From Figure 4.6(d), node deletion is quadratic in n. Here we see that the "attribute" representation is slightly more optimal for small to medium-sized graphs by 30%.

Update Nodes. In Figure 4.6(e), updating light data represented in the "attribute" approach is 30 times slower than the "object" approach. As for heavy-weight data, either approach is as slow by a factor 10^3.

Traverse. Finally, from Figure 4.6(f), traversal of the graph is independent of the data representation and size. The plotted graphs are quadratic reflecting the traversal's complexity.

4.3.3 Optimal Representation of Data of Models

The previous experiment considered graphs in general. In the following experiment, we investigate an optimal representation of attributes of AToMPM models. Elements of these models can hold an arbitrary number of attributes. A typical element of an AToMPM model includes the following data: a universally unique id, two integers, two booleans, two 50-character long strings, an additional 10-character long string encoding the type of this element, a 1000-character long string representing an action or constraint on the element (typical for elements of transformation models), and a list of seven 10-character long strings enumerating all the sub-types of the type of this element. The total size of this typical element is thus 1,382 bytes, which is an average size according to the experiment in Section 4.3.2.

We now consider three different alternatives for representing data in the nodes of IGraph graphs:

- Node attribute mechanism used for each of the above attributes (this is the *AT* approach used previously).
- A Python object encapsulating all the attributes, stored as one node attribute (this is the *OT* approach used previously).

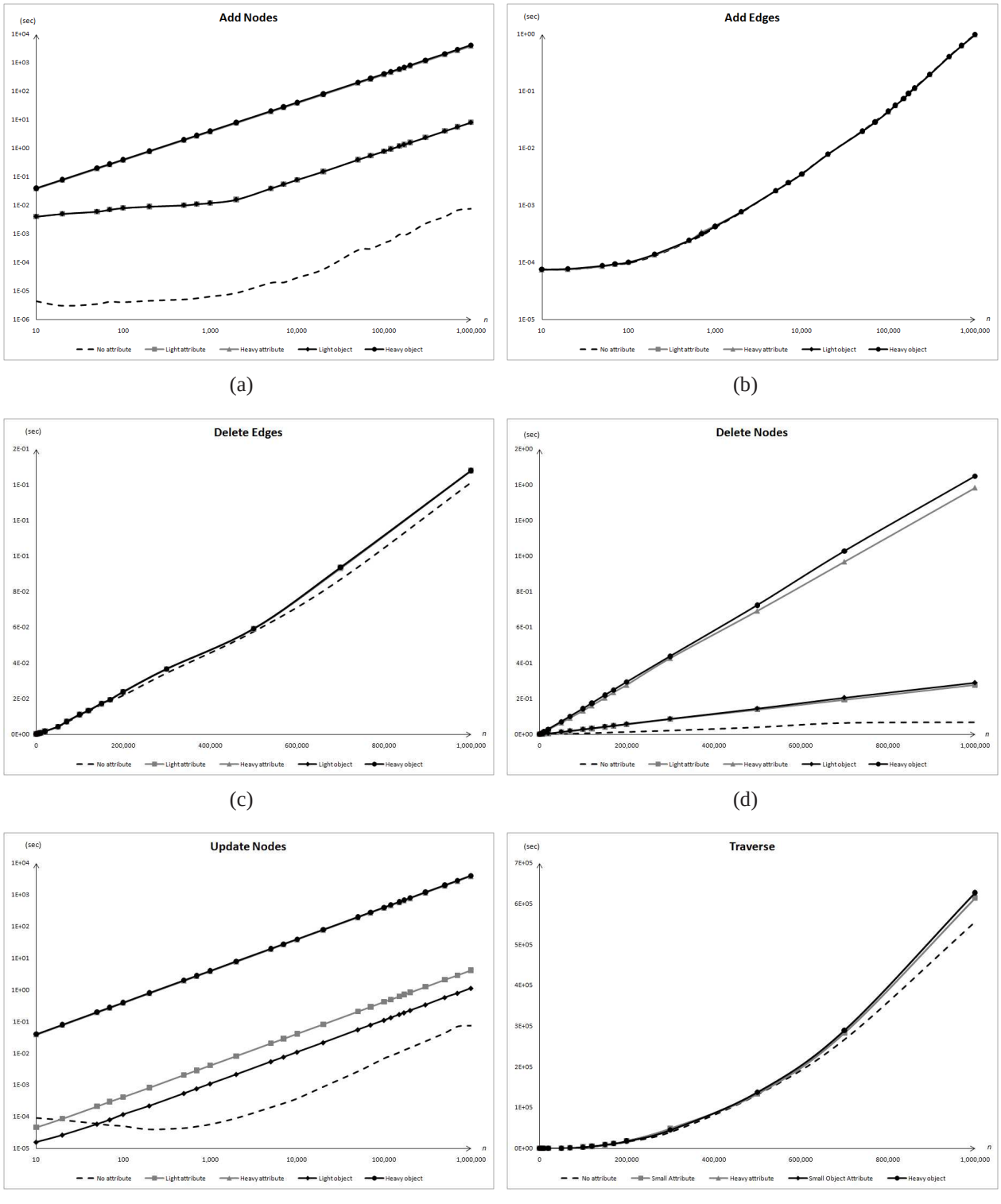
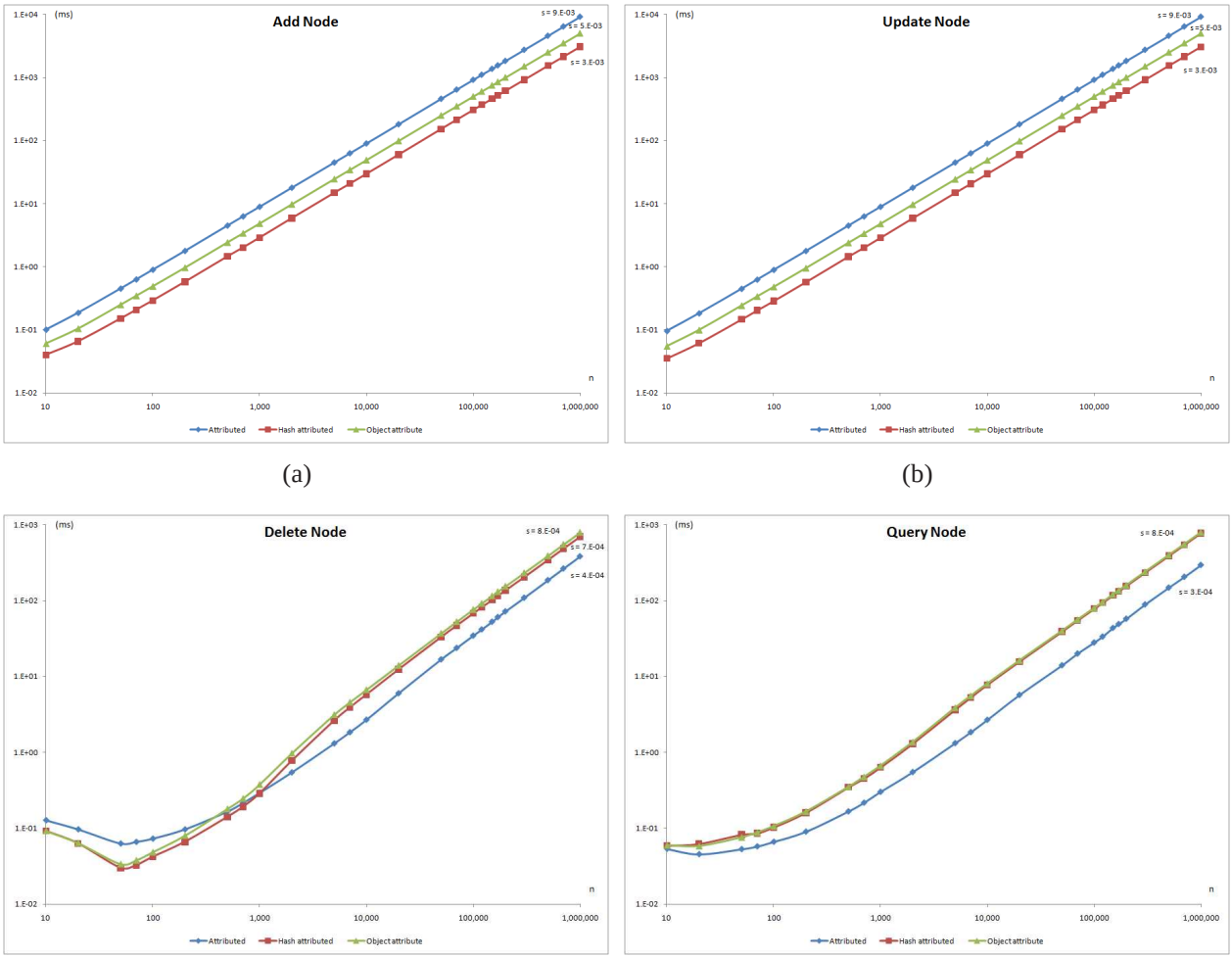


Figure 4.6: CRUD operations on nodes for each representation of data. Plots (a) and (b) are on a log-log scale.



(a) (b)

(c) (d)

Figure 4.7: CRUD operations on nodes for each representation of data. The plots are on a log-log scale.

- A hash table holding all the attributes, stored as one node attribute (this will be referred to as HT).

In order to determine which of AT, OT, or HT is the optimal representation to use in Himesis, we evaluate their performance on CRUD operations applied to nodes only, since Section 4.3.2 has confirmed that data stored in nodes has no impact on the performance of edge operations.

Create Nodes. Figure 4.7(a) shows the time orders of magnitude for creating[5] nodes for each representation: 9×10^{-3} for AT, 5×10^{-3} for OT, and 3×10^{-3} for HT.

[5]Addition of nodes and initialization of their attributes.

Update Nodes. Figure 4.7(b) shows the time orders of magnitude for updating nodes for each representation: 9×10^{-3} for AT, 5×10^{-3} for OT, and 3×10^{-3} for HT. Not surprisingly, this is the same order as for adding nodes since, according to Section 4.3.2, the addition of nodes takes significantly less time than initializing its attributes (about 10^3 times faster).

Delete Nodes. Figure 4.7(c) shows the time orders of magnitude for deleting nodes for each representation: 4×10^{-4} for AT, 8×10^{-4} for OT, and 7×10^{-4} for HT.

Query Nodes. To query nodes, we have investigated the optimal way of retrieving the data from nodes: using the mechanism built in IGraph for querying nodes (the `select` method) or programmatically retrieving attribute values (in a loop). The results were very conclusive: using the IGraph query mechanism for HT and OT is 1.6 times faster than the programmed loop, and 3.1 times faster if using the IGraph query mechanism for AT. Thus we only consider the IGraph mechanism for querying nodes. Figure 4.7(d) shows the time scales for querying nodes for each representation: 3×10^{-4} for AT and 8×10^{-4} for both OT and HT.

We would like to minimize the time each of the CRUD operations takes in a rule. Here, we assume that a rule consists of a LHS pre-condition pattern graph and a RHS post-condition pattern graph. After performing a regression analysis of the plots in Figure 4.7, we have computed the slopes of each curve and added them as labels in the figure. With these observations, we can write the following formulas representing the time cost of a rule application for each representation of data:

$$AT: \quad 90a + 90u + 4d + 3q \tag{4.1}$$
$$OT: \quad 50a + 50u + 8d + 8q \tag{4.2}$$
$$HT: \quad 30a + 30u + 7d + 8q, \tag{4.3}$$

where a, u, d, and q are the number of times the add, update, delete, and query operations[6] on nodes happens in a rule, respectively. Therefore, choosing the optimal representation depends on the solution of the following inequalities:

$$\text{Choose } HT \text{ over } OT \quad \Leftrightarrow \quad 20(a+u) + d > 0 \tag{4.4}$$
$$\text{Choose } AT \text{ over } OT \quad \Leftrightarrow \quad q > 8(a+u) - 0.8d \tag{4.5}$$
$$\text{Choose } AT \text{ over } HT \quad \Leftrightarrow \quad q > 12(a+u) - 0.6d \tag{4.6}$$

Equation (4.4) is obtained by reducing the inequalities (4.3) smaller than (4.2). The remaining equations are obtained in a similar way. Equation (4.4) is always true since, by definition, a rule applies at least one of the add, update, or delete operations. Hence OT will not be considered anymore and equation (4.5) can be discarded. The left-hand side of (4.6) represents the operation performed in the matching phase of the rule (querying nodes). The right-hand side of (4.6) represents the operation performed in the rewriting phase of the rule (add, update, delete nodes). Recall that the matching phase queries all nodes of the pre-condition pattern as well as all nodes of the source graph G

[6]The update and query operations are performed on all the attributes of each node.

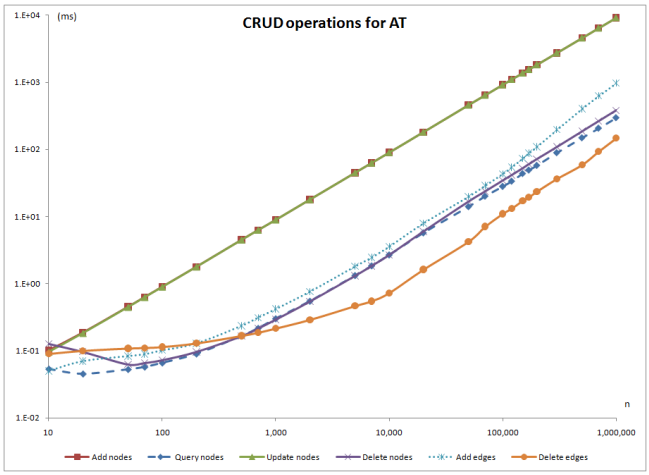
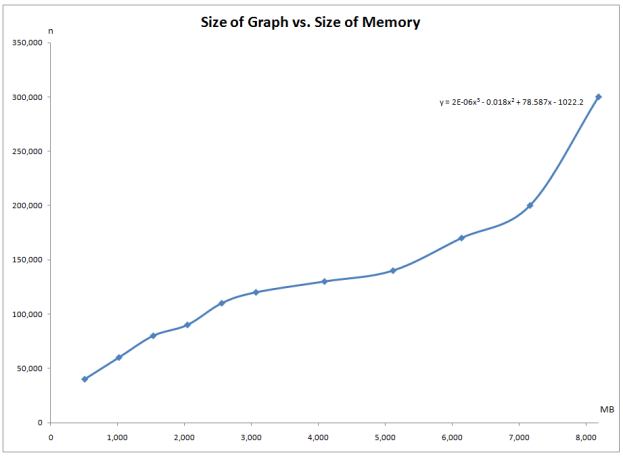


Figure 4.8: Performance of all operations on Himesis graphs. The graph is plotted on a log-log scale.

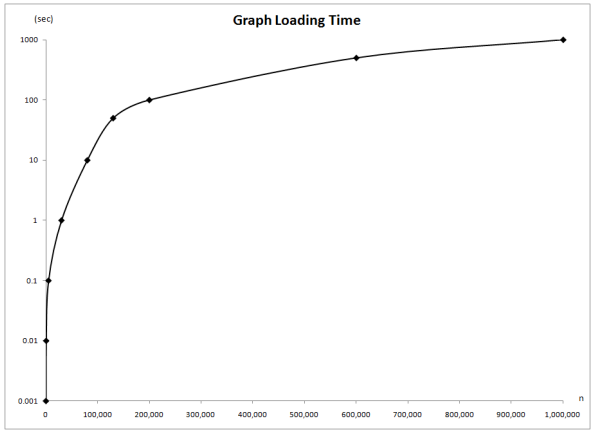
(in the worst case). Hence $q \in O(|V(LHS)| + |V(G)|)$. Following a similar reasoning, we have that $a \in O(|V(RHS - LHS)|)$, $u \in O(|V(LHS \cap RHS)|)$, and $d \in O(|V(LHS - RHS)|)$. On the one hand, in the extreme case where the LHS is empty, we therefore have that $|V(G)| > 12.6|V(RHS)|$. On the other hand if the RHS is empty, then $|V(G)| > -13.6|V(LHS)|$, which is always true if both the LHS and G are not empty. Therefore, a sufficient condition for choosing the *AT* approach is if there are 13 times more nodes in the source graph than in the RHS. This is very likely to hold given that relation-like model elements are also represented as nodes in Himesis. Moreover, favouring the *AT* approach reduces attribute access time for other model manipulations as well. Figure 4.8 classifies the performance of each graph operation performed on a Himesis graph implemented with the *AT* approach.

4.3.4 Memory Requirements for Himesis Graphs

Models used in industry typically hold a significant amount of data. One issue that may arise is how much physical memory is required to hold these models. Figure 4.9(a) shows the amount of physical memory required for loading a Himesis graph in memory. The measures were obtained with an average node size of 1,382 bytes as before. The values are computed as follows. For a fixed amount of physical memory, we attempt to load a graph and increase the size of the graph until IGraph starts thrashing. This maximum size is recorded every time. Each data point of Figure 4.9(a) thus represents the average over 100 repetitions of this experiment. Interestingly, the size of the graph follows a cubic function with respect to the minimum amount of memory needed. Although the plot only shows graphs of size up to 3×10^5 nodes, it can still load graphs with 10^6 nodes but it requires virtual memory and hard disk as secondary memory storage. Figure 4.9(b) illustrates the time required to load a Himesis graph in memory. It takes less than a second to load a graph with 8×10^4 nodes and almost a minute for 2×10^5.



(a) Physical memory required for loading a Himesis graph.



(b) Time for loading a Himesis graph.

Figure 4.9: Measuring the effect of memory.

4.4 Match and Rewrite Operations in Himesis

Model transformation plays a crucial role in model-driven development. A transformation is commonly expressed as a set of *transformation rules*. A rule consists of a pre-condition pattern and a post-condition pattern; both are *T-Core* primitives. The former describes a pattern that should occur in the input model and the latter describes how this occurrence shall be modified. When models are implemented as graphs, the pre-condition pattern specifies that an instance of this pattern must be a sub-graph of the input graph. Pattern matching and, in particular the sub-graph homomorphism problem, is NP-complete [Meh84]. There are however various exponential-time worst case solutions for which the average-time complexity can be reduced with the help of heuristics. These approaches can be divided into two major categories: *search plans* and *constraint satisfaction problems* (CSP).

Search plan techniques [Zün94, GBG+06] define the traversal order for the nodes of the model to check whether the pattern can be matched. This is done by computing the cost tree of the different search paths and choosing the least costly one. Complex model-specific optimization steps can be carried out for generating efficient adaptive search plans [VVF05]. Examples of such heuristics are the use of typing information with respect to meta-model elements or the use of cardinality constraints defined in the meta-model.

Graph pattern matching can also be described as a constraint satisfaction problem [Rud98], where the pre-condition elements are variables, the elements of the model form the domain and typing, and the links and attribute values form the set of constraints. These techniques make use of backtracking algorithms [KH04] for finding a sub-graph of the input graph that is isomorphic[7] to the pre-condition graph. The algorithm explores the search space in a depth-first order. Well-known algorithms such as Ullmann [Ull76] and VF2 [CFSV04] are some of the most efficient for solving the sub-graph

[7] In fact, it is homomorphic since the added attribute constraints in the pattern graphs describe constraints on the attributes of the source graph.

isomorphism problem as a constraint satisfaction problem. Let us first explore these two algorithms.

4.4.1 Ullmann

Ullmann's algorithm [Ull76] is an efficient solution to the sub-graph isomorphism problem. Given two undirected graphs $H = (V_H, E_H)$ and $G = (V_G, E_G)$, the Ullmann algorithm tests whether H is a sub-graph of G. We denote by $\mathbf{H}$ and $\mathbf{G}$ their respective adjacency matrices. Let $\mathtt{deg} : V \to \mathbb{N}$ be a function mapping a vertex to its degree: the number of incident edges it is connected to ($\mathbb{N}$ represents here the set of non-negative integers). The algorithm first constructs the $|V_H| \times |V_G|$ binary matrix $\mathbf{M}^*$ such that:

$$\mathbf{M}^*_{vw} = \begin{cases} 1 & \text{if } \mathtt{deg}(v) \leq \mathtt{deg}(w) \text{ for } v \in V_H \text{ and } w \in V_G, \\ 0 & \text{otherwise.} \end{cases}$$

In this notation, $\mathbf{M}^*_{vw}$ denotes the element of $\mathbf{M}^*$ at the row corresponding to node v and the column corresponding to node w. $\mathbf{M}^*$ therefore represents the matching of all possible node candidates of V_G that are isomorphic to nodes of V_H. The algorithm tries to find a matrix $\mathbf{M}$ such that $\mathbf{M}(\mathbf{MG})^T = \mathbf{H}$ where every row has exactly one 1 and every column has at most one 1. Therefore $\mathbf{M}$ represents the isomorphic mapping of vertices of H to G. The algorithm thus enumerates all possible such matrices, starting from $\mathbf{M}^*$. At each step, a node in V_H (row of $\mathbf{M}$) is assigned one of the matches (in decreasing order of degree) by setting to 1 the appropriate column and the rest to 0. This depth-first search is optimized with a *refinement procedure* that takes into account neighbouring nodes: a node V_G may only match if all its neighbours also match. This may set other elements of the matrix to 0, hence reducing the search space. After refining $\mathbf{M}$, if there is a row with no 1, the algorithm backtracks and the next potential match is tried. Otherwise, the algorithm continues on the next row of $\mathbf{M}$. Repeated recursively, the algorithm terminates when either a complete match is found or if all possible matches have been exhausted.

Time efficiency depends highly on how sparse $\mathbf{M}^*$ is initially. Because in graph transformation we consider typed, attributed, labelled, directed graphs as representing models, the number of $\mathbf{M}$ matrices generated through the search is much smaller than in the general case. This requires comparing incoming and outgoing edges, attribute values, and type compatibility to appropriately fill $\mathbf{M}^*$ with 1s. Some approaches also extend the test of the degree of the node to more sophisticated compatibility tests.

4.4.2 VF2

VF2 [CFSV04] is yet another algorithm for the sub-graph isomorphism problem. Like in Ullmann's approach, VF2 constructs a search-tree traversing the host graph depth-first and backtracks when the current search-state fails a compatibility test. The algorithm also performs a pruning of the search space during the matching process.

Consider H and G as directed graphs. We denote $M : V_H \to V_G$ to be the isomorphic node mapping and $M(s)$ holds the set of current matches (v_G, v_H) at search state s (the dashed lines in Figure 4.10 linking the black nodes). Then let $M_H(s)$ and $M_G(s)$ respectively represent the nodes of V_H and V_G

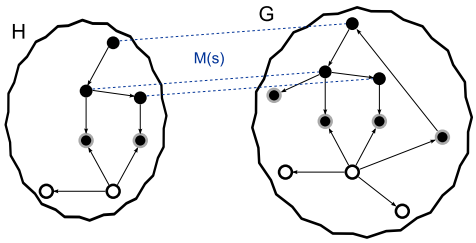


Figure 4.10: Partial sets for the pruning technique of VF2

contained in $M(s)$. They are respectively the black nodes in H and in G. VF2 then considers the neighbourhood of $M_H(s)$ by defining $N_H(s)$ and $\overline{V}_H(s)$ (respectively the highlighted and white nodes in Figure 4.10). $N_H(s) = N_H^{in}(s) \cup N_H^{out}(s)$ where $N_H^{in}(s)$ is the set of nodes adjacent to $M_H(s)$ along incoming edges and $N_H^{out}(s)$ is the set of nodes adjacent to $M_H(s)$ along outgoing edges, not yet in the partial mapping $M_H(s)$. $\overline{V}_H$ is defined as $\overline{V}_H(s) = V_H - M_H(s) - N_H(s)$, representing the nodes not connected to the current mapping. Similar expressions hold for $N_G(s)$ and $\overline{V}_G(s)$.

At each step of the depth-first search, the search-state s is augmented by a candidate pair $p = (v_p, w_p)$ only if it passes a *feasibility test*. (v_p, w_p) is chosen from the ordered list $P(s)$ of candidate pairs. The order suggested by VF2 gives priority to nodes in N_H^{out} and N_G^{out}, then in N_H^{in} and N_G^{in}, and finally (in case of unconnected graphs) in $\overline{V}_H(s)$ and $\overline{V}_G(s)$. The feasibility test run on $s' = s \cup p$ tests three criteria in this order:

1. if the new mapping $M(s')$ is still a valid isomorphism, *i.e.*, edges between v_p and its adjacent nodes in $M_H(s')$ and edges between w_p and its adjacent nodes in $M_G(s')$ correspond,
2. if the number of external edges between $M_H(s')$ and $N_H(s')$ is *smaller than or equal* to the number of external edges between $M_G(s')$ and $N_G(s')$,
3. if the number of external edges between $N_H(s')$ and $\overline{V}_H(s')$ is *smaller than or equal* to the number of external edges between $N_G(s')$ and $\overline{V}_G(s')$,

This way VF2 reduces the search space and ensures that no incompatibilities will occur in future search steps. If p fails the feasibility test, the procedure backtracks to the previous state s and tries another candidate. The algorithm terminates when $M(s)$ covers all the nodes of H (success) or when all candidate pairs of $P(s)$ have been tried (failure).

Efficiency-wise, experimental results show that VF2 performs better than Ullmann for larger graphs. Considering $N = |V_H| + |V_G|$ search states to visit in the best case, the time complexity of VF2 is $\Theta(N^2)$. In the worst case, there are $N!$ search states, leading to a time complexity of $\Theta(N!N)$. In both cases, VF2 is a linear order of magnitude more efficient than Ullmann. Furthermore, its spatial complexity is linear, while cubic in the case of Ullmann. These measurements are based on a benchmark [FSV01] considering 10^4 sub-graph matching experiments and are hence deduced from empirical results.

The major difference between Ullmann and VF2 is that, within one backtracking step, Ullmann compares pairs of adjacent nodes, while VF2 compares a node with its neighbourhood. Moreover,

Ullmann's $\mathbf{M}^*$ matrix verifies the semantic compatibility between pairs of nodes in the match, while VF2's feasibility test ensures a correct structure of the match. A combination of VF2 and Ullmann for hierarchical graphs was proposed in [Pro05]. The idea was to merge the two search plans providing containment edges and local edges to denote hierarchy. The time complexity was thus improved.

4.4.3 An Efficient Sub-graph Isomorphism Algorithm

The matching algorithm of Himesis combines our own variation of the VF2 algorithm together with the refinement strategy of Ullmann's algorithm, as outlined in Algorithm 14. The procedure `extend` augments the state of the algorithm with all possible mappings from the pattern graph to the source graph. In the following, we call a *mapping* the one-to-one correspondence between a pattern node and a source node. We denote by a *match* the set of mappings in which all source nodes form a graph that is homomorphic to the pattern graph. Lines 4-14 recursively compute further mappings given the current state of the algorithm. The *state* stores the following information:

Algorithm 14 `extend`(state)

```
1:  if mappingIsComplete(state) then
2:      storeMatch(state)
3:  end if
4:  for p, s in suggestMapping(state) do
5:      if areCompatible(p, s) then
6:          if areSyntacticallyFeasible(p, s) then
7:              if areSemanticallyFeasible(p, s) then
8:                  state.storeMapping(p, s)
9:                  extend(state)
10:                 state.undoMapping(p, s)
11:             end if
12:         end if
13:     end if
14: end for
```

- M^P and M^S are the mapping sets holding the pattern nodes and the source nodes respectively in the current mappings,
- T^P_{out} and T^S_{out} hold the set of adjacent nodes to respectively M^P and M^S following outgoing edges, at any time;
- T^P_{in} and T^S_{in} hold the set of adjacent edges coming in respectively M^P and M^S following incoming edges, at any time;
- $T^P_{inout} = T^P_{out} \cap T^P_{in}$ and $T^S_{inout} = T^S_{out} \cap T^S_{in}$.

$T^P_{out}, T^P_{in}, T^P_{inout}$, and T^S_{inout} are called the *terminal sets*. Each step of the search computes a partial mapping of the nodes and verifies that it does not violate the topology of the pattern graph. `suggestMapping` suggests a potential mapping of a source node s with a pattern node p (the pair (p,s) is also known

as the candidate pair in [CFSV04]). The choice of the pair is done in the following order: first from $(T^P_{inout}, T^S_{inout})$, then from (T^P_{out}, T^S_{out}), then from (T^P_{in}, T^S_{in}), and finally from all other nodes.

Afterwards, `areCompatible` verifies if it is worth continuing this mapping. This is done by comparing the number of incident edges of s and p (this is known as the refinement step in [Ull76]). The compatibility check verifies that:

$$|Out(p)| \leq |Out(s)| \wedge |In(p)| \leq |In(s)| \tag{4.7}$$

where $In(n)$ and $Out(n)$ respectively represent the set of incoming and outgoing adjacent edges of a node n. This is similar to the refinement step of Ullmann's algorithm.

Then come the feasibility checks. `areSyntacticallyFeasible` ensures that the topology of the current mapping corresponds to a sub-graph of the pattern graph. This is done by looking at the number of incident edges when (p,s) is added to the current set of mappings (M^P and M^S).

Let $InOut(n) = In(n) + Out(n)$, for any node n,
let $Out_p = Out(p) \cap T^P_{out}$ and $Out_s = Out(s) \cap T^S_{out}$,
let $In_p = In(p) \cap T^P_{in}$ and $In_s = In(s) \cap T^S_{in}$,
let $All_p = M^P \cup T^P_{out} \cup T^P_{in}$ and $All_s = M^S \cup T^S_{out} \cup T^S_{in}$.

Then the following must be true to ensure syntactic feasibility of s and p:

$$\begin{gathered} \left|Out_p\right| \leq |Out_s| \wedge \left|In_p\right| \leq |In_s| \quad \wedge \\ \left|Out_p\right| + \left|In_p\right| + \left|InOut(p) - All_p\right| \leq |Out_s| + |In_s| + |InOut(s) - All_s| \end{gathered} \tag{4.8}$$

The last test ensures that the semantics of s corresponds to the semantics of p. In our case, semantic information of the nodes is encoded in their attributes, but the details of the function `areSemantically-Feasible` will be elaborated later on. When s and p satisfy all of the above conditions, (p,s) is considered a valid mapping and is stored in the state (line 8). The algorithm then continues looking for remaining mappings. When all valid mappings have been computed (lines 1-3), the corresponding match is stored. The algorithm backtracks to the previous state when either a complete match is found or if the current partial match (set of mappings in M^P and M^S) does not allow for any further valid mapping. Note that a nice property of this algorithm is that any state in the search tree is visited exactly once.

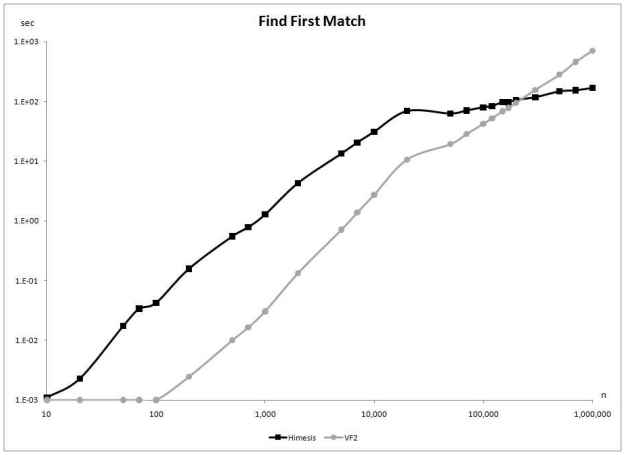
Algorithm 15 allows us to compute all matches between a pattern graph P and a source graph S. Furthermore, an initial set of mappings can be specified to prune the search tree constructed by the procedure `extend`. This initial mapping can also be seen as the initial context in which the matchings must be computed: it restricts specific pattern nodes to be mapped exactly to predefined source nodes.

Algorithm 15 `computeMappings`(S, P, context)

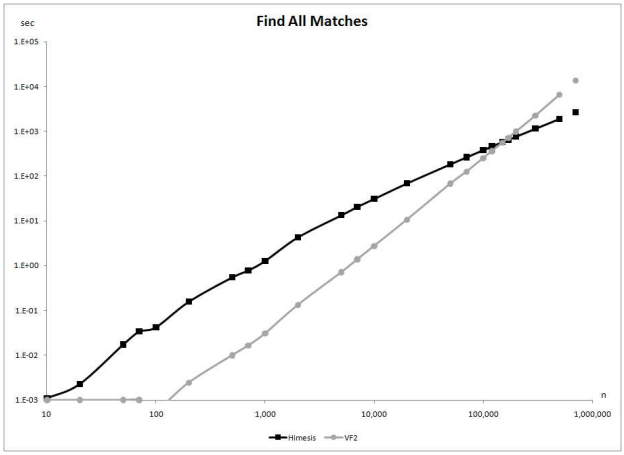
```
1: state ← initState(S, P)
2: for p, s in context do
3:     state.update(p, s)
4: end for
5: extend(state)
6: return state.getMatches()
```

Performance Evaluation of the Implementation

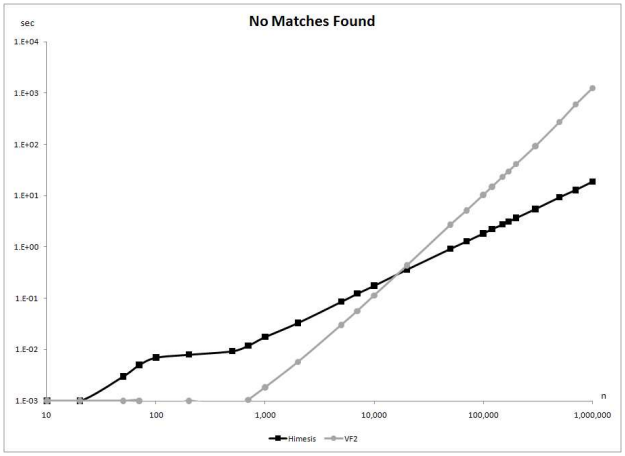
Let us first analyse the space complexity of the `extend` procedure. The state of the algorithm is encoded in the *state* variable. It holds the two partial mapping sets as well as the six terminal sets. Thus, the number of nodes stored in the state is at most $5 \times |V(P)| + 3 \times |V(S)|$ which is linear in terms of the nodes of the source and pattern graphs. Moreover, since IGraph stores the nodes as integers, *state* is quite compact. Additionally, the experiments below show that the algorithm performs better if the adjacency list (encoded as a hash table) is memoized as well. The size of this hash table is in the worst case $|V(P)|^2 + |V(S)|^2$ for fully connected, directed, simple graphs.



(a)



(b)



(c)

Nodes		Edges	Class	Attribute	Associations	Matches	All	First	None
Small	$[0,10^3[$	399	51	45	51	729	0.016	0.04	0.034
Medium	$[10^3,10^5[$	33,727	4,133	4,249	4,196	83,766	0.067	0.125	0.348
Large	$[10^5,10^6]$	304,552	38,158	38,312	37,919	891,171	2.929	5.006	22.305

(d)

Figure 4.11: Size average of sub-graph isomorphism matching over the six pattern graphs. The graphs are plotted on a log-log scale.

We now compare the time performance of the `extend` algorithm of Himesis with VF2's sub-graph isomorphism algorithm. We have chosen the IGraph implementation of VF2 as a benchmark which is in direct correspondence with the original implementation. Note that Himesis is implemented in

Python whereas VF2 was implemented in C. Anecdotal studies have shown that Python is in general slower than C by an average factor of 23 (c.f. [Ful10]), which is not integrated in the results presented here. In these experiments, we gathered the computation time with respect to the number of nodes n of the source graph. The source graph represents random valid class diagrams encoded as Himesis graphs. The average number of class diagram elements is shown in Figure 4.11(d). For each source graph we have run the algorithm on six pattern graphs whose sizes range from 2 to 12 nodes. Our experience shows that this is a typical size for LHS and NAC pre-condition patterns assuming an expressive control flow language such as *MoTif* [SV11]. For both the source and pattern graphs, the number of edges is the same order as the number of nodes (which is typical in class diagrams). Each data point of the plots in Figure 4.11 represents the average time over the six pattern graphs.

Figure 4.11(a) shows the performance of both algorithms for finding the *first* match only. For small graphs, VF2 is about 25 times faster than Himesis. For medium graphs, VF2 is twice as fast as Himesis. However, at around 2.2×10^5 nodes, both perform equally fast. At this point, Himesis overtakes VF2 by a factor of 6 for large graphs.

Figure 4.11(b) shows the performance of both algorithms for finding *all* matches. For small graphs, VF2 is about 60 times faster than Himesis. For medium graphs, VF2 is 5 times faster than Himesis. However at around 1.5×10^5 nodes, both perform as fast. At this point, Himesis overtakes VF2 by a factor of 5 for large graphs.

Figure 4.11(c) shows the performance of both algorithms when *no match* exists. For small graphs, VF2 is about 24 times faster than Himesis. The medium graph category must be divided into two. For graphs with 10^3 to 10^4 nodes, VF2 is 3.6 times faster than Himesis. As for graphs with 10^4 to 10^5 nodes, Himesis overtakes by a factor of 2.2. The break even point is around 1.7×10^4 nodes. At this point, Himesis overtakes VF2 by 3 times for large graphs.

The table in Figure 4.11(d) summarizes these observations. Notice how Himesis significantly outperforms VF2 for large graphs.

4.4.4 Pattern Matching

The transformation kernel of AToMPM is *T-Core*. In *T-Core*, the pre- and post-condition patterns of a rule are encoded as Himesis graphs. A pre-condition is composed of a positive condition graph (LHS) and optional negative condition graphs (NACs). Proposition (4.9) defines the semantics of a rule with n NACs: if an occurrence of the LHS is found in the source graph before the rule is applied and none of the NACs are found, then an occurrence of the RHS must be found in the source graph after the rule has been applied. A more formal definition based on category theory can be found in [EPT04].

$$LHS \wedge \neg NAC_1 \wedge \neg NAC_2 \wedge \ldots \wedge \neg NAC_n \Rightarrow RHS \tag{4.9}$$

Some approaches have extended the semantics of graph transformation rules by redefining the LHS as a combination of different patterns. This is still compatible with traditional graph transformation rules if the patterns are conjuncted. In [BV06], both conjunction and disjunction of patterns are allowed. In [RK09], existential and universal quantifiers have been added, leading to amalgamated rules. In the

previous chapter we showed how these two extensions can be emulated in *T-Core* with pattern nesting and pivots.

In Himesis, a node n of a pattern graph holds the following information:

- A universally unique identifier: such identifiers are ensured to be unique at all time.
- The type t of the model element n encodes: the absolute path (across packages) of the name of the type element.
- A boolean flag stm specifying whether a source node mapped to n must be of type t or a sub-type of t.
- The set st of all sub-types of t.
- The identifier of a binding pivot $\overleftarrow{x}$ (for pre-condition graphs). If specified, it predefines which source node that was assigned to the pivot x must be matched to n.
- The identifier of a pivot assignment $\overrightarrow{x}$. If specified, it indicates that the source node mapped to n will be assigned to the pivot x.
- A label global to the scope of the rule. Node labelling in the different pattern graphs of the rule is used as follows. In the LHS, a label allows one to distinguish between two nodes of the same type that must be mapped to different source nodes. A label present in both the LHS and the RHS or in both the LHS and a NAC corresponds to the same matched source node. A label present in a NAC but not in the LHS allows one to distinguish between two nodes of the same type that must be mapped to different source nodes.
- Each attribute of the meta-model element corresponding to t is subject to the RAM procedure [KMS+10]. In the LHS and the NAC, the node is assigned one constraint per attribute. The constraint can be of arbitrary complexity, but can only refer to source nodes bound to the corresponding pattern (LHS xor NAC). In the RHS, the node is assigned an action code per attribute. The action can be of arbitrary complexity, but can only refer to source nodes bound to the LHS pattern.

The size of the data stored in each pattern node is 1,342 bytes, without taking into consideration the meta-model attributes. Additional information is stored at the graph pattern level: the set of all meta-models involved in the pattern[8] as well as an additional constraint (for a LHS or a NAC) or action (for an RHS). The constraints and actions are treated similarly to pattern node attributes.

Up to now, we have described an efficient solution for finding a sub-graph of the source graph isomorphic to the pattern graph. However, this is not sufficient for pattern matching as it only takes into account the topology of the pattern graph. Constraints attached to match patterns as well as NACs must be taken into consideration as well. Algorithm 16 specifies a procedure that modifies the previous sub-graph isomorphism solution for pattern matching purposes. We must first modify the `extend` procedure to handle constraints on meta-model attributes and node typing. The type of a pattern node p and a source node s must correspond. This requirement must be verified as early as possible to reduce the search space. We therefore modify the function `areCompatible` in Algorithm 14. More

[8] Because in AToMPM, rules can involve many meta-models as in *e.g.*, multi graph grammars [KS06a].

specifically, condition (4.7) must now take into consideration the types of the candidate pair (p,s) as specified in (4.10), such that the type of s is the same as the type of p or one of its sub-types. (4.7) can then be rewritten as:

$$|Out(p)| \leq |Out(s)| \wedge |In(p)| \leq |In(s)| \wedge ((s.t = p.t) \vee (p.stm \wedge s.t \in p.st)) \tag{4.10}$$

Additionally, the function `areSemanticallyFeasible` must ensure that the attributes held in s each satisfy the corresponding meta-model attribute constraints in p. Also, to help the algorithm find a match as soon as possible, we have parametrized the `suggestMapping` function with a priority mechanism to suggest a candidate pair. Our implementation allows us to specify an arbitrary order of a terminal set. By default, `suggestMapping` will suggest an unmatched pattern node such that its type occurs the least often in the graph. This heuristic ordering can be modularly extended with further knowledge of the pattern graph and the source graph.

The pattern matching algorithm of Himesis is described in Algorithm 16. The procedure `match` takes a source graph G and the LHS pattern graph as input. Pivot bindings may also be specified in the *context*. The procedure can be one of three cases. In the following, we consider a match as *valid* if the source nodes in the mappings of the match satisfy the constraint of the pattern graph.

No NACs. When no NACs are specified in the pre-condition pattern, the `computeMappings` procedure is called on the LHS and returns the valid matches.

Unbound NACs. We denote a NAC as unbound if none of its nodes have a label present in the corresponding LHS. If the pre-condition has unbound NACs, it suffices to find one valid NAC match to prevent the pre-condition pattern from successfully finding any matches. Lines 3-14 describe this behaviour. First, G is matched on the NAC with the provided context. If no valid match is found, the procedure then tries to find matches for the LHS as in the previous case. Otherwise, no match is output.

Bound NACs. All other NACs are bound to the LHS. Since `computeMappings` is the most costly procedure, we want to avoid computing mappings twice, *i.e.*, the common part between the LHS and a NAC. Our approach is to first match the common part between the LHS and a NAC, then continue the matching along the NAC, and finally, if no valid NAC matches were found, continue from the match of the common part along the LHS.

A NAC having a common part with the LHS means that there is a sub-graph of the LHS that overlaps with the NAC. We denote this intersection as a pre-condition graph called *bridge*. In general, computing the bridge would require us to find the maximum common sub-graph (MCS) between these two graphs. Solving the MCS isomorphism problem is NP-complete. However, making use of the labels in the Himesis pattern graphs reduces the complexity to linear-time. The bridge can therefore be constructed as follows: if a node has a label present in nodes of both the LHS and the NAC, then this node is part of the bridge. Also, every edge in the smallest graph between the LHS and the NAC whose source and target nodes are in the bridge is part of the bridge. However, recall that pattern nodes may also hold a constraint for each attribute from the meta-model of the domain of the transformation. Thus, each meta-model attribute of

Algorithm 16 `match`(G, LHS, context)

```
1:  validMatches ← ∅
2:  moreNACs ← False
3:  for NAC in LHS.getNACs() do
4:      bridge ← NAC.getBridge()
5:      if V(NAC.getBridge()) > 0 then
6:          moreNACs ← True
7:      else
8:          for nacMatch in computeMappings(G, NAC, context) do
9:              if NAC.checkConstraint(nacMatch) then
10:                 return ∅
11:             end if
12:         end for
13:     end if
14: end for
15: if not moreNACs then
16:     for lhsMatch in computeMappings(G, LHS, context) do
17:         if LHS.checkConstraint(lhsMatch) then
18:             validMatches ← validMatches ∪ {lhsMatch}
19:         end if
20:     end for
21:     return validMatches
22: end if
23: maxNAC ← LHS.getNACwithMaxBridge()
24: B ← maxNAC.getBridge()
25: for bMatch in computeMappings(G, B, context) do
26:     for maxNACMatching in computeMappings(G, maxNAC, bMatch ∪ context) do
27:         if not maxNAC.checkConstraint(maxNACMatching) then
28:             goto 20
29:         end if
30:     end for
31:     for lhsMatch in computeMappings(G, LHS, bMatch ∪ context) do
32:         if LHS.checkConstraint(lhsMatch) then
33:             for NAC in LHS.getNACs() do
34:                 if NAC ≠ maxNAC and V(NAC.getBridge()) > 0 then
35:                     for nacMatch in computeMappings(G, NAC, lhsMatch ∪ context) do
36:                         if not NAC.checkConstraint(nacMatch) then
37:                             validMatches ← validMatches ∪ {lhsMatch}
38:                         end if
39:                     end for
40:                 end if
41:             end for
42:         end if
43:     end for
44: end for
45: return validMatches
```

a bridge node is computed as the conjunction of the corresponding attribute constraint in the LHS and the corresponding attribute constraint in the NAC. Note that no constraint is added on the pattern graph of the bridge as in the LHS or NAC cases. It is easy to show that the time complexity of constructing the bridge between the LHS and an NAC is $O(V+E)$, where $V = max(|V(LHS)|, |V(NAC)|)$ and $E = min(|E(LHS)|, |E(NAC)|)$[9].

In the `match` procedure, line 24 computes the bridge B with the largest number of nodes. Since a bridge can be statically computed, all bridges have already been precomputed and integrated in the corresponding NACs at compile-time. On line 25, G is matched on B with the provided context. Then, on lines 26-30, G is matched on the NAC corresponding to B. To prune the search space of this matching, the bridge mappings are provided as context together with the initial context. Those mappings are valid since the nodes in B are in the NAC as well. If a valid match for this NAC is found, then the current match of B is discarded and the next one is tried. When a match of B is found such that it does not induce a valid match, we match G on the LHS with again the bridge mappings provided as context together with the initial context. Each valid match of the LHS represents a potential valid match of the procedure. However, there may be additional bound NACs with a bridge having fewer nodes than B. In this case, lines 33-41 ensure that only the valid matches of the LHS that do not satisfy the remaining NACs are stored. Note that when applying the `computeMappings` procedure on G with the remaining NACs, the LHS mappings are provided as context together with any pivot node bound in the LHS that were given in the initial context. Finally on line 45, only the valid matches are output.

4.4.5 Rewriting the Matches

A rule is successfully applied when proposition (4.9) is satisfied. The pre-condition satisfaction is ensured by the pattern matching algorithm described previously. One way to satisfy the post-condition is to modify the matched nodes in the source graph appropriately. To transform (or rewrite) the matches, a Himesis RHS pattern graph is provided with a compiled `execute` function encoding the appropriate modification actions. Given the LHS and the RHS pattern graphs, the rewriting of a match $M = \{(p,s)|p \in \text{LHS} \wedge s \in G\}$ can be statically determined. For each $(p,s) \in M$ we perform the following steps in order:

1. If the label of p is present in both the LHS and the RHS, then an *update operation* is executed. Each attribute of s is set according to the action specified in the corresponding meta-model attribute of the RHS node that has the same label as p.
2. Let C represent the graph whose node labels are present in the RHS but not in the interface graph K. Also edges of C are constructed in a similar way as for the bridge, *i.e.*, $E(C) = \{(n_i,n_j)|n_i,n_j \in V(C) \wedge (n_i,n_j) \in E(\text{RHS})\}$. Then a *create operation* is applied to the nodes and edges of C. For each node (or edge) in $V(C)$ (or $E(C)$), a corresponding source node (edge) is created in the source graph. Furthermore, the attributes of the new nodes are initialised according to the action specified in the corresponding meta-model attribute of the respective node in C.

[9] V should also be multiplied by the maximum number of meta-model attributes, which is small in practice.

3. If the label of p is present in the LHS but not in the RHS, then a *delete operation* is applied and removes s from the source graph. Note that in IGraph, deleting a node automatically deletes its adjacent edges.
4. If p is assigned a pivot identifier $\vec{x}$, then $\vec{x}$ will be mapped to s.
5. Finally, after all nodes have been processed, we apply the action specified in the RHS on the source nodes that are in M as well as those created from C.

Since the rewriting phase is compiled, its run-time complexity is linear: $O(|V(\text{LHS})| + |E(\text{LHS})| + |V(\text{RHS})| + |E(\text{RHS})|)$ Note that according to the graph transformation literature [EEKR97], Himesis' transformation procedure follows the Single-Pushout (SPO) approach as opposed to the Double-Pushout (DPO) approach. The identification issue of the glueing condition in DPO is avoided thanks to our labelling mechanism in place. That is because every node in each pattern graph is unique and thus may be mapped to exactly one node in each matching. We have explicitly chosen to solve the dangling edges issue automatically. That is, if a matched source node must be deleted, all its adjacent edges will be deleted too. This has the advantage of reducing the number of rules in a transformation.

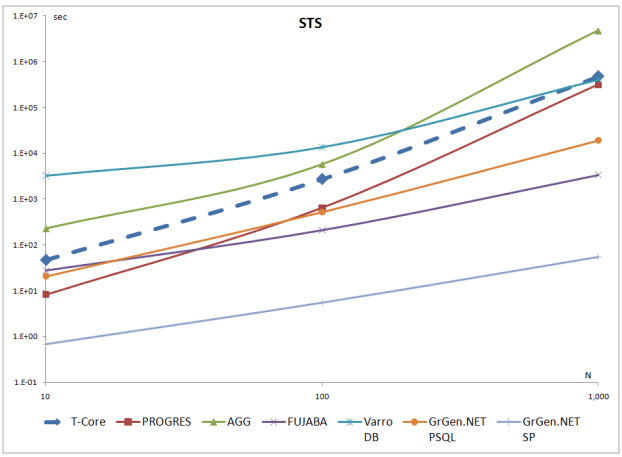
4.5 Related Work

In his Masters thesis, Provost [Pro05] described an efficient framework for graph-sub-graph isomorphism. The implementation of Algorithm 14 is based on his work. However, his approach does not address pattern matching as used in model transformations. Also, there is no evaluation of the performance of each CRUD operation as done in this chapter.

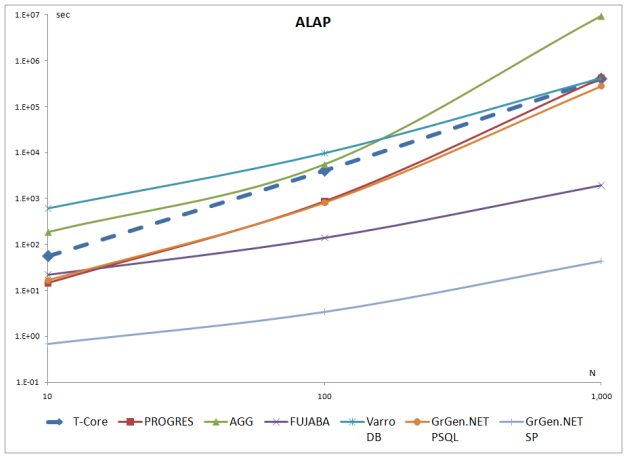
To compare Himesis with other graph transformation approaches, we provide our results for a standard graph transformation benchmark: the *Distributed Mutual Exclusion Algorithm* benchmark presented by Varró in [VSV05]. Although some measurements were reported in the original paper, Geiss *et al.* [GBG+06] provide a more complete spectrum of measurements with more tools. In the latter paper, the measurements were carried out on an AMD Athlon 3000+ with 1GB of RAM. To reuse these results, we multiplied[10] Geiss' figures by 0.684 to compensate for the speed of our processor.

The tools used for this comparison are the following. The transformation tools GrGen.NET SP [GBG+06], FUJABA [FNTZ00], and PROGRES [ZS92] use search plan techniques for the matching phase. An approach from Varró [VFV06] (hereafter referred to as VarroDB) to execute graph transformations directly in a relational database is also considered. We also include GrGen.NET PSQL which, in contrast with GrGen.NET SP, also stores the graphs in a relational database. Finally, AGG [Rud98] is the only tool that uses a CSP for the matching phase. All experiments were performed without any of the optimizations suggested by the benchmark, as no measurements for these cases were available for the other tools. As Himesis provides a framework for manipulating graphs, we integrated it in *T-Core*, in combination with Python and called it *Py-T-Core*. More specifically, the *T-Core matcher* calls the procedure `match` form Algorithm 16 and the *T-Core rewriter* calls the `execute` method of the corresponding RHS graph to perform the rewriting.

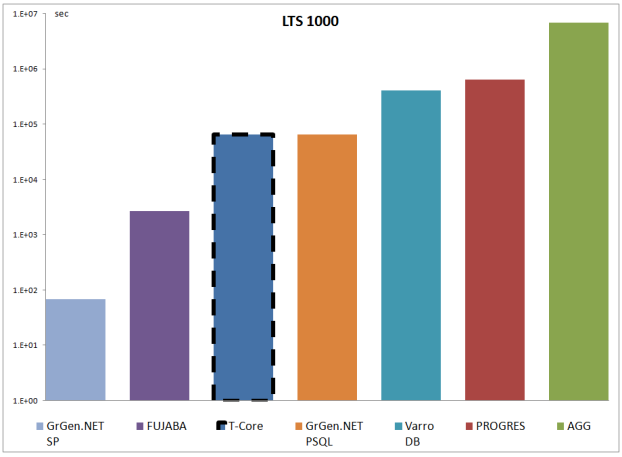
[10] This factor is obtained from the SPEC organization at http://www.spec.org/cpu2000/results/cpu2000.html.



(a)



(b)



(c)

Case	N	T-Core	PROGRES	AGG	FUJABA	Varro DB	GrGen.NET PSQL	GrGen.NET SP
STS	10	48	8	226	27	3,213	21	1
STS	100	2,745	647	5,677	209	13,560	520	5
STS	1,000	474,325	313,956	4,706,604	3,370	405,954	18,957	54
ALAP	10	56	14	185	22	611	16	1
ALAP	100	4,087	867	5,490	139	9,636	807	3
ALAP	1,000	407,493	417,650	9,339,336	1,930	408,211	277,704	44
LTS	1,000	65,929	644,396	6,840,000	2,651	405,749	65,996	68

(d)

Figure 4.12: Performance comparison for the Distributed Mutual Exclusion Algorithm benchmark with no optimization.

For the Short Transformation Sequence experiment (STS), Figure 4.12(a) shows that *T-Core*'s performance is average compared to the other tools. It however performs 5.6 times better than *AGG*, which is the only other tool whose matching phase is also implemented as a CSP. For the As Long As Possible experiment (ALAP), Figure 4.12(b) shows that, once more, *T-Core*'s performance is average compared to the other tools. It however performs 9.2 times better than *AGG*. For the Long Transformation Sequence experiment (LTS), the only results available are for N=1,000 (N processes with one resource). Figure 4.12(c) shows that *T-Core* performs quite well compared to the other tools. It now performs on average over 100 times better than *AGG* and about as fast as GrGen.NET using ProgresSQL. The table in Figure 4.12(d) summarizes the results.

4.6 Conclusion

This chapter describes an efficient implementation of *T-Core*, a library of transformation building blocks. Himesis, a low-level framework for graph manipulation based on the IGraph library, allows one to efficiently manipulate models encoded as graphs. The primitive CRUD operations are very fast even for models with up to 10^6 elements. Moreover, an efficient pattern matching algorithm was implemented to perform model transformation on models encoded as Himesis graphs. The comparison of performance with other existing tools and approaches shows that Himesis is indeed an efficient framework. The implementation of *Py-T-Core*, described in Section 3.5 of Chapter 3, relies entirely on Himesis.

The *T-Core* API can be called from a modelling language or a programming language. This "glue language" provides the scheduling of transformation units encapsulated in *T-Core*. *Py-T-Core* is the result of implementing *T-Core* in Python, thus making model transformation available to programmers. Himesis is therefore an optimal choice for implementing the underlying data-structures and matching for *Py-T-Core*.

Regarding speed, one reason for the average performance results for graph transformation tasks may be that Himesis is entirely implemented in Python. Future plans are to implement the core algorithms in a faster target language, such as C. Regarding scalability beyond graphs of size 10^6 elements, one direction would be to store the host graph on a secondary storage, such as a database, to overcome size limitations because of limited physical memory. A hybrid solution is envisaged where the host graph is stored in physical memory for as long as no thrashing is observed as anticipated by Figure 4.9(a). After that threshold, the system would switches to database storage *e.g.*, by adapting the IGraph library as described in [ABFL$^+$09].

5

Explicit Modelling of Transformations

The previous two chapters focused on the building blocks of the engine behind a transformation language, *i.e.*, how it operates. In this chapter, transformations are *modelled* explicitly following MPM principles. The focus is on the other components of the transformation language, namely the transformation units and the patterns language. Despite the pivotal significance of transformations for model-driven approaches, there have not been any attempts to explicitly model transformation languages yet although a number of benefits are to be gained. First, transformation developers may change the design of their transformation languages by modelling, rather than programming. Second, they may use environments to create transformations that are customized with respect to the input and output languages involved. Therefore, this chapter identifies, discusses, and demonstrates some of the above advantages. In particular, it suggests ways to systematically support developers in creating transformation languages by means of semi-automated meta-modelling.

5.1 Introduction

Model-driven approaches are gaining popularity both in the form of being based on standard modelling languages, such as the UML [Obj09], as well as domain-specific modelling languages (DSL) [GTK+07]. In both instances, the aim is to increase developer productivity by (2) raising the level of abstraction at which systems can be specified (for UML) and (2) by lowering the impedance mismatch between a modelling language and its application domain [AK07] (for DSLs). There are still many open problems with respect to the economic development of DSLs, but their definition is well understood.

This shifts the focus on transformations which have a number of applications (c.f., Chapter 1.2). A number of transformation paradigms exists, *e.g.*, template-based, rule-based, relational, with or without explicit control flow and are supported by various implementations (c.f., Chapter 2). They provide tremendous value for developers but, in each implementation, the transformation paradigm is hard-coded to be used as is. The implementations do not provide a way to interrogate or modify transformation definitions as first-class transformation models [BBG+06]. This is surprising as there are a number of benefits to be gained when treating transformations as first-class citizens [BFJ+03, TJF+09] which are explicitly modelled and amenable to introspection and modification. We identify the following potential advantages:

- It becomes easier to explore the language design space by making alterations to the control flow, mapping, and pattern specification parts of the language. Obviously, this requires modelling the respective semantics, but once available, alterations to the syntax and semantics definitions of such transformation (meta-)models should be easier to perform than the respective changes in a code base.
- Instead of using a *generic pattern specification language* to be used for all input and output languages, one can utilize *customized pattern specification languages* on a case-by-case basis. Automating the creation of such customized pattern specification languages opens up a cost-neutral way to achieve customized transformation definition environments providing increased rigour.
- Transformation definitions can be the subjects of other transformations, thus facilitating the concept of higher-order transformations [TJF+09]. Higher-order transformations are of particular interest since they enable a separation of transformation concerns which are either harder, or even impossible, to realize with standard multi-stage transformations. This is achieved by splitting a complex transformation into simpler ones and then integrating them with a higher-order transformation.

The following section introduces our typical transformation example which will be used as the basis of subsequent discussions. In Section 5.3, we investigate the automated construction of customized pattern specification languages, using the components relaxation, augmentation, and modification, exploring and discussing alternative solutions. This provides a systematic *procedure* for explicitly modelling transformation languages. Finally, further related work is discussed in Section 5.5.

5.2 A Typical Transformation

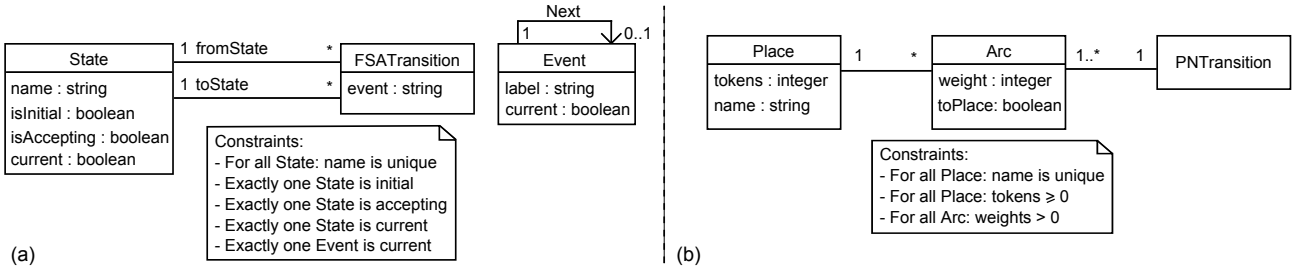


Figure 5.1: (a) FSA & (b) Petri net meta-models.

The example that we will use in the remainder of the chapter to illustrate our arguments is a typical case of a DSL being assigned a semantics by translating it into a target formalism with known semantics. In order to define the semantics of Statecharts and/or perform reachability analyses on them, one can translate them to Petri nets [dLV02]. Another reason for considering this particular translation is that one can use Petri nets as a common semantic domain for Statecharts, sequence diagrams, and activity diagrams. For the purposes of this paper, however, we restrict ourselves to translating finite state automata (FSA), rather than Statecharts, into Petri nets. The resulting transformation definitions

of this translation are much simpler but still rich enough to illustrate our arguments. Figure 5.1 shows both meta-models. The *event* attribute of the Transition class should have been represented as an association to the class Event instead. However, this choice was made to intuitively represent the concrete syntax of a transition (displaying the event label on top of the arrow) in *AToM*[3].

5.2.1 Finite State Automata as Language Recognizers

More specifically, we interpret our state automata to be language recognizers, *i.e.*, they either accept input sequences as belonging to respective regular languages or not. For the sake of example, recall Figure 1.5 from Chapter 1. On the left hand side, the figure shows a sample input sequence ("yees") and a finite state automaton accepting the language y(e)*s. In our example, we want to simulate the execution of the finite state automaton in the context of receiving the events from the input sequence in order to ascertain whether the input sequence is a sentence of the language. Note that in the meta-model of the FSA, we assume only one accepting state as mentioned in the constraints. To this end, we translate such scenarios into corresponding Petri nets (see the right hand side of the figure) so that the behaviour of the Petri net model is equivalent to the intended behaviour of the FSA model.

5.2.2 Translating Finite State Automata To Petri Nets

Figure 5.2 shows the transformation rules that are required to translate a finite state automaton plus an input sequence into a Petri net that can be used to simulate the automaton execution. The rules behave as graph transformation rules with single push-out (SPO) semantics[1]. Their concrete visual representation is composed of three components: a LHS pattern to the left of the arrowhead, a RHS pattern to its right, and possibly multiple NAC patterns bounded by dashed lines. Also, these rules employ numerical labels to indicate identity of pattern elements as explained in Section 1.5.

The scheduling of the rules is controlled by a modelling and simulation language adapted to graph transformation [SV11]. Part III of the thesis is dedicated to the precise specification of this transformation language. The general idea of the transformation is first to map the automata of the FSA model, then the event list, then making the link between the two in the Petri net model, and finally to ensure the language recognizer behaviour and remove all temporary artefacts. The rules in Figure 5.2 are applied following the order they appear:

1. The rule State2Place is the first to be applied, mapping an FSA state to a Petri net place. Only the place corresponding to the initial state has a token. Moreover, it is applied iteratively for as long as there are unmapped FSA states remaining. Note the presence of generic nodes and links (the small filled rectangles linked to dashed lines). They act as traceability links to retain which Petri net element is mapped to which FSA element.
2. The rule NextPlace is applied only once. This creates a single place "NEXT" that will make the bridge between Petri net model part modelling the automata and the part modelling the event list.

[1] Actually, it is a variant of SPO where the dangling edge problem remains but the identification problem is resolved: if two elements of the same type occur in the same pattern, they will be mapped to two distinct model elements.

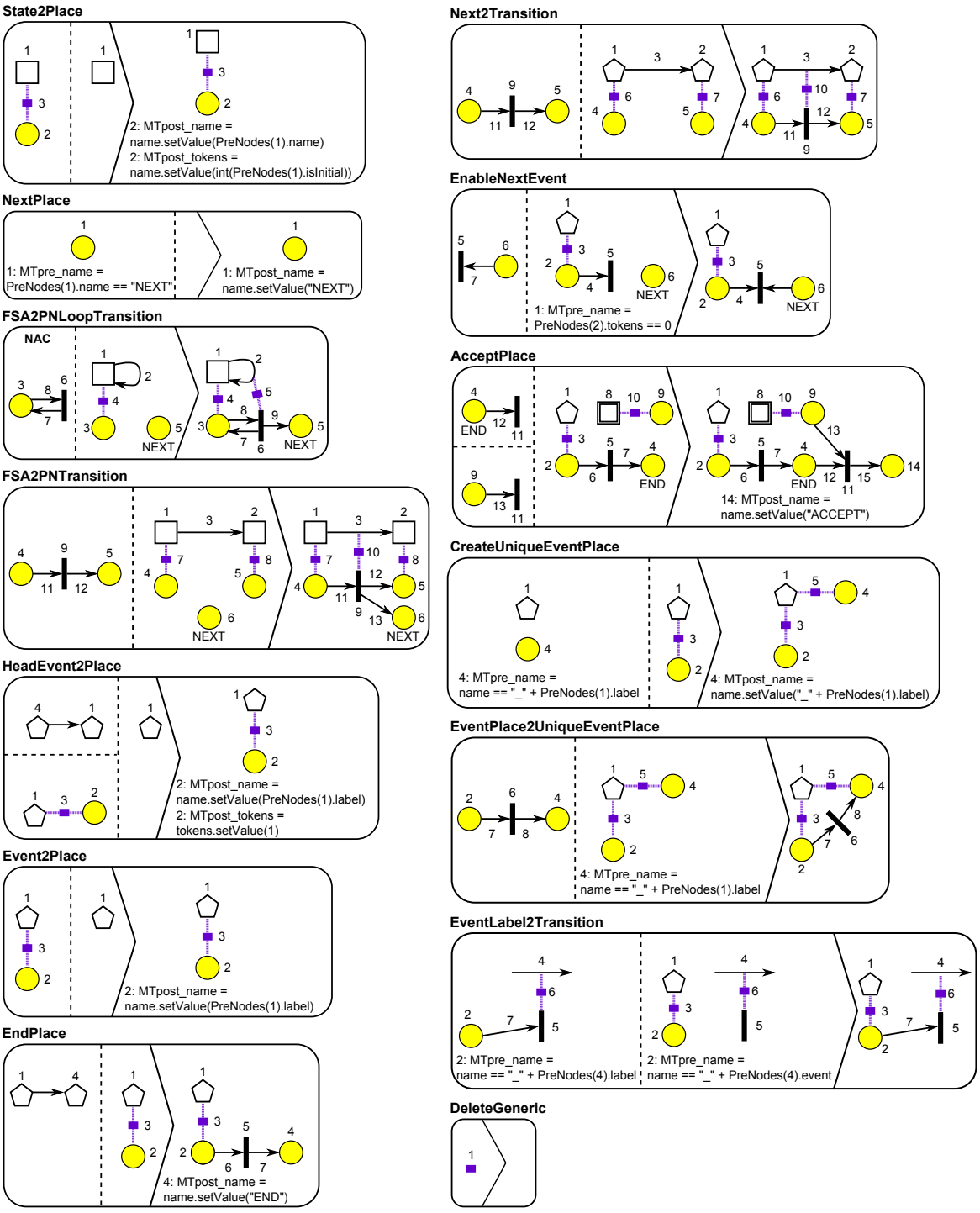


Figure 5.2: The transformation rules for the translational semantics transformation from FSA to Petri nets.

3. The rules `FSA2PNLoopTransition` and `FSA2PNTransition` map FSA transitions to Petri net transitions. The two rules are applied iteratively as before, but the choice of which rule is applied is non-deterministic, since they are sequential independent.
4. The rules `HeadEvent2Place`, `Event2Place`, and `EndPlace` create Petri net places corresponding to the first FSA event, those in the middle of the list, and the last event (called "END") respectively. Only the place corresponding to the head event in the list has a token. The rule `Event2Place` is again applied iteratively, but the other two are only applied once.
5. Then the rule `Next2Transition` maps the event list links to Petri net transitions. The rule is applied iteratively.
6. The rule `EnableNextEvent`, applied iteratively, creates an arc between the NEXT place and every transition in the Petri net modelling the event list, except the transition that has an incoming arc from the place in the head. This forbids the automata part of the Petri net to fire transitions without the event list part firing first.
7. The rule `AcceptPlace` is applied once. It creates a transition from both the place corresponding to the accepting state and the END place to a place "ACCEPT". The semantics of a token present in the ACCEPT place means that the string encoded in the event list is recognized by the automaton.
8. The rules `CreateUniqueEventPlace`, `EventPlace2UniqueEventPlace`, and `EventLabel2Transition` create intermediary places between the automata part and the event list part in the Petri net. This ensures that the label of the event in the event list is correctly mapped to the transition with the same event in the Petri net model. The three rules are applied in this order, each iteratively.
9. Finally, the rule `DeleteGeneric` removes all generic nodes. Thanks to the SPO semantics of the rule, all the links connected to generic nodes (*i.e.*, generic links) are removed implicitly. The rule is applied on *all* its matches at once.

Although a proof of correctness of this transformation would be ideal, we only focus on a potential issue of semantics between FSA and Petri nets. We are well aware of the tension between the "must transition" and "may fire" semantics of finite state automata and Petri nets, respectively. In timed Petri nets, this difference may lead to a situation where a finite state automaton does not change states anymore even though it *should*, just because the Petri net used for simulating it does not fire transitions anymore, even though it *could*. However, the place/transition nets we assume do not create this mismatch and a simulator for them will fire enabled transitions. We leave the support for these kinds of property preserving proofs for future work.

5.3 Explicit Transformation Modelling

In this section, we describe and discuss the explicit modelling of transformation definitions as an enabler of customized transformation *development environments*.

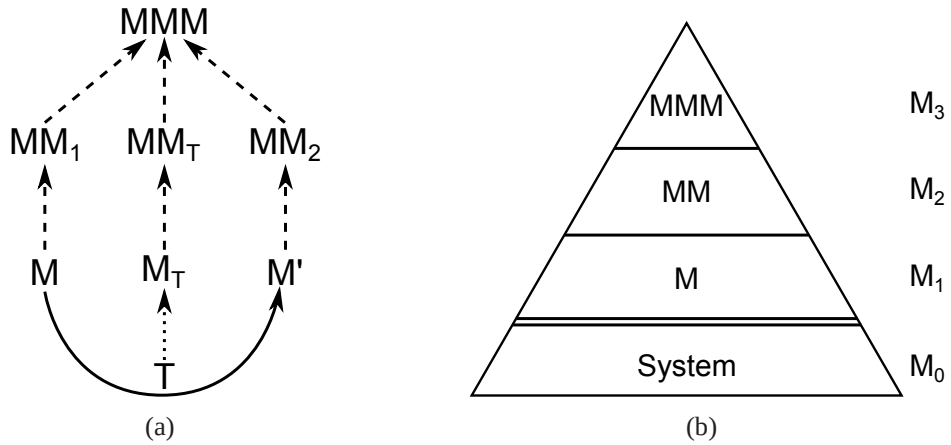


Figure 5.3: (a) Meta-modelling of model transformations and (b) the MDA meta-layers.

Meta-modelling[2], *i.e.,* the explicit specification of a language's well-formedness constraints, has become popular because of a number of associated advantages:

- the specification is not hidden in the code of a tool, making it easier to understand and correct;
- the specification can be altered by users of the tool instead of requiring a new tool release;
- one can reason about the specification and the models it describes.

The same advantages apply if metamorphosing is not only applied to modelling language definitions, but also to transformation definitions. While there is a considerable initial investment to be made in explicitly modelling a transformation language including its semantics, the prospect to more easily experiment with language features, customize them for certain purposes, and allow transformations to be reasoned about (since modelled) and/or modified makes that investment worthwhile. Clearly, in order to enable the last aspect mentioned above, the transformation language's mapping approach, *e.g.,* rule-based graph transformation, needs to be explicitly modelled.

5.3.1 Models, Meta-Models, and Transformations

The diagram in Figure 5.3(a) depicts the relations between a transformation and the artefacts it is involved with. T is the *operation* that transforms a model M into a model M'. They conform to their respective meta-model MM_1 and MM_2. M_T models this transformation and, conversely, T executes M_T. In fact, M_T is a *model* of a transformation that transforms any model of MM_1 into a model of MM_2. MM_T is a *meta-model* of all transformations that transform any meta-model. Since everything is modelled explicitly, *MMM* is the *meta-meta-model i.e.,* it is the meta-model of the language used to describe meta-models. Typically, *MMM* conforms to itself in a sound bootstrapped environment. This explicit point of view on models of transformations is compatible with the model-driven architecture (MDA) meta-layers [KWB03] depicted in Figure 5.3(b). It places a transformation at the level of real systems (the M_0 layer), a model transformation (or transformation model) at the instance level

[2]Linguistic meta-modelling [AK07], to be precise.

(the M_1 layer) and the transformation language at the level of UML class diagrams used to define a meta-model (the M_2 layer). In MDA, the M_3 layer would represent the meta-model of UML *i.e.,* the meta-object facility (MOF).

The goal here is to provide transformation development environments customized to the specific domain of application. Therefore the focus should be on MM_T, the meta-model of all transformations, to provide a general solution. Unlike the mapping and control aspects of a transformation language, its pattern specification sub-language depends on other languages: the domain of the transformation. The input and output languages of a transformation determine which pattern specifications for the pre-condition and the post-condition can be considered well-formed. The underlying assumption here is that the pattern specification language should not be generic to fit all possible input and output languages, but specifically tailored to the input and output languages involved.

5.3.2 Generic versus Customized Pattern Specification Languages

Some studies have shown that using appropriate visual models is advantageous for a better understanding and a faster modification of software [Whi97, NPC01]. As this is one of the main aims of DSLs, we will assume visual concrete syntax of rule pattern specifications rather than textual ones.

The most economic approach to providing a pattern specification language is to offer a generic one. Most tools do not use concrete syntax for specifying transformation patterns and thus are able to use the same generic (often UML object-diagram-inspired) pattern specification syntax for all possible input/output languages. They often also have an underlying generic (often MOF-like) representation format which can be used to represent elements from any input/output language.

There are good reasons, however, to consider using a pattern specification language which is customized to the input/output languages involved:

- One may use pattern specification visualizations which are adapted to the languages involved. Even if no concrete syntax is used, one may still want to customize the syntax, *e.g.,* to adequately visualize connector elements.
- A customized syntax allows excluding patterns from being specified that do not have a chance of matching sub-graphs in the host graphs. For instance, in the context of Petri nets, a pattern consisting of an arc linking two places will never be matched on any valid Petri net instance (*i.e.,* conforming to the meta-model in Figure 5.1).

A generic pattern specification language will allow any pattern to be expressed whether or not it will be able to match sub-graphs from the input language(s) or generate sub-graphs conforming to the meta-model(s) of the output language(s). Just as a plain domain-specific modelling tool has advantages for its users, guiding them to produce meaningful models, a customized transformation pattern specification tool also aids in avoiding meaningless pattern specifications. The main disadvantage however is that a customized pattern language requires more work for the transformation language engineer. Whether this customization is achieved by changing the representation format for each generated transformation definition environment or by just exchanging a language definition against which generic pattern specifications are checked is immaterial to the user, but a tool builder decision. In the

following, we assume that, in one way or another, pattern specifications can be checked for conformance to a pattern specification language definition. As a result, a method needs to be identified that enables these conformance checks in an economic manner, while offering the transformation language user maximum benefits.

5.3.3 Meta-models versus Conformance Checks

Unfortunately, providing a customized pattern specification language is not as easy as simply reusing the corresponding input/output meta-models. First, demanding a full adherence of pattern specifications to original language definitions is not practical. If all minimal multiplicity requirements of language definitions were enforced, one could not specify useful patterns such as `EventLabel2Transition` of Figure 5.2, which refer to model fragments, ignoring minimal multiplicity requirements. Second, one may want to provide several levels of rigour with respect to checking the well-formedness of pattern specifications. While the transformation designer edits a pattern specification, one most certainly does not want to enforce all well-formedness constraints. It also should be possible to save ill-formed sketches to be worked on later. This does not mean, however, that the complete absence of all potential well-formedness checks is always the best choice in such cases. Table 5.1 lists potentially useful levels of conformance checking rigorousness. There are two ways to enable the use of such levels of conformance:

1. either one creates modified language definitions and performs a normal conformance check against them, or
2. one uses original language definitions, but with accordingly modified conformance checks.

The second option has a number of advantages:

- One can simply use the original language definitions; there is no need to create multiple variants of them.
- Switching between conformance levels does not require the switch of a meta-model; the latter is quite feasible though with an appropriate architecture.
- The alternative 1. (above) cannot use a standard conformance check anyhow (see Sections 5.3.4 and 5.5).

However, there are also a number of disadvantages:

Level of rigour	Description
Free form	no constraints at all
Valid elements	elements are typed by the meta-model
Valid multiplicities	(relaxed) multiplicity constraints are enforced
Valid constraints	(a subset of) meta-model constraints are enforced

Table 5.1: Levels of Conformance.

- Some generic way to extend languages defined by meta-models is required; pattern specification languages require additional features beyond the original input/output languages (see Section 5.3.4). Customized meta-models can easily incorporate these.
- Custom conformance checks are harder to reason about than custom meta-models; in the absence of a fully modelled action language, conformance checks will be implemented in some programming language making it harder to see and analyse what relation they actually implement.
- Conformance checks are harder to customize by users; transformation engineers can be expected to alter the transformations that yield tailored meta-models but may not be able to reprogram conformance checks.
- Swapping conformance checks means that the transformation development will remain the same; swapping meta-models opens up the possibility to use them for the automated generation of dedicated development environments with differing sets of control elements.

Finally, there is another motivation for supporting more than one mode of well-formedness checking which can only be enabled by using multiple meta-model versions: typically, transformation definitions comprise layers of rules in the sense that one will expect all rules from one layer to have matched, and then match no more, before the next layer of rules will be used. This layering often exists independently of whether or not it is dealt with explicitly (such as in *AGG*). For example in the FSA to Petri nets transformation, the nine steps enumerated in Section 5.2.2 correspond to such layers. In particular with in-place transformations, the input and output languages change from layer to layer. The first layer's input language is the source language while its output, the input to the next layer, will typically contain generic links which are not part of the source language (see Section 5.3.4). The last layer's output language is the target language, whereas all preceding layers will produce either augmented versions of it or mixtures between the source and target languages. The availability of a series of adapted meta-models may aid the transformation developer to understand what the layers involved are and assign rules to them accordingly.

We have not yet pursued the idea of using a series of transformation layer interface language definitions and it would be challenging to automate the generation of these intermediate language definitions. Luckily, however, the creation of customized pattern specification languages from original input/output language definitions can be automated very well.

5.3.4 Semi-Automated Meta-Modelling of Pattern Specifications

The previous section motivated the use of variants of original meta-models for defining the well-formedness of pattern specifications. In this section, we discuss how one can create such variants systematically and thus automate the process.

Figure 5.4 proposes a meta-model of a rule-based transformation unit: it refers to pre- and post-condition patterns as well as the pattern elements they contain. When adapting transformation languages to specific input and output languages, one needs to tailor these pre- and post-condition patterns so that they are fit to be used for the respective input and output languages. We obtain the re-

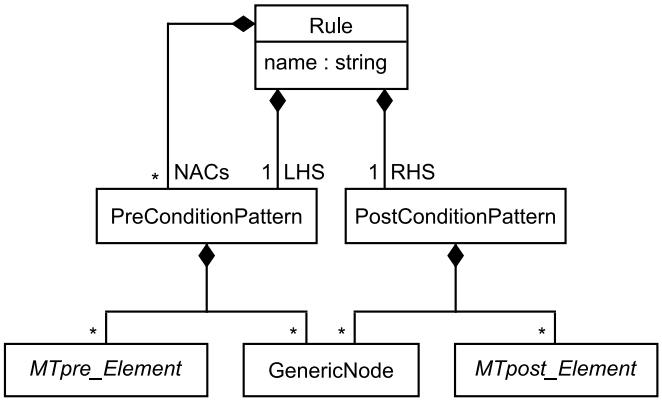


Figure 5.4: The meta-model of a rule.

quired tailored pattern specification meta-models by starting with the original language meta-models and then subjecting them to a number of changes. The required meta-model metamorphosis, called the **RAM process**, has three distinct components: relaxation, augmentation, and modification. Figure 5.5 shows the result of applying these steps to the finite state machine meta-model of Figure 5.1. This defines the pattern language of the rules designed in Figure 5.2.

Relaxation

Original language definitions cannot be used as is for defining the well-formedness of pattern specifications. First, often transformation designers aim to match for any one-of-many element types, *e.g.*, all sub-classes of a super-class. Such generalizations are typically present in original language definitions but as abstract concepts which cannot be instantiated. One relaxation step therefore is to turn such abstract concepts into concrete ones.

Second, as mentioned before, enforcing minimal multiplicity constraints would be completely impractical. A further relaxation step is, therefore, to reduce all minimal multiplicities to zero. This allows representing fragments of meta-model instances rather than pattern models that completely conform to the original meta-model. For example, the `next` association now has a 0..1 multiplicity as opposed to 1 originally on its source end. This allows one to represent such an association link isolated from its source and target elements as shown in `EventLabel2Transition`.

Third, only a subset of explicitly formulated original constraints (*e.g.*, using OCL) can be active for the purpose of checking pattern specification well-formedness. All constraints concerned with ensuring completeness of models are potentially unsuitable for the inherent fragment-like nature of specification patterns. The relaxation process could automatically filter out constraints with the help of a corresponding naming scheme for constraints or manually provided augmentations. But we currently believe any further automation will be difficult to achieve. This is why we refer to the meta-model generation as *semi*-automated.

A potential further relaxation is to raise all maximum multiplicities to “unbounded” in order to allow intermediate results that can be helpful to drive the transformation process, despite the fact

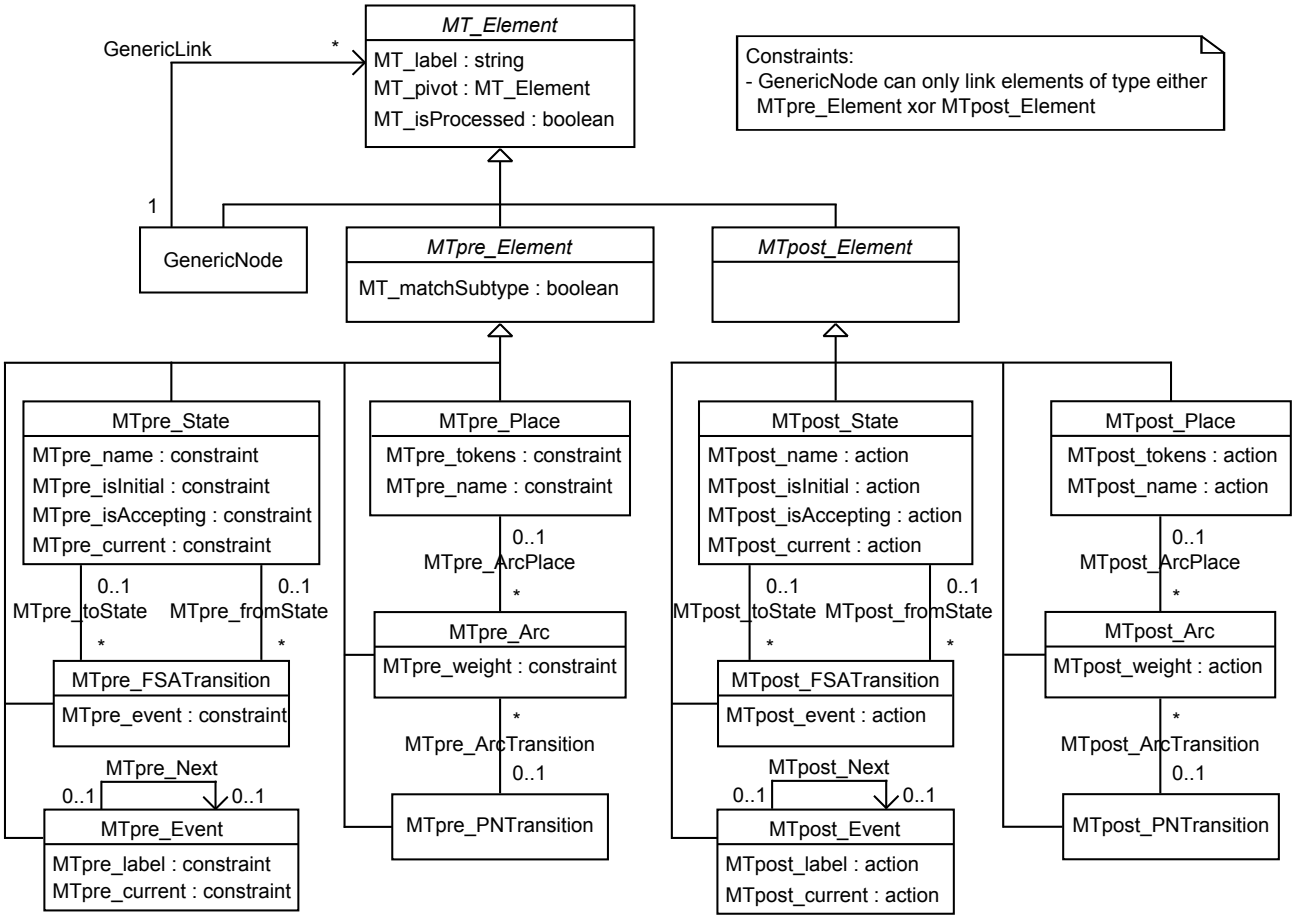


Figure 5.5: Generated pattern specification meta-model from the RAM process.

that they would be ill-formed as end results. However, we argue that purposefully violating well-formedness requirements in this way amounts to "hacking" and should be avoided. We recommend using so-called generic links for these purposes instead.

Augmentation

To be fit as pattern specification meta-models, input/output meta-models also need to be augmented with features required for transformation purposes. In Figure 5.5, all types are made descendants of `MT_Element` so that they inherit features that all elements that may appear in a pattern specification must have, *e.g.*, a way to label them for identity matching.

The generated meta-models also feature additional generic nodes and links which are often necessary to drive the transformation (*e.g.*, see the generic connectors between states and places in Figure 5.2). They allow connecting elements that conform to the source and target meta-models whenever needed, without violating the meta-model of the patterns. Trace elements are considered here as special kinds of generic elements. Note that although Figure 5.4 shows a single class diagram

representing the meta-models of the pre- and post condition patterns of each the input and output meta-models, they are in fact four separate meta-models: one for the pre-condition patterns of FSA, one for the post-condition patterns of FSA, one for the pre-condition patterns of Petri net, and one for the post-condition patterns of Petri net. Nevertheless, the meta-models of the pre-condition pattern of FSA and Petri net are connected (carteisan product) through the generic nodes and links. The same applies for the meta-models of post-condition pattern of each formalism.

Elements which are used in pre-condition patterns (subtypes of `MTpre_element`) also need a flag feature that tells the pattern matcher whether to look for exact types or allow sub-type matching as well. This allows one to have *abstract rules* [dLBE+07] which is very handy for the transformation designer who, instead of specifying the same rule for each sub-type, only needs to specify one rule with the corresponding element flagged to also match sub-types.

Parameter or pivot passing is also a very useful feature. Pre-condition pattern elements may be pre-bound to model elements before the rule matches: input parameters/pivots. A post-condition pattern element (subtypes of `MTpost_element`) may be assigned to a specific model element that can be used as input pivot in another rule: output parameters/pivots. Note that, in the case of query (as opposed to a rule), pre-condition pattern elements also have the possibility to specify an output pivot.

Depending on the expressiveness of the pattern language, the meta-model can be augmented with further features. A former version of *VMTS* offered the possibility to specify multiplicities on associations in pattern [LLC05]. For example, if an association link A—B is annotated with 1 on the A end and 1..2 on the B end, this would mean that exactly one A must be found and it must be connected to one or two B's.

In the relaxation step, some all or none of the original meta-model constraints may have been removed. However, additional constraints on the pattern elements may be required on the augmented structure, *i.e.,* on the properties that were added on top of those existing in the original meta-models, not in the shared dimension. For example, the notion of label was added. Then an additional constraint on the pattern meta-model is then required to ensure the uniqueness of a label within a pattern.

In the original meta-models, it is possible that no concrete syntax was specified for abstract elements as they will never be instantiated. Nevertheless, because of the relaxation step, such elements can be instantiated in the patterns and therefore require a concrete representation. The remaining differences between the original and generated meta-model elements are all modifications of existing features.

Modification

The source and target meta-models of the transformation are different from the meta-model of the patterns. The latter should then have a different name or be in a different namespace. Furthermore, although all the concepts of the original meta-models are present in the pattern meta-model, the individual elements are completely modified. The modifications that need to be applied depend on whether we want to obtain pre-condition (*i.e.,* NAC and LHS) or post-condition (*i.e.,* RHS) pattern specifications.

In the pre-condition pattern specifications, one does not want to assign an actual value to attributes of the original meta-model classes, but specify a constraint. For example, a pattern requiring a place to have at least one token cannot assign a value to the token attribute of the *MTpre_Place* class. On the contrary, the *MTpre_token* attribute holds a constraint on the *token* value of a matched model element. Thus, we need to replace the respective types of attributes with the type "*constraint*". This allows the transformation designer to specify constraints for element features, such as

PreNode(1).name="NEXT"

in the NextPlace rule of Figure 5.2. For post-condition pattern specification we need to allow actions rather than constraints, so that the transformation designer can assign values of attributes, among other potential actions. In rule NextPlace , the "=" in the RHS part of the rule is an assignment action rather than an equality check. Note that the same naming and modification scheme is applied to classes, associations, and role names.

Finally, we sometimes need to modify the concrete syntax of language elements whose size or natural layout is not conducive for specifying patterns. For example, in a modelling environment for designing Petri nets, tokens (represented by a dot) will be centred inside a place (represented as a circle). If places and tokens were modelled by classes in the Petri net meta-model, the concrete syntax of the association between these two concepts would be a topological constraint on the visual syntax. In this case, the presence of the association is implicit to the modeller since the token is centred in the place. However for the transformation designer, that association must be explicitly represented and accessible to, for example, assign values to the augmented attributes or bind it to a pivot. Also, elements which are normally not rendered at all, such as instances of formerly abstract classes or association ends, need to be assigned some concrete syntax so that they may be referred to in a visual manner.

5.3.5 Implementation of the RAM process

The RAM process was integrated in *AToM*^{3}'s meta-modelling process. After having defined the meta-model of the source and target domains of the transformation, the transformation engineer can request to generate the relaxed, augmented, and modified (RAMified) version of the meta-models, one for the meta-model of the pre-condition patterns and one for the meta-model of the pre-condition patterns. He can then load the meta-model of the transformation units, consisting of LHS, RHS, and optionally NAC components of a rule or query. Patterns can then be constructed for each component as illustrated in Figure 5.6. The following summarizes how each step of the RAM process is implemented:

Relaxation

Concretize abstract classes: changed the *isAbstract* attribute of all classes. This is accessed from the abstract syntax graph (ASG) nodes: the elements at the M_3 layer.

Reduce multiplicities: changed the *cardinality* attribute of all associations.

Constraint filtering: manual. The modeller must modify/remove them prior to the RAM generation.

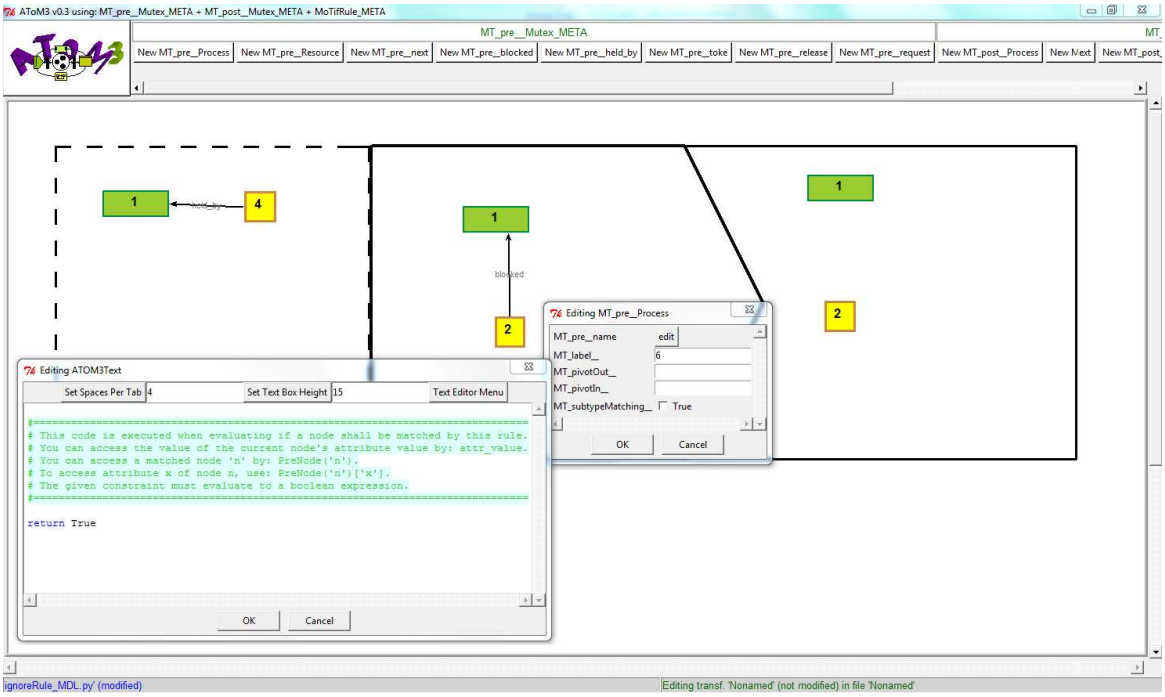


Figure 5.6: A transformation rule model in *AToM*[3].

Augmentation

Pattern elements re-typing: all classes inherit from the pre-defined classes `MTpre_Element` or `MTpost_Element`.

Connection with generic elements: automatically inherited from `MTpre/post_Element`.

Augmented attributes: added the attributes to all classes (sub-type matching, pivot passing ...).

Augmented constraints: added label uniqueness constraint.

Concrete syntax: all copied from the original meta-model. Added a default one for abstract classes annotated with the original class name.

Modification

Namespace: the process creates a new meta-model with the same name prefixed with `MTpre__` or `MTpost__`.

Pre-condition attribute types: all attributes are typed by the type of constraint language[3]. Their value correspond to the body of a constraint method (`constr`) such that:

$$\texttt{constr: object} \times \texttt{Graph} \rightarrow \texttt{boolean}.$$

[3]The type of constraint and action languages are implemented as strings for the moment.

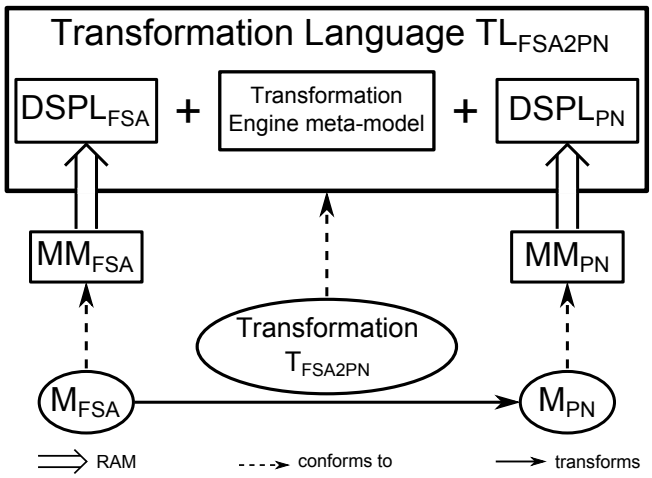


Figure 5.7: Schema of domain-specific transformation languages.

Post-condition attribute types: all attributes are typed by the type of action language. Their value correspond to the body of an action method (`action`) such that:

$$\texttt{action: object} \times \texttt{Graph} \rightarrow \texttt{object} \times \texttt{Graph}.$$

The original attribute types are preserved thanks to getters and setters added to the RAMified meta-model. The setter takes as input the new value of type `object` and makes sure it is a (sub-)type of the original attribute.

Adaptation of concrete syntax: manual.

To summarize, this section has discussed various alternatives for enabling transformation designers to make use of customized pattern specification languages and environments. We proposed the semi-automated generation of customized, maximally constrained, meta-models based on relaxation, augmentation, and modification operations. Figure 5.7 depicts how a transformation is defined with this approach. Following the finite state automata to Petri nets example, we call T_{FSA2PN} the (transformation) model mapping an FSA model M_{FSA} to a Petri net model M_{PN}. T_{FSA2PN} is the transformation model described in Section 5.2.2 and M_{FSA} and M_{PN} are represented in Figure 1.5. The two models conform to their respective meta-models MM_{FSA} and MM_{PN}. Applying the technique described in this section, *domain-specific pattern languages* are generated from these meta-models, namely $DSPL_{FSA}$ and $DSPL_{PN}$ respectively. The meta-models of the patterns (specific to this transformation) combined with the meta-model of the general transformation language, form the *transformation language* TL_{FSA2PN}. The transformation T_{FSA2PN} is thus completely modelled and conforms to its meta-model TL_{FSA2PN} shown in Figure 5.5.

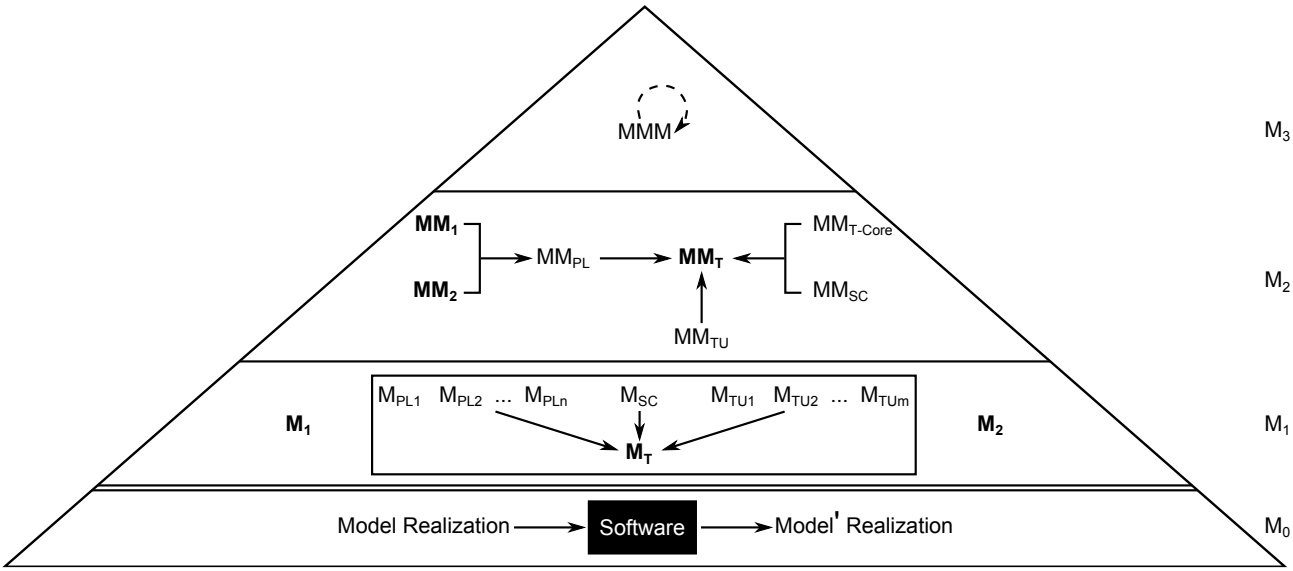


Figure 5.8: The meta-layers of a transformation language.

5.4 Engineering of Model Transformation languages

Let us now take a step back and look at the chapters of this part of the thesis as a whole. In Chapter 3, we described the building blocks of a transformation such that, when combined with an appropriate scheduling language, a transformation engine is completely described. *T-Core* is provided as a collection of components with a dedicated API so that a transformation language engineer can combine them with his favourite scheduler. *T-Core*'s syntax is modelled in the UML class diagram formalism. Chapter 4 demonstrated how a transformation engine can be implemented to allow executing transformation models. It is described using a dedicated programming language, such as Python. Finally, in the current chapter, both the transformation units and the patterns used to specify transformation mapping are modelled. The meta-model of the transformation units is integrated with the meta-model of the modelling framework, such as the one of *AToM*^{3}. Also, following MPM principles, one should use the most appropriate language to describe the patterns of a transformation unit. The most appropriate formalisms are undoubtedly the original DSLs of the source and target domains, albeit adapted for model transformation tasks. Thanks to the RAM process, the level of abstraction at which patterns can be described is the same as the one a domain-specific engineer works at for designing models. Therefore all aspects of a transformation language have been modelled explicitly at the most appropriate level of abstraction, using the most appropriate formalisms. Figure 5.8 shows an integrated view of (1) how to engineer a transformation, (2) how to engineer a transformation language, and (3) how this MPM model transformation engineering methodology fits with the standard model-driven architecture. Note that it describes how custom-built transformation languages tailored to the needs of the domain-specific engineer are modelled.

At the very top of the meta-layers hierarchy is the **meta-meta-model** *MMM*, a MOF-like formalism *e.g.*, the Ecore meta-model. It is the formalism that allows us to describe meta-models of

modelling languages and, in particular, transformation languages. This layer is often bootstrapped, *i.e.,* the meta-model of *MMM* is *MMM* itself.

The **meta-model** MM_T of a transformation language consists of three components. First, a meta-model MM_{TU} of the *transformation units* is required. These basic units can take the form of rules, queries, functions, etc. It is up to the transformation language engineer to define them as they indicate the building blocks used by the transformation engineer to design a transformation model. Second, the engine component is defined by selecting *transformation primitives* encountered in the meta-model of *T-Core* and weaving them with a *scheduling language*. Depending on the type of transformation required (query, synthesis, translation, simulation, migration, synchronization, etc), the transformation language engineer will combine the appropriate *T-Core* components with the most appropriate formalism MM_{SC} for the scheduler. The transformation language may grant direct access to the scheduler so that the transformation engineer can explicitly specify the order in which transformation units are executed. Another possibility is to hide the scheduler from the transformation designer and internally predefine the scheduling. The third component is the pattern language MM_{PL} that describes how the transformation engineer specifies the body of a transformation unit. As far as rules are concerned, the pattern language can be semi-automatically generated by the RAM process. This allows the transformation engineer to specify rule patterns at the same level of abstraction and using the same constructs as the DSMs he builds rely on. In addition to the meta-model of the customized transformation language MM_T, the source and target meta-models, respectively MM_1 and MM_2, are also present in the meta-layer M_2.

All the elements in the M_1 layer conform to elements in the M_2 layer. M_1 represents a model of the system to transform and conforms to MM_1. M_2 is the result[4] of this transformation and conforms to MM_2. M_T is the **model** that the transformation engineer designs to manipulate instances of MM_1. It conforms to MM_T. To achieve that, the transformation engineer must define patterns (M_{PLi} instances of MM_{PL}) that are embedded in transformation units (M_{TUi} instances of MM_{TU}), such as rules. Furthermore, he can assign a specific schedule (M_{SC} instances of MM_{SC}) for executing rules.

Although we are working at higher levels of abstraction than source code, one should not forget that we are building **software products** after all. Thus, at the M_0 level, the actual system modelled by M_1 is input to the transformation software modelled by M_T that modifies it as intended by the transformation model. The resulting system is modelled by M_2.

5.5 Related Work

The idea of employing languages tailored to certain domains is not a new one [Lan66], and numerous DSLs have been devised since. Meanwhile, the DSL approach has also propagated into the model engineering community. Actually, considerable controversy exists about of whether MDE should focus on a general-purpose modelling language (such as UML) or make use of a number of smaller, domain-specific languages.

The need to provide domain-specific model transformation languages was first pointed out in

[4] M_2 may still conform to MM_1, but we make this distinction to be more general.

[RKR+06]. The authors' approach is to generate an execution environment for DSLs. The framework, called Marius, takes as input the EBNF grammar describing the abstract syntax of the DSL, as opposed to our approach which uses meta-models described as UML class diagrams with a graphical concrete syntax. Marius is not MOF-compliant. The model transformation languages produced by their approach are template-based, rather than rule-based. In our opinion, rule-based languages are more declarative and hence reduce the cognitive effort of the domain-specific engineer for designing transformations. Most importantly, the environment generated by Marius is not modelled and hence does not allow for higher-order transformation.

In his Ph.D. thesis, Van Gorp [VG08] proposes to model model transformations in an object-oriented manner. The advantage is that a model transformation is then considered as an object-oriented model where refactoring, refinement and synchronization are well understood. In contrast, our approach models model transformations as DSMs which reduces accidental complexity and the effort required by a developer to map domain-specific information onto the transformations to implement.

Bézivin *et al.* explicitly model transformations with "transformation models" [BBG+06] but for capturing the relations maintained by transformations rather than supporting their customization or generation.

The need to relax conformance rules occurs in other areas as well. Morin *et al.* also relax an original meta-model in order to allow the formulation of pointcut specifications in the context of aspect-oriented modelling [MBJR07]: (1) invariant, pre-, and post-conditions are removed, (2) all features of all classes are changed to optional, (3) abstract model items are removed. The approach is similar to our meta-model relaxation, but is less generic and currently works only on class diagrams and Java programs. The RAM process described in this chapter assumed that the meta-models are defined as UML class diagrams. This is not really a restriction since most CASE tools define meta-models in this language.

Levendovszky *et al.* capture domain-specific design patterns which also inherently are fragments of proper models [LLM09]. Instead of creating a relaxed version of the meta-model, they use relaxed conformance, *i.e.*, "relaxed instantiation". This allows them to use one original language definition to check both proper models and design patterns. Since they only need to support this one variant of conformance checking, this is a viable approach. In general, however, the explicit modelling of transformations may require a multitude of conformance levels, making the relaxation of meta-models a more attractive option (see Section 5.3.3). Levendovsky *et al.*, furthermore, observe that simply setting all minimal multiplicities to zero will allow the formulation of fragments which cannot be completed to proper models. They suggest detecting such fragments by using constraint solving. This approach is applicable in our context as well and could be realized by adding corresponding constraints to the relaxed meta-models.

5.6 Conclusion

In this chapter, we demonstrated the benefits of explicitly modelling transformations and proposed ways to economically enable their definition. As a result, model transformation languages can be tai-

lored to the source and target languages of transformations, in the same way as DSLs are tailored to model domains. While it is not necessary to explicitly model *all* aspects of transformation definitions, we have illustrated that there are benefits associated with each such step. The explicit modelling of pattern specifications allowed the semi-automatic generation of customized pattern specification language definitions based on the components of relaxation, augmentation, and modification. It thus provided a cost-effective way to obtain customized transformation development environments. The transformations we presented are furthermore applicable in a wide range of similar contexts. This and their re-usability is a direct result of explicitly modelling all aspects of transformations including their control flow aspects. We provided a tool-support and a methodology to design custom-built model transformation languages *i.e.,* the automatic generation of domain-specific model transformation development environments.

The automatic meta-modelling proposed in this chapter focuses on the pattern language customized to fit with the domain of application of transformations. Another interesting research direction is to investigate how to *automatically* define a scheduling language tailored to the type of transformation intended. A possible direction could be language design patterns [LLM09] or using transformations by example [KWSB10].

A possible problem with the proposed approach is that, for every language used in a transformation, both the language meta-model MM and its RAMified version MM_R will co-exist. Thus a co-evolution problem may arise if, on the one hand, MM is later modified and evolves to a meta-model MM'. Certainly, MM_R should be adapted accordingly. For example, if the RAM process is implemented as a model transformation (operating at the meta-model level), co-evolution techniques such as in [SK04, HBJ09, MV11] can be applied. Another possibility is to define an incremental model transformation for RAM and thus MM_R is modified automatically [RBÖV08]. On the other hand if MM_R evolves, MM should remain unchanged. That is because typical changes in MM_R do not modify core concepts in the language, but are specific to transformation purposes only. Therefore future improvements should only consider automatically adapting the RAMified meta-model when the original meta-model evolves.

Part III

A Modular Timed Graph Transformation Language

"If you're asking your kids to exercise, then you better do it, too. Practice what you preach."

Bruce Jenner

6

Modelling DEVS and its Simulators

This chapter presents some background on the *Discrete Event system Specification* (*DEVS*) [Zei84] formalism used for modelling and simulation. It is the semantic domain of the transformation language that will be introduced in the following chapter. On the one hand, *DEVS* allows one to model a system focusing on its behaviour. On the other hand, *DEVS* models are executed through simulation. Following MPM principles, this chapter provides a model of not only the *DEVS* language but also a model of its simulation protocol. An extension of this MDE approach is to replace the sequential simulation model by a distributed simulator. Hence, the structure and behaviour of a distributed simulator for the *DEVS* formalism is modelled explicitly. Simulation-based analysis of this model of the simulator allows for the investigation of measures such as reliability and performance across different alternative designs and hence for optimal design. In particular, using a model of a distributed simulator allows one to simulate scenarios such as failures of computational and network resources, which can be hard to realize in reality. We demonstrate our model-based approach by modelling, simulating, and ultimately synthesizing a distributed DEVS simulator.

6.1 Introduction

Distributed environments overcome many of the limitations imposed by single processor implementations of large-scale tasks. Simulation of large models consumes a lot of computational and memory resources. In this chapter we focus on models in the Discrete Event system Specification (DEVS) [Zei84] formalism. Our experiences show that this formalism, thanks to its modularity and compositionality, is highly suitable for large-scale tasks, such as model transformation [SV08a].

Simulators in general and distributed, discrete-event simulators in particular, are typically realized using different implementation languages and hardware platforms (processing as well as network resources). This hampers realistic performance comparisons between simulator implementations. Furthermore, details of the distributed algorithms used are commonly present in the form of code rather than explicitly modelled which hampers re-use and rigorous analysis. A distributed environment for the simulation of DEVS models is attractive for several reasons. Although not the primary goal of distributed simulation, model execution time can be reduced. Also, the limited memory issue for a single machine can be overcome and models with an extremely large state-space can be handled. Our main goal of distributing DEVS models is for interoperability. Handling geographically distributed

users and/or resources (*e.g.*, databases or specialized equipment) and exploiting greater data handling capability through specialized nodes is our main interest. As a side effect, this allows integrating simulations running on different platforms. Furthermore, properties of distributed systems such as fault-tolerance capabilities become accessible.

In the next section, we review the essence of the classic DEVS formalism from an MPM perspective as well as its simulation protocol. Section 6.3 proposes a DEVS model representing a DEVS simulator. It is then extended to a distributed DEVS simulator together with preliminary fault-tolerance capabilities. From this model, we can synthesize or build a distributed DEVS simulator, implemented on a dedicated middleware. Section 6.4 outlines the implementation used for this work. In Section 6.5, we calibrate the modelled distributed simulator with values gathered from the implemented distributed DEVS simulator. Then, simulation experiments on the modelled distributed DEVS simulator allow us to determine optimal values for the variables and thus calibrate back the implemented simulator to behave optimally for the given input model. Section 6.6 compares some of the current distributed DEVS implementations and shows how our modelling and simulation-based approach can be considered as a generalisation of these different implementations.

6.2 Classic DEVS

We introduce the classic *Discrete EVent system Specification* formalism and review its simulation definition. The notation introduced here will be used for the remainder of the article.

6.2.1 Formalism

The *DEVS* formalism was introduced in the late seventies by Zeigler as a rigorous basis for the compositional modelling and simulation of discrete event systems [Zei84]. It has been successfully applied to the design, performance analysis, and implementation of a plethora of complex systems such as peer-to-peer networks [XBZPZ08], transportation systems [LLC04], and complex natural systems [FB04].

Figure 6.1 shows a possible meta-model of DEVS in UML Class Diagram notation. A DEVS model (the abstract class Block) is either an AtomicBlock or a CoupledBlock. An atomic model describes the behaviour of a timed, reactive system. A coupled model is the parallel composition of several DEVS sub-models which can be either atomic or coupled. Sub-models have *ports*, which are connected by channels (represented here by the associations between the different ports). Ports are directional and are either Inport or Outport. The abstract classes (In/Out)port can be instantiated as an Atomic(In/Out)port or a Coupled(In/Out)port, respectively. Ports and channels allow a model to receive and send events (any sub-class of Event) from and to other models. A channel must go from an output port of some model to an input port of a different model, from an input port of a coupled model to an input port of one of its sub-models, or from an output port of a sub-model to an output port of its parent model, as depicted by the associations of Figure 6.1. Note that the dynamic semantics of *DEVS* is not expressed by the meta-model. It will be informally presented hereafter.

An **atomic DEVS** model is a structure $\langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$. S is a set of sequential **states**. X is a set of allowed **input events**. Y is a set of allowed **output events**. There are two types of transitions

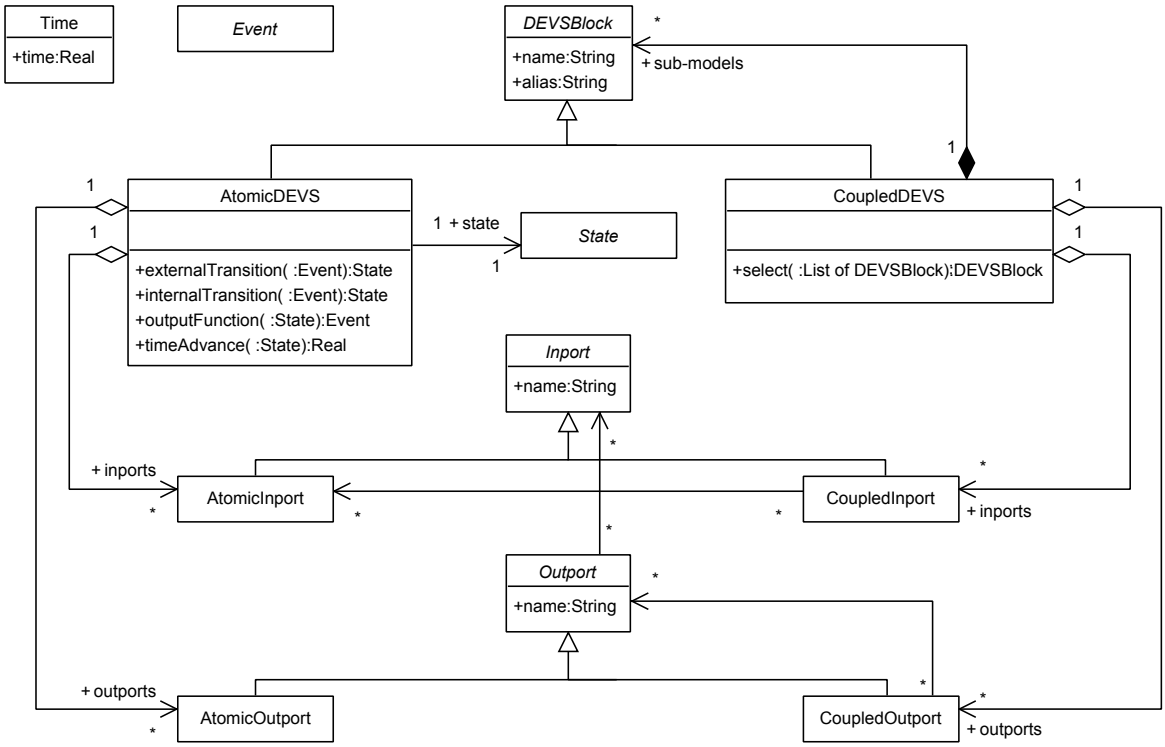


Figure 6.1: The DEVS meta-model.

between states: $\delta_{int} : S \to S$ is the **internal transition function** and $\delta_{ext} : Q \times X \to S$ is the **external transition function**. Associated with each state are $\tau : S \to \mathbb{R}^+ \cup \{+\infty\}$, the **time-advance** function and $\lambda : S \to Y$, the **output function**. In this definition, $Q = \{(s,e) \,|\, s \in S, 0 \leq e \leq \tau(s)\}$ is called the **total state space**. For each $(s,e) \in Q$, e is called the **elapsed time**, the time the system has spent in a sequential state s since the last transition. The state of the atomic DEVS is initialized to $q_0 = (s_0, e_0)$, but in the sequel we only consider s_0 for simplicity. When the time is infinite, it is said to be *passivated* and when it is zero, it is said to be *transient*.

Informally, the operational semantics of an atomic model is as follows: the model starts in its initial state. It will remain in any given state for as long as the time-advance of that state specifies or until input is received on some port. If no input is received, after the time-advance of the state expires, the model first (before changing state) produces an output event as specified by the output function. Then, it instantaneously jumps to a new state specified by the internal transition function. However, if an input event is received before the time for the next internal transition, then it is the *external transition* which is applied. The external transition depends on the current state, the time elapsed since the last transition, and the input event.

To illustrate the atomic DEVS concept, consider a user of a transformation system, who receives a graph G every time a rule is applied. Furthermore, after analyzing the graph, he outputs a decision encoded as an integer $n \in \mathbb{N}$ five time units later. We model the user's behaviour by an atomic model m. Its state space is $S = \{IDLE, ANALYZING\} \times \mathbb{N}$; the state is also used to store the computed

integer. m can only receive a graph as input, hence X is the set of all graphs. It also sends an integer as output, hence $Y = \mathbb{N}$. S is initially in $IDLE$ mode. Upon reception of a graph G, m applies the external transition $\delta_{ext}((IDLE, e), G) = ANALYZING$. It stays in $ANALYZING$ mode, until the time advance $\tau(ANALYZING) = 5$ expires. Then, m outputs $\lambda(ANALYZING) = n \in \mathbb{N}$ and subsequently applies the internal transition $\delta_{int}(ANALYZING) = IDLE$. m then stays in this mode until an external input is received, since $\tau(IDLE) = +\infty$.

A **coupled DEVS** model named C is a structure $\langle X, Y, N, M, I, Z, \Xi \rangle$ where X and Y are as before. N is a set of **component names** (or labels) such that $C \notin N$. $M = \{M_n | n \in N, M_n$ is a DEVS model (atomic or coupled) with input set X_n and output set $Y_n\}$ is a set of DEVS **sub-models** or **components**. The set of **influencees** of a component labelled n is I_n, denoting all components influenced by n. $I = \{I_n \mid n \in N, I_n \subseteq N \cup \{C\}\}$ is the set of all **influencees** describing the coupling network structure. That is, for a given model label m, I_m represents all models (denoted by their label) that may receive an event output from model m. $Z = \{Z_{i,n} | \forall n \in N, i \in I_n, Z_{i,n} : Y_i \to X_n \vee Z_{C,n} : X \to X_n \vee Z_{i,C} : Y_i \to Y\}$ is a set of **transfer functions** between connected components. $\Xi : 2^N \to N$ is the **select** or tie-breaking function. 2^N denotes the powerset of N (the set of all sub-sets of N).

The connection topology of sub-models is expressed by the influencee set I_n of the components. Note that for a given model n, this set includes not only the external models that receive inputs from n, but also its own internal sub-models that produce its output (if n is a coupled model). Transfer functions ($Z_{i,n}$) represent output-to-input translations between components. They can be thought of as channels that make the appropriate type translations. For example, a "departure" event output of one sub-model is translated to an "arrival" event on a connected sub-model's input. The select function Ξ takes care of conflicts as explained below.

The semantics for a coupled model is, informally, the parallel composition of all its sub-models. A priori, each sub-model in a coupled model is assumed to be an independent process, concurrent to the rest. There is no explicit method of synchronization between processes. Blocking does not occur, except if it is explicitly modelled by the output function of a sender, and the external transition function of a receiver. There is however a *serialization* whenever there are multiple sub-models that have an internal transition scheduled at the same time (this set is referred to as the **imminent set**). The modeller controls which of the conflicting sub-models undergoes its transition first by means of the select function.

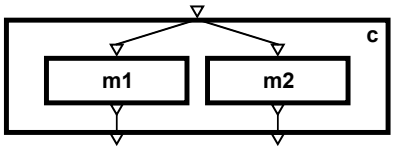


Figure 6.2: A hierarchical DEVS model.

To illustrate the coupled DEVS concept, we extend the previous example by involving different decision makers. It is visually depicted in Figure 6.2. Suppose we now have two decision blocks m_1 and m_2, where m_i deterministically outputs i. The task is to output the computed numbers when a graph is received. For that, we construct a coupled model c where $X = \{G\}$ and $Y = \mathbb{N}$. We label

the two inner models $M = \{m_1, m_2\}$ by $N = \{1,2\}$ respectively. We then connect the inport of c to the inport of m_1 and the inport of m_2. We also connect the outports of both m_1 and m_2 to the outports of c. Therefore, $I = \{I_1 = \{c\}, I_2 = \{c\}, I_c = \{1,2\}\}$. As for the transfer function, we define $Z_{c,1}(G) = Z_{c,2}(G) = G$ for the input-to-input channels and $Z_{1,c}(n) = Z_{2,c}(n) = n$ for the output-to-output channels. At simulation time (run-time), after G is received, m_1 and m_2 are scheduled to output and then perform their internal transition at the same time, since their time advance is 5 and they received the input at the same time. The select function then chooses which inner model will execute first, *e.g.*, set $\Xi(\{1,2\}) = 1$.

In this thesis, we have used a *DEVS* simulator called *pythonDEVS* [BV01], grafted onto the object-oriented scripting language Python.

6.2.2 The DEVS simulation protocol

To simulate a DEVS model, a **solver** is attached to each atomic DEVS, a **coordinator** is attached to each coupled DEVS and a **root coordinator** initiates, ends, and keeps the simulation running. Figure 6.3 illustrates the simulation entities (modelled in DEVS) representing the simulation of a client-server model, depicting the tree hierarchy of the coupled and atomic models. It also shows the corresponding hierarchy of simulation entities.

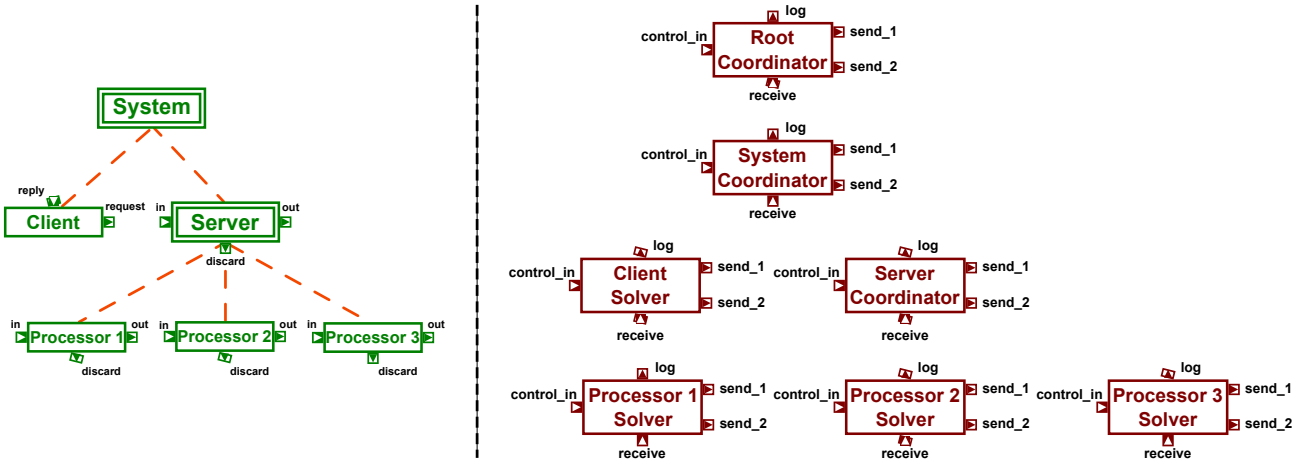


Figure 6.3: The hierarchical model of client-server example (on the left) and and its corresponding simulation entities as DEVS models for the distributed simulation (on the right). In this case, there are two machines involved.

In an atomic DEVS solver, the last event time t_L as well as the local state s are kept. In a coordinator (the coupled DEVS' solver), only the last event time t_L is kept. The next event time t_N is sent as output of either solver and is stored in the solvers. This requires consistent (recursive) initialization of the t_Ls. The t_N allows one to check whether the solvers are appropriately synchronized. The operation of an abstract simulator (solver or coordinator) involves handling four types of messages sent at time from a *source* DEVS model to a *target* DEVS model:

- `INIT`: $(s_0, source, target, t)$ message holding the state at which the model and the simulation time t are (re)set;
- `*`: $(source, target, t)$ message to indicate that an internal transition is due, at simulation time t;
- `X`: $(x, source, target, t)$ message to carry the external input information x, at simulation time t;
- `Y`: $(y, source, target, t_N)$ message to carry the output information y; and
- `DONE`: $(source, target, t_N)$ message to acknowledge that one of the above messages has successfully been handled.

When a coordinator receives a `*` message, it selects an imminent component i^* from the imminent set by means of the select function Ξ specified for the coupled model it is associated to. The message is then routed to i^*. When a solver receives a `*` message, it generates an output message `Y` based on the old state of the atomic model it is associated to. It then computes the new state by means of the internal transition function. Note how DEVS output messages are only produced while executing internal events. When a simulator outputs a `Y` message, it is sent to its parent coordinator. The coordinator sends the output, after appropriate output-to-input translation ($Z_{i,n}$), to each of the influencees of i^* (if any). If the coupled model C itself is an influencee of i^*, the output, after appropriate output-to-output translation ($Z_{i,C}$), is sent to C's parent coordinator.

When a coordinator receives an `X` message from its parent coordinator, it routes the message, after appropriate input-to-input translation, to each of the affected components. When a solver receives an `X` message, it executes the external transition function of its associated atomic model.

After processing an `X` or `Y` message, a solver sends a `DONE` message to its parent coordinator to prepare a new schedule. Once a coordinator has received `DONE` messages from all its components, it sets its next-event-time t_N to the minimum t_N of all its components and sends a `DONE` message to its parent coordinator. This process is recursively applied until the top-level root coordinator receives a `DONE` message.

To run a simulation experiment, the initial conditions on t_L, s, and t_N must first be set in all simulators of the hierarchy. If t_N is kept in the simulators, it must be recursively set too. Once the initial conditions are set, the main loop which sends a `*` message and waits for a `DONE` message is executed until a termination condition is satisfied.

The classic *DEVS* formalism has some limitations such as:

- A conflict may arise in the occurrence of simultaneous internal and external events. In this case, the external transition has precedence by default.
- The select function is an artificial legacy of the semantics of traditional sequential simulators based on an event list. It serializes simultaneously triggered internal transitions.
- Therefore, the potential for parallel implementation is limited to only external transitions.
- It is not possible to explicitly describe variable structure since, once designed, the network of components and their connections is fixed.

Some of these limitations are resolved in the widely used parallel DEVS formalism [CZ96]. It allows

collision handling and parallelism of DEVS models. A confluent transition function $\delta_{con} : Q \times X \rightarrow S$ is added to the classic atomic DEVS model. It is triggered when an atomic block receives an external event at the time of its internal transition. The event set X is now a set of bags of events since atomic simulators may output events concurrently. As for the coupled model, the select function is removed since all delta functions run in parallel.

6.3 Modelling a Simulator

In this section, we show how the DEVS simulator described above can be explicitly modelled in DEVS. First, each of the simulation entities is represented as an atomic DEVS model. Then, we model a distributed simulation engine for an arbitrary DEVS model. The model takes into consideration the simulation entities, the different machines, and the communication layer. Figure 6.4 illustrates the cluster integrating all the different modelled entities. A star on an inport is a shorthand notation to represent channels incoming from all the components of the same type as the source of the channel (*e.g.,* every Simulator's log port is connected to the log_in port of the Log). A star on an outport is a shorthand notation to represent channels outgoing to all the components of the same type as the target of the channel (*e.g.,* the control_out port of the Master is connected to every Simulator's control_in port). Dots between ports with a generic label is a shorthand notation to represent as many ports on the host component as there are components of the same type as the source/target of the channel (*e.g.,* the outport pattern labelled simID on Machine denotes one such outport per AS, CO, and RC).

6.3.1 The Simulation Entities

The simulation model is composed of Atomic Solvers, Coordinators, and a Root Coordinator, each modelled as an atomic DEVS block. Solvers and coordinators hold their corresponding model in their state. Each of these simulator models has one inport receive and as many send outports as there are machines (this is needed since DEVS models lack variable structure). Sending and receiving *simulation messages* `*`, `X`, `Y`, and `DONE` (encoded as events) is performed via these ports. The simulator also receives a *reallocation message* indicating to which machine the simulator interoperates. Additional ports handle *reallocation messages*, *control messages* (stop and resume), and *logging messages* for fault-tolerance purposes.

The Atomic Solver Model

An Atomic Solver (AS) is an atomic DEVS model. Its state is composed of:

- the atomic DEVS model M it simulates;
- a unique identifier `id` such that `map`(id) uniquely identifies M;
- the identifier of the parent of M, `parentId`;
- the last event time t_L;
- the next event time t_N;
- the output set Λ of M;

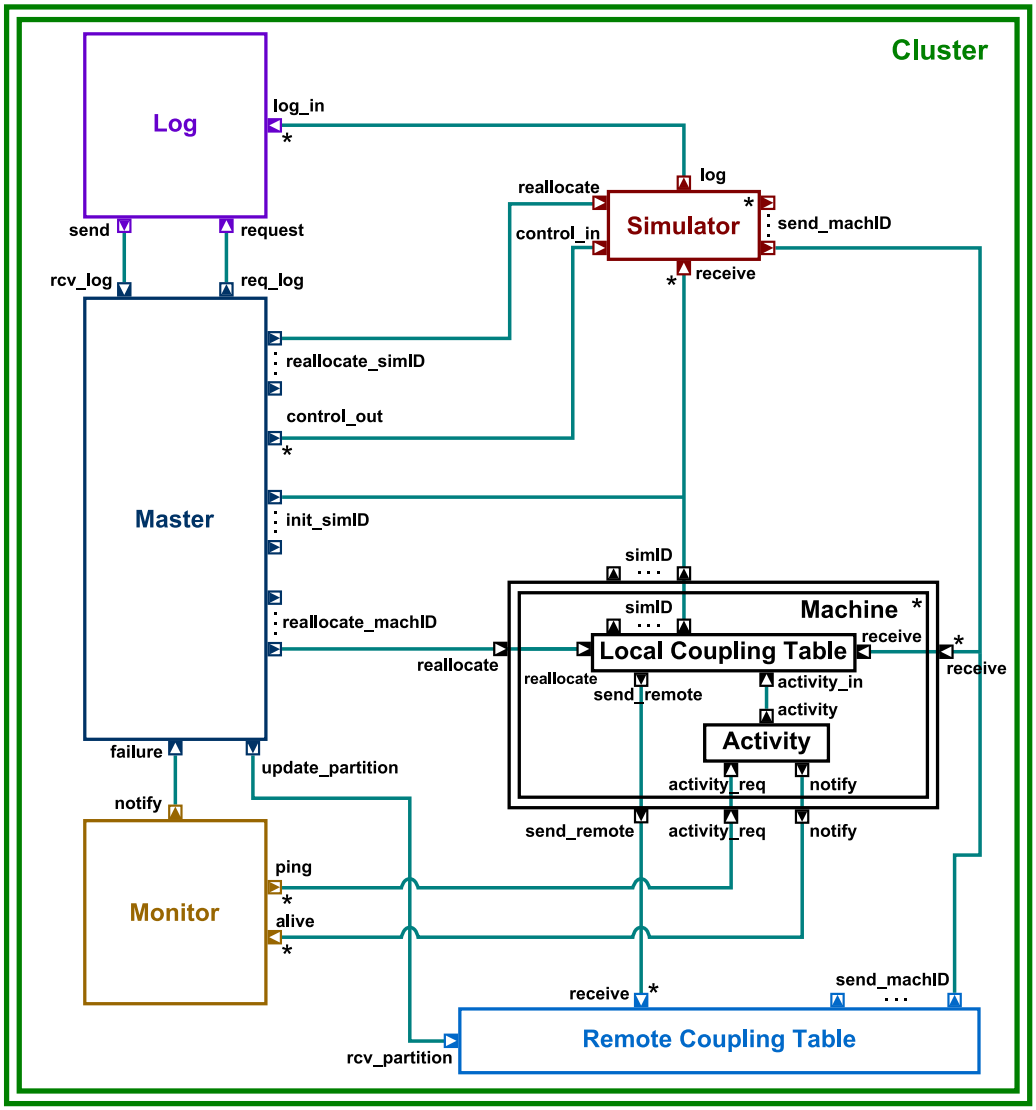


Figure 6.4: A DEVS model representing a distributed environment for a DEVS simulation.

- a bit-vector `activePorts` identifying which outport is currently active (its size is determined by the number of machines); and
- the mode μ the AS is in: *PAUSED* or *RUNNING*.

An AS is reactive. It waits for either a * event to process the internal transition function of M or an X event to process the external transition function of M. The state of the AS is updated accordingly in δ_{ext} as described in Algorithm 17. It also receives a special simulation message for initialization, `INIT`, holding the state the AS should be (re)set to. The DEVS model M is set at instantiation-time of the AS and `activePorts` is set to the zero vector. Its mode is triggered by a control message χ received from the control_in port and, from the reallocate inport, it receives reallocation message ρ

indicating the (possibly) new active outport. In the algorithm, we distinguish structural elements of the AS from those of its model M by annotating the latter with $\bullet^M$.

Algorithm 17 The external transition function $\delta_{ext}((s,e),x)$ of an AS

UPON RECEIVE INIT: $(s_0, source, target, t)$ **do**
  if $s_0 \neq$ `nil` **then**
    $s^M \leftarrow s_0$
  else
    $id, parentId, \Lambda \leftarrow target, source, \emptyset$
    $t_L \leftarrow t - M.e$
    $t_N \leftarrow t_L + \tau^M(s^M)$
    $\mu \leftarrow PAUSED$
  end if

UPON RECEIVE X:$(x, source, target, t)$ **do**
  $s^M \leftarrow \delta^M_{ext}((s_M, t - t_L), x)$
  $t_L \leftarrow t$
  $t_N \leftarrow t_L + \tau^M(s^M)$
  $M.e \leftarrow 0$

UPON RECEIVE *:$(source, target, t)$ **do**
  $\Lambda \leftarrow \Lambda \cup \{\lambda^M(s^M)\}$
  $s^M \leftarrow \delta^M_{int}(s^M)$
  $t_L \leftarrow t$
  $t_N \leftarrow t_L + \tau^M(s^M)$
  $M.e \leftarrow 0$

UPON RECEIVE χ **do**
  if $\chi ==$ PAUSE **then**
    $\mu \leftarrow PAUSED$
  else if $\chi ==$ RESUME **then**
    $\mu \leftarrow RUNNING$
  end if

UPON RECEIVE ρ **do**
  $activePorts \leftarrow (0, \ldots, 0)_{|activePorts|}$
  $activePorts[\rho] \leftarrow 1$

The internal transition function δ_{int} clears Λ. If $\mu = RUNNING$ and either $\Lambda \neq \emptyset$ or one of * or X was received, then $\tau(s)$ is the real execution time spent for applying λ^M, δ^M_{int}, or δ^M_{ext}. On the other hand, if INIT or χ was received, then $\tau(s) = 0$. Otherwise, the time advance of the AS is infinity.

In case $\Lambda \neq \emptyset$, the output function λ first sends Y: $(\Lambda, id, parentId, t_L)$, followed by DONE: $(id, parentId, t_N)$. Note that the output is sent via the currently active outport. The output function is extended to allow logging for handling fault-tolerance (see section 6.3.3): whenever a simulation message is received, the new state of the AS is serialized and output through the log port.

The Coordinator Model

A *Coordinator* (CO) is an atomic DEVS model. Its state is the same as for the AS with additional information:

- M is now a coupled DEVS model;
- the list of events to be processed L;
- the list of children (simulators this CO coordinates down the simulators hierarchy) *children*;
- the list of children still processing an event activeChildren; and
- Ψ is the output set of X messages for its children (not to be confused with Λ).

The external transition function of the CO is modified from the one of an AS as described in Algorithm 18. *activePorts*,*children*, and *activeChildren* are set at instantiation-time of the CO.

The internal transition function δ_{int} clears Λ and Ψ. The time advance of the CO is infinity unless $\mu = RUNNING$ and either $\Lambda \neq \emptyset$, $\Psi \neq \emptyset$, INIT was received, or χ was received. In this case, $\tau(s) = 0$.

If INIT was received, the output function of the CO produces INIT: $(nil, id, \text{children}, t)$, where t is the same time in the INIT message the CO received. Having $s_0 = nil$ in the INIT message means that the simulator is starting (usually sent only once). If $\Psi \neq \emptyset$, the CO sends X: $(\Lambda, id, \text{activeChildren}, t)$, where t is the same time in the X message the CO received. Otherwise, if $\Lambda \neq \emptyset$, the output function first produces Y: $(\Psi, id, \text{parentId}, t)$, where t is the same time in the X or Y message the CO received. However, the CO sends *: (Λ, id, i^*, t) when it received a * message with time t. Finally, if activeChildren $= \emptyset$ then DONE: $(id, \text{parentId}, t_N)$ is sent.

The Root Coordinator Model

A *Root Coordinator* (RC) is also an atomic DEVS model. Its state consists of:

- *id*, *activePorts*, and *children* as defined before;
- the current simulation time T;
- a termination condition θ.

When the external transition function receives DONE: $(source, target, t)$ T is set to t. However, if θ is not satisfied, the simulation is stopped. The time advance function τ of the RC is always infinity except when DONE was received (in which case it evaluates to 0). The output function returns INIT: $(nil, id, id, 0)$ when the simulation starts and *: $(id, \text{children}, T)$ when DONE was received.

Algorithm 18 The external transition function $\delta_{ext}\left((s,e),x\right)$ of a CO

UPON RECEIVE INIT: $(s_0, source, target, t)$ **do**
  if $s_0 \neq nil$ **then**
    $s^M \leftarrow s_0$
  else
    $id, parentId, \Lambda, \Psi \leftarrow target, source, \emptyset, \emptyset$
    $t_L \leftarrow t$
    $t_N \leftarrow +\infty$
    $\mu \leftarrow PAUSED$
  end if

UPON RECEIVE X:$(x, source, target, t)$ **do**
  $activeChildren \leftarrow activeChildren \cup \left\{i | \mathtt{map}(id) \in I_i^M\right\}$
  $\Psi \leftarrow \bigcup_{i \in activeChildren} Z^M_{\mathtt{map}(id),i}(x)$
  $t_L \leftarrow t$

UPON RECEIVE *:$(source, target, t)$ **do**
  $immList \leftarrow \{i | (i,T) \in L \wedge T = t\}$
  $i^* \leftarrow \Xi(immList)$
  $activeChildren \leftarrow activeChildren \cup \{i^*\}$
  remove(i^*, L)
  $t_L \leftarrow t$

UPON RECEIVE Y:$(y, source, target, t)$ **do**
  $activeChildren \leftarrow activeChildren \cup \left\{i | i \in I^M_{source} \setminus \{\mathtt{map}(id)\}\right\}$
  $\Psi \leftarrow \bigcup_{i \in activeChildren} Z^M_{i,source}(y)$
  $\Lambda \leftarrow \Lambda \cup \left\{ Z^M_{i,\mathtt{map}(id)}(y) \,|\, i \in I^M_{\mathtt{map}(id)} \right\}$
  $t_L \leftarrow t$

UPON RECEIVE DONE:$(source, target, t)$ **do**
  $L \leftarrow L \tilde{\cup} \{(source, t)\}$ // replace *source* entry if it already exists in L, otherwise add entry to L
  $activeChildren \leftarrow activeChildren \setminus \{source\}$
  $t_N \leftarrow \min(t_N, t)$

6.3.2 Communication between Simulators

Each simulation entity runs on a machine. This is modelled by a channel from the simulator to the machine being active, determined by the non-zero dimension of *activePorts*. There can be at most one active channel per simulator at a time.

The Local Coupling Table (LCT) holds a table mapping each simulator running on the machine to a unique port. When it receives an event, it is forwarded to the appropriate port of the target after some delay time. The delay time for local search is sampled from a parameterized uniform distribution (typically order of milliseconds). However, if the target is not in the local table, it is forwarded to the send_remote outport. The events received are stored in a queue to handle concurrency (reception of remote and local events). In order to ensure sequential execution of simulators on the same machine, the LCT waits for a call-back from the simulator currently processing before the next event in the queue is sent. Since ASs always send a DONE message after the reception of a simulation message, the LCT expects such a message before sending the next event. As for COs, they only send a DONE message after the reception of a simulation message that induces *activeChildren* to be empty. Therefore the output function of the CO is extended to send a RETURN message after a simulation message is received. An LCT models the intra-machine communication of simulators.

The Remote Coupling Table (RCT) has a similar behaviour to the LCT. Additionally, it holds a table mapping each simulator in the cluster to the machine it is running on. The parameterized delay time is typically longer than for an LCT, taking in consideration network communication delays (typically order of tens to hundreds of milliseconds). However, the event queue does not depend on call-backs. An RCT models inter-machine communication of simulators.

Machines are modelled as coupled models comprising two atomic sub-models: LCT and Activity. The state of an Activity is either *ACTIVE* or *FAILED*. The Activity model generates failures on the machine. After some time (specified in the time advance), it sends a *failure message* to the LCT[1]. When the LCT receives a failure, it is passivated (the time advance evaluates to infinity).

6.3.3 Fault-tolerance Entities

When running a distributed environment, several fault-tolerance issues must be handled. Among them is machine failure. As a consequence, mechanisms such as state restoration and resource reallocation come into play. There are three major components modelled as atomic DEVS models that ensure fault detection correct restoration: the Monitor, the Log, and the Master servers.

The Monitor server monitors each machine to detect failures. At regular time intervals, it pings all the machines through its ping outport. The Activity then receives this request from activity_req. After some small delay, it sends back an acknowledgement via its notify outport. Note that if the state of the Activity is *FAILED*, then no acknowledgement will be output. The monitor accumulates all responses within a certain timeout. It continues pinging (at the regular frequency) as long as it receives responses from all the machines (from its alive inport). However, if timeout is reached beforehand, the Monitor considers the remaining machines that have not responded yet as failed. It subsequently notifies the Master model.

The Log server receives the log messages from the AS, CO, and RC entities through its log_in inport. The log message of a simulator, identified by *id*, is LOG: (id, m, s), where m is the last simulation message received and s is the resulting state after δ_{ext} or δ_{int} is applied. At the level of the Log

[1] It is possible to model machine replacement by allowing the Activity to send a *revival message*.

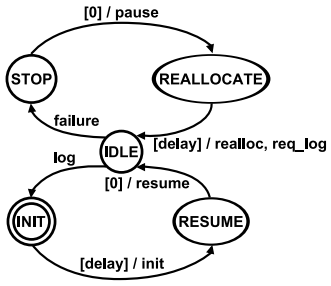


Figure 6.5: The modal behaviour of the Master

model, a cleaning mechanism removes unnecessary traces from the log. Whenever a third entry from the same simulator is received, the external transition function of Log removes the first one. This is sufficient since state restoration is only applied from the latest previous state.

The Master server coordinates the whole environment. Its state holds the simulation hierarchy (coupling of all the AS, CO, and RC models) and the resource allocation (which simulator is currently running on which machine). Before the simulation starts, the master sends an INIT message to all the simulators. Recall that this message provides knowledge of the parent of the simulator, the machine it will be running on, its children (in case of a CO or a RC), and the initial simulation time (for the RC). The Statechart in Figure 6.5 expresses the behaviour of the Master. After initialization, the Master sends the control message $\chi =$ RESUME. Subsequently, the RC sends a * message to its children and the simulation runs as long as the termination condition is satisfied. When it receives a failure notification (from Monitor), the Master first sends $\chi =$ PAUSE to all the simulators to halt the simulation. It also requests (from Log) for the last saved state of the simulators formerly running on the failed machines. We call a simulator (AS, CO, or RC) to be failed if it is allocated to a failed machine (*i.e.,* the Activity is in *FAILED* state). In the mean time, the Master repartitions the simulators. The output function sends the appropriate ρ message to the failed simulators. Note that the repartition may also need to reallocate simulators that were running on non-failed machines. It notifies the RCT and as well as each machine about the new allocation of resources. Upon receiving the log entries from Log, the Master then sends an INIT message to the failed simulators in order to restore their state to the previous "safe" state. Finally, it sends $\chi =$ RESUME to all the simulators to continue the simulation from their current (or newly modified) state.

We have modelled the Master, Log, and Monitor components as three different servers. It is possible to consider them as one single server. Note that the Master could even be modularly split further. This is an implementation design consideration.

6.3.4 Generic Instantiation and Parametrization

This model of a distributed DEVS simulator was implemented in pythonDEVS. To be able to simulate this model, several simulation experiments are provided as a library. It instantiates the Cluster coupled DEVS model which, in turn, creates the necessary ASs and COs according to the given host DEVS model. The necessary inports, outports, and channels are created. For experimental purposes, the

Cluster system expects several parameters.

- The initial host model M;
- the number of machines in the cluster n;
- the initial partition of resources specifying the node location of every simulation entity;
- the initial states of all the AS, CO, and RC models, referred to by s_0 (the start time of the simulation can be specified in the initial state of RC);
- the termination condition θ of the simulation, specific to M;
- the distribution of the delay Δ_{LCT} for LCT to respond;
- the distribution of the delay Δ_{RCT} for RCT to respond;
- the distribution of the delay Δ_{LOG} for Master-Log communication (typically very fast);
- the distribution of the delay Δ_{MON} for Master-Monitor communication (typically very fast);
- the ping frequency p of the Monitor; and
- the distribution of the delay Δ_{ACT} for the Activity to notify that machine has not failed (typically $\Delta_{ACT} \approx \Delta_{LCT}$).

We propose an experiment where the only variable is the time before the Monitor times-out. The simulation collects performance results for different timeout values. Performance can be measured, for example, by the number of log entries in the Log server since every simulation operation of AS, CO, and RC is logged.

The termination condition for the simulation experiments of the modelled DEVS simulator is satisfied when either θ is satisfied at the RC level or if all Activity models are in *FAILED* state.

6.4 A Distributed DEVS Simulator

We have chosen the RMI (Remote Method Invocation) as a middleware layer for our implementation for many reasons. For our purposes of implementing a sequential classical DEVS simulation protocol, RMI simplifies distributed computing, through the transparency it provides over remote procedure calls. It also hides all internal implementations of the lower level communication to maintain remote references locally. Given the nature of our particular simulations, any type of object can be sent between the solver objects as an event. This flexibility in object types would be much more tedious to implement using TCP. In the following, we briefly describe the concrete realization of the distributed simulator. We use “Python Remote Objects” (PyRO), a RMI-based middleware solution similar to Java RMI. PyRO is implemented in Python, which makes it compatible with our `pythonDEVS` sequential DEVS simulator implementation of which we re-use parts.

6.4.1 Location Configuration and Model Partitioning

Partitioning is often referred to as the deployment of different solver objects onto different processes running on different servers. Many algorithms allow for optimal partitioning whenever dynamic reconfiguration is necessary, but this is not the scope of this work. Nevertheless, such an algorithm

is easily pluggable into our implementation. In Section 6.5.1, different partitioning possibilities of the input DEVS model will be simulated. The optimal partitioning of solver objects over the cluster machines can then be obtained from simulation experiment results.

6.4.2 Simulation Tracing - Log Server

As described previously, the DEVS simulation is meant to produce a trace which describes events occurring at certain times, as well as the way they are handled. This trace can be produced from all solver objects executing on different machines. To aggregate these individual traces into a single simulation trace a *Log server* is used. The Log server also captures the communication trace between solvers. It is used to calculate the network delay variables (LCT, RCT) by logging the time the messages were sent and received. The communication trace allows for estimation of the network and the framework overheads. This log server also stores the most recent states of the solvers for fault tolerance purposes.

6.4.3 Instantiation

To run the simulation, the initial step is to start a naming server on one of the machines on the cluster. Then, a server containing a factory object (solver factory) which can host solver objects is instantiated on each of the cluster machines. It will subsequently register itself with the naming server. Through the naming server, the solver factory can be discovered by the simulation engine clients. These clients instantiate solver objects, atomic or coupled, destined to live on the factory object's machine.

A remote reference is created for each factory and is used to create the solver objects. These solver objects are passed the log server reference to send their traces. The simulator has been implemented to accept a mapping object, describing which machine in the cluster a solver should be running on. The simulator can then instantiate the solvers according to the partitioning mapping locations. If no specific mapping was provided, they are instantiated locally. Figure 6.6 illustrates the overall architecture of the implementation.

6.4.4 Simulation protocol

The simulation protocol was implemented in a asynchronous fashion. This allows for better performance of the overall simulation. For example, when an event is destined to multiple solvers, it is broadcast in an asynchronous fashion. In turn, the receiving solvers can receive and process the event simultaneously, and then through a callback respond asynchronously.

6.4.5 Fault Tolerance Implementation

Since solvers are dispersed over several machines, it is expected to encounter a new class of errors which were not present in the classical non-distributed DEVS simulator. Like in any other distributed settings, we would have to deal with unexpected machine crashes. It is important to handle such crashes with affecting minimally the simulation flow. This becomes more critical for long running simulations. In the case of a crash, the simulation will stop and would have to be manually restarted later on. This is deemed inconvenient and not scalable as we expect the possibility of machines crash-

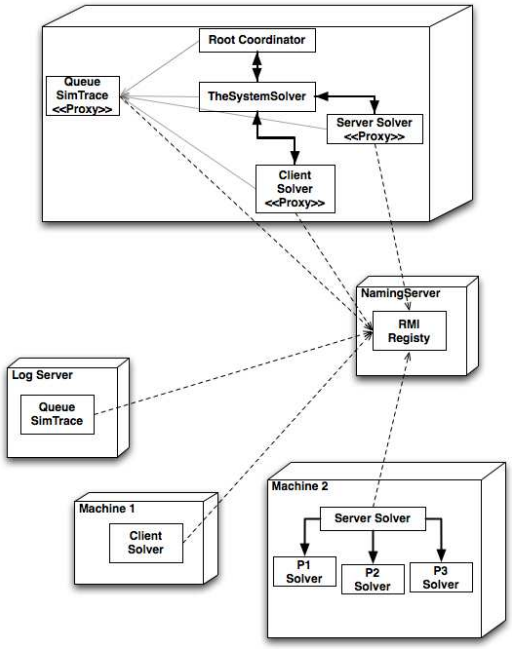


Figure 6.6: The RMI architecture of a distributed DEVS simulator using PyRO

ing increasing with running time. As in any fault-tolerant technique, there are two phases for achieving that: fault detection and fault recovery. For fault detection, there exists many techniques, e.g., acceptance tests or *n*-version programming. For recovery, first perform regular backup operations whenever a modification to the state of the system (or object) takes place. Second, we could restore the latest fault-free state.

In the application of the distributed DEVS simulation, we consider a fault model where cluster machines involved in the simulation can potentially crash. To detect failures, a *timeout* is set on each methods being executed on remote objects. Therefore if a solver or the machine it lives on failed to respond to a call (simulation call, heartbeat call, or other calls) after a certain timeout, the site or the component on the site is considered as crashed. Then, the recovery mechanism takes over. Ideally these timeouts should be specified according to the partitioning or mapping of the simulation.

Recovering from such a fault requires collecting enough data about the state of each solver and they simulate. For Atomic Solvers: t_L, t_N, and the state of the DEVS model being simulated. For Coordinators: t_L, t_N, *eventList*, *subsolvers* and the state of the DEVS model being simulated. The solver objects (coupled or atomic) which simulate the sub-DEVS component of the current solver's DEVS model.

Having the last valid version of this information for all the solvers is sufficient to restore the simulation at a correct state to resume the simulation from where it left off. The choice of when to update this version is one of the key optimization issues. For our application we have chosen to "piggy-back" this information in a conservative fashion from each solver to the log server whenever the solver state gets updated.

6.5 Calibration and Optimization

The simulation experiments we perform use an input model *M*. This is an abstracted model of a city. A city (coupled DEVS) is divided into several districts (coupled DEVS). A district encapsulates a road network with houses and offices (both atomic DEVS). Roads (coupled DEVS) are bidirectional and thus consist of two road segments (atomic DEVS). In the road network of a district, roads can connect to an intersection (coupled DEVS composed of eight road segments) at specific points, with or without a traffic light (atomic DEVS). Some districts communicate via highways (coupled DEVS) modelled as sequences of roads. At periodic intervals, houses generate cars to go to a predefined office. Note that a house and its corresponding office can be in different districts. After some random time interval, a car leaves the office and returns back home. The simulation ends when all cars are back home or are involved in a car accident.

The simulator of the DEVS model *M* is itself modelled as a DEVS model as described in Section 6.3. The simulator model was calibrated with model execution parameters from the PyRO-based distributed simulator described previously.

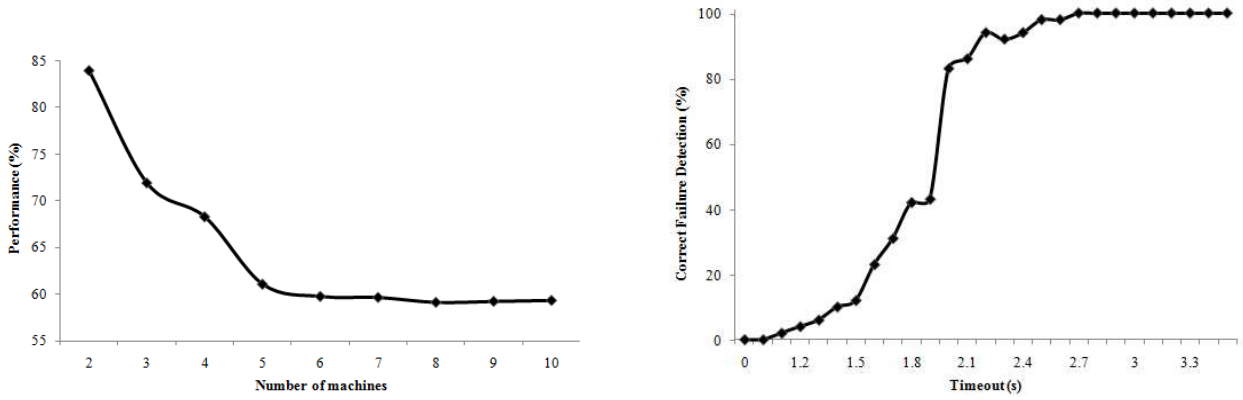
6.5.1 Optimization of Performance Metrics

One of the most important configuration questions in the distributed implementation, with regards to efficient fault-tolerance, is the timeout to set on solver calls, before assuming that the site has crashed or even has a fault. This timeout really depends on the model being simulated and the partitioning that is used for the solvers. The timeout value might also be different at different solver levels. Setting an arbitrary constant value may not be sufficient, as it has to take into account the network message passing time and other properties.

To accomplish a realistic simulation, several parameters need to be calculated from the middleware and the cluster. Network statistics for a specific cluster can be gathered using the current implementation of the log server. As discussed, the log server keeps track of both the simulation and communication traces. Communication trace analysis allows one to estimate remote and local delays for message passing between different solvers. Each solver outputs a communication trace before sending a message to another solver and when it receives one. Traces are appended with extra information to allow calculating latency and are classified as local or remote messages. For example, the trace message produced at each solver is augmented with the following parameters:

- the name of the model in the solver,
- the global current time at which this trace was produced,
- the local time at the solver machine when the message was received,
- the local time at the machine when the trace was produced.

After several experiments on the cluster simulating the city example, we analysed the trace information to calculate the distribution of remote and local delays. These values were then used as parameters in the modelled simulator to simulate message passing delays.



(a) (b)

Figure 6.7: (a) Effect of adding machines and (b) delay before master reaction on correct detection of machine failure.

By running a simple simulation, the log server can produce some average network parameters with regards to the time a message takes to travel over the network from a solver to another in the hierarchy. These numbers can be given to the modelled distributed simulator described earlier. It then can simulate this delay which is specific to the cluster.

The optimal timeout for a specific solver message on another can be discovered using the modelled simulator: before a solver makes a call on another solver, it can invoke the modelled simulator to run a simulation of the current call and expect the time length it should take. This simulation is expected to be much faster than the actual implementation since it will only simulate and not run the actual model. The computed value is then used as a timeout limit for the call.

Figure 6.7(a) illustrates the performance behaviour of the simulator depending on the number of machines used. In our experiments the city model has five districts and 10 bridges, totalling 1000 coupled DEVS models and 10,000 atomic DEVS models. For our example, the partitioning of the city models on the different available machines follows one constraint: the coupled DEVS representing a district and all its sub-models are always on the same machine. Performance was measured by taking the total simulation time of the simulation run. The graph shows a decrease in performance when the number of machines involved increases. This is because the network communication between components adds an overhead. The graph hence shows that, for our model, the optimal number of machines is two. (The case where there is only one machine is ignored since it is not distributed). Because our goal is not to increase performance but rather maximize interoperability using specialized nodes. For example, in some cases parts of the partition would be fixed. Then such analysis becomes more valuable. Furthermore, the performance levels off when more than six machines are used. This is due to the limited number of components in the modelled simulator.

The graph in Figure 6.7(b) allows one to find the optimal timeout for the given configuration. In this case the minimal timeout with 100% reliable fault detection is 2.7 seconds. This is significantly less than the monitor frequency (which is 7 seconds).

6.6 Related Work

The DEVS formalism and its simulation protocol as described in Section 6.1 are well suited for local execution on a single machine. In addition, many approaches for distributing DEVS simulation have been proposed and implemented (*e.g.,* [SPB+04, CSPZ04, ZZH06, SKHP07]). Thanks to the modularity and hierarchical structure of DEVS, distributing a DEVS model execution on several processors can be achieved without modifying the simulation models.

The distributed architecture for DEVS can be divided into layers. The *application layer* (highest level of the system) is the modelling and simulation problem under study. The DEVS model lies in the *modelling layer*. At the *simulation layer*, the protocol to simulate the DEVS model is implemented. The lowest level layer is the *middleware layer* where the communication between computing nodes is implemented.

James II [URH03] is a dynamic simulation framework where a model of a DEVS simulator can be simulated. However, the purpose is to enhance the *DEVS* formalism with dynamic restructuring capabilities. That is, the simulation model (composed of atomic solvers and coordinators) is modified at run-time to, for example, re-partition the model distributed on several machines.

6.7 Conclusion

In this chapter, we have explicitly modelled both the structure and the behaviour of a distributed DEVS simulator, as a DEVS model. From this DEVS model, a distributed DEVS simulator was realized (*i.e.*, partially synthesized). This simulator runs over RMI using *PyRO*. The actual performance data obtained from this implementation (simulating the model of traffic in a synthetic city) was used to obtain realistic parameter values to be used in the DEVS model of the simulator. Isolating system variables, such as timeouts, optimal values were found by simulating multiple alternative models of the DEVS simulator. These variables were finally used to calibrate the real simulator.

In the future, we want to completely automate the synthesis of simulators from their DEVS models. Also, we plan to synthesize DEVS/RMI instead of the rather inefficient PyRO.

7 MoTif

It is possible to design transformation languages that are entirely explicitly modelled. We will now use the framework presented in Part II to engineer a new model transformation language. This language is completely modelled: its syntax, its semantics, and its execution engine.

In Chapter 3, *T-Core* was presented as the result of a de-construction process of model transformation languages to a set of most primitive constructs. In this chapter, we propose to combine all *T-Core* primitives with the modelling and simulation formalism modelled in Chapter 6 to design a new model transformation language. The choice of the underlying formalism allows one to easily add the dimension of time and asynchrony to model transformation. A nice side-effect of explicitly modelling the transformation language is the facility to design higher-order transformations.

7.1 Introduction

In 1996, Blostein et.al. [BFG96] described some issues regarding the practical use of graph rewriting, at that time very sporadic. Graphs are a versatile and expressive data representation, and there are many advantages to the explicit representation (as opposed to encoding in the form of programs) of graph transformations. Issues such as expressiveness, scalability and re-use of models of graph transformation as well as the ability to integrate such models with traditional software components were considered critical enablers for wide-spread use of graph transformations. During the last decade, several of these issues have been addressed and tools have been developed. In particular, tools such as *GReAT* [AKK+06], *FUJABA* [NNZ00], and *ProGReS* [SWZ95] (just to name a few) allow for *programmed graph rewriting*. The purpose of programmed graph rewriting is to be able to model the control structure of (graph) transformation. This is done in terms of control flow primitives[1] such as *sequence*, *branching* (choice), and *looping* (iteration). *Hierarchical encapsulation* allows for *modular construction* (and re-use) of control flow structures. Some tools add expressiveness through *non-determinism* and *parallel composition*. In general, it is also desirable for a control language to be target (programming) language *neutral*. The explicit incorporation of *time* is rare in current transformation languages. Programmed (or controlled) graph transformation is one of the keys to making graph transformation scalable and hence industrially applicable. Most current graph transformation tools

[1] These requirements were summarized in Section 2.1.3.

support programmed graph transformation, but mostly introduce their own control flow language[2].

This chapter presents a novel model transformation language. The main contribution of this chapter is the re-use of a discrete-event modelling and simulation formalism, such as the *DEVS* (c.f., Chapter 6), to describe the scheduler of a model transformation. Since *DEVS* inherently allows one to build hierarchical models, the transformation language becomes highly modular by re-using specific components of a transformation. Another side-effect of using *DEVS* is the explicit introduction of the notion of time in model transformations. This allows one to model a time-advance for every rule as well as to interrupt (pre-empt) rule execution.

In Section 7.2, we formally define the *MoTif-Core* model transformation language, as well as its semantics, based on the *DEVS* formalism. Section 7.3 discusses some properties of the language. Then Section 7.4 describes a model transformation language *MoTif* that is more convenient to use at the transformation modelling level. It encapsulates model transformation features defined with *MoTif-Core* building blocks. Section 7.5 explains how a *MoTif* transformation is executed. As a side effect of explicitly modelling all aspects of *MoTif*, Section 7.6 illustrates how to express higher-order transformations in this language. Finally, Section 7.7 explores related work.

7.2 Semantic Mapping Onto DEVS

In Chapter 3, we have shown how model transformation languages can be de-constructed in a collection of model transformation primitives, which makes it easier to reason about transformation languages. These primitives are encapsulated in the *T-Core* module. By properly combining *T-Core* primitives with existing well-formed programming or modelling languages allowed us to re-construct some already existing transformation languages and even construct new ones. Recall Figure 3.5 which showed a combination involving the whole *T-Core* module with the *DEVS* formalism. In the context of rule-based graph transformation, the *DEVS* formalism can be used as an underlying basis for rule scheduling in transformation languages [SV07]. *DEVS* is a compositional, timed discrete-event language and is thus an attractive framework for a general purpose model transformation language. The combination of *T-Core* with *DEVS* is a transformation language called *MoTif-Core*.

The meta-model of *MoTif-Core*, described by the UML class diagram in Figure 7.1, shows how *MoTif-Core* is an extension of *DEVS* integrating all[3] *T-Core* constructs. The so-called *T-Core* primitives can be found encapsulated in the state of different atomic DEVS AtomicPrimitive elements. Those classes are prefixed by "TC", depicting that they are semantically identical to their *T-Core* counterparts (*e.g.*, in *MoTif-Core*, TCMatcher represents the Matcher from *T-Core*). However, the messages sent through their method calls use Event objects from *MoTif-Core* instead of *T-Core* Messages.

All the RulePrimitive elements have two inports (PacketIn and CancelIn) from which packets and cancel events are respectively received and two outports (SuccessOut and ExceptionOut) from which packets and exception events are sent. The Matcher, Iterator and Rollbacker have an additional FailOut outport from which packets are sent. The latter two atomic blocks have an extra NextIn inport from

[2]Though *FUJABA*'s Story Diagrams are heavily based on UML Activity Diagrams.

[3]Except for the Composer since the composition of primitives is specific to the language combined with *T-Core*.

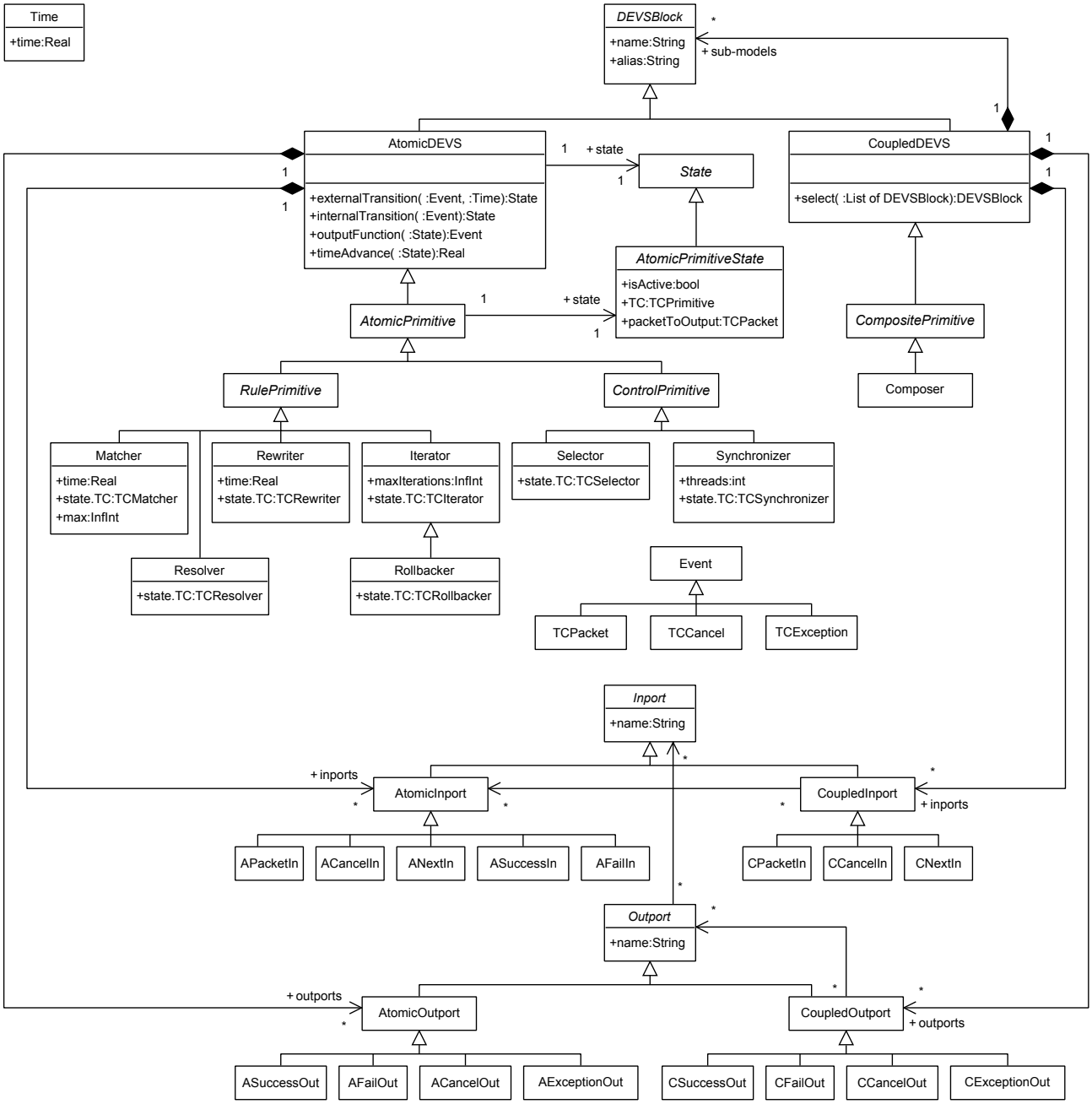


Figure 7.1: The meta-model of *MoTif-Core*.

which packets can be received. For compositionality and transparency reasons, the Composer has all of the ports mentioned. As for ControlPrimitive elements, they have the SuccessIn and FailIn inports, and SuccessOut and FailOut outports to receive and send packets. The Selector has an additional CancelOut ouport from which cancel events are transmitted. When a *MoTif-Core* primitive (any sub-class of RulePrimitive, ControlPrimitive, or CompositePrimitive) receives an event from an inport, its external transition function invokes the appropriate method of its corresponding *T-Core* primitive according to the activated inport. Algorithms 19 and 20 depict this integration for RulePrimitves. In the ControlPrimitive elements, the Selector's `select` method is no longer used. It is the select function of the Composer that takes care of the selection of the appropriate primitive to output. In *MoTif-Core*, the Composer inherently composes other DEVSBlock elements (*MoTif-Core* primitives as well as pure DEVSBlocks) as it is a CoupledDEVS and hence stateless. CompositePrimitives specify the connection between the different in/outports to ensure a proper flow of the transformation.

Algorithm 19 rulePrimitive.`extTrans`$((s,e),x)$

```
if x received from ACancelIn then
    s.state.cancelIn(x)
else if x received from APacketIn then
    s.packet = s.state.packetIn(x)
else if x received from ANextIn then
    // If defined
    s.packet = s.state.nextIn(x)
end if
```

Algorithm 20 rulePrimitive.`output`(s)

```
if s.state.isSuccess then
    output(s.packet, ASuccessOut)
else if s.state.exception ≠ nil then
    E = toDEVSEvent(s.state.exception)
    output(E, AExceptionOut)
else
    output(s.packet, AFailOut)
end if
```

In the following, we formally define the semantic mapping of *MoTif-Core* onto the *DEVS* formalism. The behaviour of every *T-Core* element was precisely described in Chapter 3. Note that the time base used is $T = \mathbb{R}^+ \cup \{+\infty\}$. Also, for the sake of completeness of the formal DEVS models, we assume an input segment function[4] $\omega : T \to X$ determining the input event on an inport at a certain time. At the end of this section, an example illustrates the use of these constructs to build graph transformation rules.

7.2.1 The different events

There are exactly three types of events. In our notation, E denotes any event instance of a sub-class of Event. We write $E \in$ `INSTANCESOF`(Event)[5]. Packets, cancel, and exception events correspond to the *T-Core* messages defined in Section 3.2.

A packet is a structure $\pi = (\gamma \in G^*, \texttt{current}, \{MS_i | i \in \mathbb{N}\})$, where γ is a graph taken from the set G of all directed, attributed, typed graphs[6]. `current` is an instance of PreConditionPattern referring to the currently processed match set. MS is an instance of MatchSet. Furthermore, $MS = \left(\texttt{condition}, j \in \mathbb{N}, \left\{m_j | j \in \mathbb{N}\right\}\right)$, where `condition` is an instance of PreConditionPattern and m

[4] Recall that the input segment function triggers the external transition function.

[5] The function `INSTANCESOF`(X) returns the set of all instances of class X and its sub-classes.

[6] $G^* = G \cup \{\texttt{nil}\}$ and $V_\gamma^* = V_\gamma \cup \{\texttt{nil}\}$

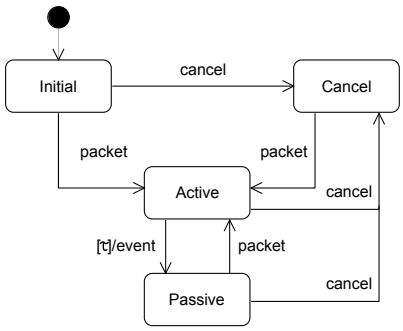


Figure 7.2: The behaviour of AtomicPrimitives.

is an instance of Match. A match set is identified by a pre-condition pattern `condition` which is an instance of PreConditionPattern. It also holds all the matches of the pattern $m \in$ `INSTANCES-OF`(Match). A match $m = (h_m : \gamma \to \gamma, \{\rho_k | k \in \mathbb{N}\})$ consists of a mapping and pivot assignments. h_m is a homomorphism mapping the nodes of a pattern graph to nodes of a source graph. A pivot ρ is an instance of Pivot and is defined by a string `label` and a single-node graph $h_\rho(V_\gamma)$ where $h_\rho : V_\gamma \to \gamma$ is a morphism.

A cancel event φ instance of TCCancel carries an `exclusions` set of PreConditionPatterns, depicting the RulePrimitive elements whose activity should not be cancelled. Finally, an exception event χ instance of TCException can also be transmitted by *MoTif-Core* primitives. The detail of exception events and their handling will be detailed in Chapter 8 and will not be covered in this chapter.

7.2.2 The AtomicPrimitives

The AtomicPrimitives are atomic DEVS models. A *name* identifies the model and optionally an *alias* can be used in the occurrence of multiple models of the same type having the same *name*. The combination $(name, alias)$ should be unique among each AtomicPrimitive type. Furthermore, an AtomicPrimitive instance M is globally uniquely identified by the function $id(M) = (\mathit{TYPE}(M), M.name, M.alias)$, where $\mathit{TYPE}(M)$ gives the exact type of M.

The general behaviour of the state of an AtomicPrimitive is defined by the statechart in Figure 7.2. It initially starts in the `Initial` state. At any point in time, whenever a packet is received, the AtomicPrimitive is in its `Active` state. After some time (defined by its time advance function τ), it outputs an event which can be a packet, an exception, or a cancel event. This then brings it to the `Passive` state. Also at any point in time, whenever a cancel event is received, the AtomicPrimitive is in a `Cancel` state.

The Matcher

The Matcher is an atomic DEVS, parametrized by a pre-condition pattern c, the time Δ it will consume, and the maximum number of matches *max* (to optimize the search of the matching process). A Matcher

is defined by the following structure:

$$Matcher_{name,alias,c,\Delta,max} = \langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

The state S is defined as:

$$S = \{(\alpha, TC, \pi) \,|\, \alpha \in \mathbb{B}, TC \in \texttt{INSTANCESOF}(\text{TCMatcher}), \pi \in \texttt{INSTANCESOF}(\text{Packet}) \cup \{\texttt{nil}\}\}$$

In this notation, the boolean[7] variable α indicates whether the Matcher is active, *i.e.*, processing a packet. TCMatcher corresponds to the matcher class in *T-Core*. Recall that TCMatcher finds *max* possible matches of c on the (graph) model embedded in a packet. The Matcher temporarily holds a packet, from the time it receives it to the time it outputs it, as no *T-Core* rule primitive stores any packet in its state. Note that all the parameters are also part of S, but are omitted in this notation for simplicity. We denote by $s_0 = (\texttt{false}, \text{TCMatcher}(max), \texttt{nil})$ the initial value of S.

The APacketIn port can receive a packet instance. We therefore define:

$$X_{\texttt{APacketIn}} = \{\texttt{INSTANCESOF}(\text{Packet})\} \cup \{\phi\}$$

ϕ represents the null event as used in [Zei84]: it covers the case when no event is present. The ACancelIn port can only receive $\varphi \in \texttt{INSTANCESOF}(\text{Cancel})$, indicating to the matcher to cancel its activity. Thus:

$$X_{\texttt{ACancelIn}} = \{\texttt{INSTANCESOF}(\text{Cancel})\} \cup \{\phi\}.$$

The Matcher can receive either a packet or a cancel event or both. Hence, the input set of the Matcher is the cross-product:

$$X = X_{\texttt{APacketIn}} \times X_{\texttt{ACancelIn}}$$

The ASuccessOut and the AFailOut ports can both send packet instances. Thus:

$$Y_{\texttt{ASuccessOut}} = Y_{\texttt{AFailOut}} = \{\texttt{INSTANCESOF}(\text{Packet})\} \cup \{\phi\}$$

Furthermore, the Matcher can output an exception, hence:

$$Y_{\texttt{AExceptionOut}} = \texttt{INSTANCESOF}(\text{Exception}) \cup \{\phi\}$$

The output set of the Matcher is therefore:

$$Y = Y_{\texttt{ASuccessOut}} \times Y_{\texttt{AFailOut}} \times Y_{\texttt{AExceptionOut}}$$

The time advance is finite only when the Matcher is active. The funtion $\Delta : \texttt{INSTANCESOF}(\text{Packet}) \to \mathbb{R}^+$ specifies the matching time, which may depend on the current packet π. Note that whenever $\alpha = \texttt{true}$ then $\pi \neq \texttt{nil}$, hence Δ is well-defined. The time advance function of the Matcher is thus defined as:

$$\tau(\alpha, TC, \pi) = \begin{cases} \Delta(\pi) & \text{if } \alpha = \texttt{true} \\ +\infty & \text{otherwise} \end{cases}, \forall s \in S$$

[7] $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ represents the set of boolean values.

The internal transition function resets this atomic DEVS to its initial state. It will thus passivate the Matcher as it will force the time advance to evaluate to infinity:

$$\delta_{int}(s) = s_0$$

The external transition function is constructed as follows:

$$\delta_{ext}(((\alpha, TC, \pi), e), x) = \begin{cases} (\texttt{false}, TC^c, \texttt{nil}) & \text{if } x = (E^*, \varphi) \wedge c \notin \varphi.\texttt{exclusion} \\ (\texttt{true}, TC^p, TC.\texttt{packetIn}(\pi')) & \text{if } x = (\pi', \phi) \vee (x = (\pi', \varphi) \wedge c \in \varphi.\texttt{exclusion}) \end{cases}$$

When φ is received (from ACancelIn), the Matcher is deactivated and the state is cleared (TC^c is the resulting TCMatcher after $TC.\texttt{cancelIn}(\varphi)$ is applied). E^* denotes any event, including ϕ. When a packet π' is received (from APacketIn), the resulting packet of the `packetIn` operation is temporarily saved in the state. Note that the state of TC may have changed (*e.g.*, the `isSuccess` attribute) and results in TC^p.

Finally, the output function is:

$$\lambda((\alpha, TC, \pi)) = \begin{cases} (\phi, \phi, TC.\texttt{exception}) & \text{if } TC.\texttt{exception} \neq \texttt{nil} \\ (\pi, \phi, \phi) & \text{if } TC.\texttt{isSuccess} = \texttt{true} \wedge TC.\texttt{exception} = \texttt{nil} \\ (\phi, \pi, \phi) & \text{otherwise} \end{cases}$$

Implicitly, it returns the transformed packet if the match was successful, otherwise it returns the original packet. However, if an exception occurred, the Matcher will instead output the exception properly converted to a DEVS event. Note that π is never `nil` since, by construction, it is only applied a finite amount of time, set by Δ, after δ_{ext} is applied.

The Rewriter

The Rewriter is an atomic DEVS, parametrized by a post-condition pattern c and the time it will consume Δ. A Rewriter is defined by the following structure:

$$Rewriter_{name,alias,c,\Delta} = \langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

The Rewriter is structurally very similar to the Matcher, with the difference that the TC element of its state S is an instance of TCRewriter instead. Recall that TCRewriter applies the required transformation for c on the match specified in a packet. Thus the initial state is $s_0 = (\texttt{false}, \text{TCRewriter}(), \texttt{nil})$. Moreover, τ, X, δ_{int}, and δ_{ext} are all identical to those of the Matcher.

The output set of a Rewriter is $Y = Y_{\texttt{ASuccessOut}} \times Y_{\texttt{AExceptionOut}}$, each defined as previously. The output function must be modified accordingly:

$$\lambda((\alpha, TC, \pi)) = \begin{cases} (\phi, TC.\texttt{exception}) & \text{if } TC.\texttt{exception} \neq \texttt{nil} \\ (\pi, \phi) & \text{otherwise} \end{cases}$$

The Resolver

The Resolver is an atomic DEVS defined by the following structure:

$$Resolver_{name,alias,res} = \langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

S is defined as before, but the TC element is an instance of TCResolver instead. Recall that TCResolver resolves potential conflicts between matchings and rewritings by means of a user-defined *res* resolution function, or a default one. The resolution function is defined as $res : \texttt{INSTANCESOF}(\text{Packet}) \rightarrow \mathbb{B}$. Thus the initial state is $s_0 = (\texttt{false}, \text{TCResolver}(res), \texttt{nil})$. X, Y, δ_{int} and λ remain identical to those of the Rewriter.

However, since the Resolver is not parametrized by a condition, the external transition function must be adapted when a cancel event is received: it becomes inactive regardless of the exclusion list.

$$\delta_{ext}\left(((\alpha, TC, \pi), e), x\right) = \begin{cases} (\texttt{false}, TC^c, \texttt{nil}) & \text{if } x = (E^*, \varphi) \\ (\texttt{true}, TC^p, TC.\texttt{packetIn}(\pi')) & \text{if } x = (\pi', \phi) \end{cases}$$

Also, the Resolver does not consume time, therefore its time advance is 0 when it is active:

$$\tau(s) = \begin{cases} 0 & \text{if } \alpha = \texttt{true} \\ +\infty & \text{otherwise} \end{cases}, \forall s \in S$$

The Iterator and the Rollbacker

The Iterator is an atomic DEVS, parametrized by a maximum number of iterations *max*. It is defined by the following structure:

$$Iterator_{name,alias,max} = \langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

The Iterator is structurally very similar to the Matcher but, in this case, TC is an instance of the iterator class in *T-Core*. Recall that TCIterator chooses a match among the set of matches of the `current` condition of a packet. Thus the initial state is $s_0 = (\texttt{false}, \text{TCIterator}(max), \texttt{nil})$.

Y and λ are identical to those of the Matcher. τ is defined as in the Resolver depicting that the Iterator does not consume time. For the input set, the Iterator can receive either a packet from APacketIn or ANextIn port as well as a cancel event from ACancelIn. In this case $X_{\texttt{ANextIn}} = X_{\texttt{APacketIn}}$. Therefore the input set is

$$X = X_{\texttt{APacketIn}} \times X_{\texttt{ACancelIn}} \times X_{\texttt{ANextIn}}$$

Consequently, the external transition function is modified to handle the packet received from ANextIn:

$$\delta_{ext}\left(((\alpha, TC, \pi), e), x\right) = \begin{cases} (\texttt{false}, TC^c, \texttt{nil}) & \text{if } x = (E^*, E^*, \varphi) \\ (\texttt{true}, TC^p, TC.\texttt{packetIn}(\pi')) & \text{if } x = (\pi', E^*, \phi) \\ (\texttt{true}, TC^n, TC.\texttt{nextIn}(\pi')) & \text{if } x = (\phi, \pi', \phi) \end{cases}$$

When a packet π' is received from ANextIn, the resulting packet of the *nextIn* operation is temporarily saved. TC^n is the state of the TCIterator after $TC.\mathit{nextIn}(\pi')$ is applied. *DEVS* constrains that exactly one atomic DEVS may output at a time. However, a single atomic DEVS may output events that result on different ports (in a one-to-many channel connection). In this case, *APacketIn* and *ANextIn* might simultaneously receive a packet. The second condition of δ_{ext} enforces that the packet received from *APacketIn* is handled and the other one is discarded.

Finally, the internal transition function behaves differently than for the previous models. It does not reset the state of the Iterator to s_0 anymore but keeps the TCIterator state unchanged. That is because when a packet arrives from the ANextIn inport, the counter of the remaining iterations should not be reset. Thus:

$$\delta_{int} = ((\alpha, TC, \pi)) = (\mathit{false}, TC, \mathit{nil})$$

As Figure 7.1 shows, the Rollbacker is derived from the Iterator. Recall that the rollbacker in *T-Core* checkpoints the packets it receives to roll-back to a previous packet when needed (it fulfils the standard back-tracking property in graph transformation [ZS92]). In *MoTif-Core*, the Rollbacker is defined exactly like the Iterator with the difference that the TC element of its state S is an instance of TCRollbacker instead. Also recall that the maximum number of iterations is set implicitly in the *packetIn* method of the Rollbacker. Thus:

$$\mathit{Rollbacker}_{name,alias} = \langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

and the initial state is $s_0 = (\mathit{false}, \text{TCRollbacker}(), \mathit{nil})$. Furthermore, since the *nextIn* method is independent from the event received, then $X_{\mathit{ANextIn}} = \{\mathit{INSTANCESOF}(\text{Event})\} \cup \{\phi\}$.

The Selector

The Selector is an atomic DEVS defined by the following structure:

$$\mathit{Selector}_{name,alias} = \langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

S is now needs to be modified to not explicitly store packets directly in the state. That is because the *T-Core* selector already does in the *success* and *fail* sets. Thus:

$$S = \{(\alpha, TC) \,|\, \alpha \in \mathbb{B}, TC \in \mathit{INSTANCESOF}(\text{TCSelector})\}$$

Recall that TCSelector allows exactly one packet to proceed, being non-deterministically selected. The initial state is then $s_0 = (\mathit{false}, \text{TCSelector}())$. τ and δ_{int} are identical to those of the Resolver. The output set is augmented with the capacity of sending cancel events:

$$Y_{\mathit{ACancelOut}} = \{\mathit{INSTANCESOF}(\text{Cancel})\} \cup \{\phi\}$$

Thus

$$Y = Y_{\mathit{ASuccessOut}} \times Y_{\mathit{AFailOut}} \times Y_{\mathit{ACancelOut}} \times Y_{\mathit{AExceptionOut}}$$

The input set is

$$X = X_{\texttt{ASuccessIn}} \times X_{\texttt{AFailIn}} \times X_{\texttt{CancelIn}}$$

where

$$X_{\texttt{ASuccessIn}} = X_{\texttt{AFailIn}} = \{\texttt{INSTANCESOF}(\text{Packet})\} \cup \{\phi\}$$

The external transition function is given by:

$$\delta_{ext}\left(((\alpha, TC), e), x\right) = \begin{cases} (\texttt{false}, TC^c) & \text{if } x = (E^*, E^*, \varphi) \\ (\texttt{true}, TC^s) & \text{if } x = (\pi, E^*, \phi) \\ (\texttt{true}, TC^f) & \text{if } x = (\phi, \pi, \phi) \end{cases}$$

As before, when a cancel event is received, the *T-Core* component calls its `cancelIn` function. When a packet π is received from ASuccessIn, $TC.\texttt{successIn}(\pi)$ is applied and TCSelector results in TC^s. Finally when a packet π is received from AFailIn, $TC.\texttt{failIn}(\pi)$ is applied and TCSelector results in TC^f. Note that if both packets are received at the same time, the one received from AFailIn is discarded. Recall that every time a packet is received, it is stored in the `success` or `fail` set of TCSelector.

The output function returns the selected packet resulting from the `select` function of the *T-Core* selector and the packet is sent via the appropriate outport. A cancel event φ is also emitted. It is the result of the `cancel` function which excludes the `current` condition of the selected packet, *i.e.*, $\varphi.\texttt{exclusions} = \{\pi.\texttt{current}\}$. This will cause all atomic primitives receiving that event to cancel their activity except for the identified one. However if the `select` function failed, then only an exception is output.

$$\lambda((\alpha, TC)) = \begin{cases} (\phi, \phi, \phi, TC.\texttt{exception}) & \text{if } TC.\texttt{exception} \neq \texttt{nil} \\ (TC.\texttt{select}(), \phi, TC.\texttt{cancel}(), \phi) & \text{if } TC.\texttt{isSuccess} = \texttt{true} \wedge TC.\texttt{exception} = \texttt{nil} \\ (\phi, TC.\texttt{select}(), TC.\texttt{cancel}(), \phi) & \text{if } TC.\texttt{isSuccess} = \texttt{false} \wedge TC.\texttt{exception} = \texttt{nil} \end{cases}$$

The Synchronizer

The Synchronizer is an atomic DEVS, parametrized by the number of threads *threads* to synchronize and a user-defined merge function *mer*. A Synchronizer is defined by the following structure:

$$\textit{Synchronizer}_{name,alias,threads,mer} = \langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

The Synchronizer is structurally identical to the Selector but, in this case, TC is an instance of the synchronizer class in *T-Core*. Recall that TCSynchronizer synchronizes multiple threads of execution by merging the received packets if all threads succeeded. Merging is performed by means of a user-defined *mer* merge function, or a default one and is defined as $mer : 2^{\texttt{INSTANCESOF}(\text{Packet})} \rightarrow \texttt{INSTANCESOF}(\text{Packet})$. Thus the initial state $s_0 = (\texttt{false}, \text{TCSynchronizer}(threads, mer))$.

X, δ_{int}, and δ_{ext} are identical to those of the Selector, but Y is the same as the Matcher's. The time advance returns 0 only when the number of events received reaches the threshold *threads*. In practice, this means that all threads have reached the Synchronizer.

$$\tau(s) = \begin{cases} 0 & \text{if } |TC.\texttt{success}| + |TC.\texttt{fail}| = threads \\ +\infty & \text{otherwise} \end{cases}, \forall s \in S$$

The output function returns the merged packet resulting from the `merge` function of the *T-Core* synchronizer and the packet is sent via the appropriate outport. However if the `merge` function failed, then only an exception is output.

$$\lambda((\alpha, TC)) = \begin{cases} (\phi, \phi, TC.\texttt{exception}) & \text{if } TC.\texttt{exception} \neq \texttt{nil} \\ (TC.\texttt{merge}(), \phi, \phi) & \text{if } TC.\texttt{isSuccess} = \texttt{true} \wedge TC.\texttt{exception} = \texttt{nil} \\ (\phi, TC.\texttt{merge}(), \phi) & \text{if } TC.\texttt{isSuccess} = \texttt{false} \wedge TC.\texttt{exception} = \texttt{nil} \end{cases}$$

7.2.3 The Composer

The Composer is a CompositePrimitive defined exactly like a coupled DEVS.

$$Composer_{name,alias} = \langle X, Y, N, M = \{M_i | i \in N\}, I = \{I_i\}, Z = \{Z_{i,j}\}, \Xi \rangle .$$

The input and output sets are defined as in their atomic counterpart:

$$X = X_{\texttt{CPacketIn}} \times X_{\texttt{CCancelIn}} \times X_{\texttt{CNextIn}}$$

and

$$Y = Y_{\texttt{CSuccessOut}} \times Y_{\texttt{CFailOut}} \times Y_{\texttt{CExceptOut}}$$

N, M, and I describe the inner-topology of the Composer. The $Z_{i,j}$ functions are the identity as in the example of Section 6.2.1:

$$\forall n \in N, i \in I_n, Z_{i,n} : Y_i \to X_n, Z_{C,n} : X \to X_n, Z_{i,C} : Y_i \to Y$$

As for the select function Ξ, it chooses one sub-model of the Composer from the *imminent set*. Recall that the imminent set is the set of sub-models from M which would have an internal transition at the same time. This set is computed at simulation time by the simulator. Ξ is described by the following prioritized algorithmic steps:

1. If a Selector is in the imminent set, choose the Selector: this ensures that the cancel event will be sent before another DEVSBlock is selected.
2. Among all the AtomicPrimitives that have a successful state ($TC.\texttt{isSuccess} = \texttt{true}$), choose one at random: this allows for optimistic execution.

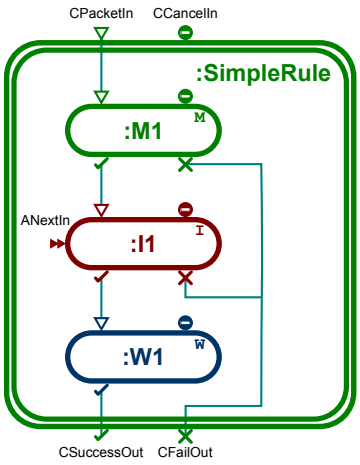


Figure 7.3: A Composer representing a simple transformation rule.

3. At this point no AtomicPrimitive is successful. Now choose any of the AtomicPrimitive or CompositePrimitive models in the imminent set.
4. Finally, the imminent set only contains custom AtomicDEVS. Select a model randomly.

Once a sub-model is selected, it first produces an output (if needed for the current state). This may trigger the δ_{ext} of the influencees of this sub-model. Then its δ_{int} is performed. The select function Ξ is called as long as the imminent set is non-empty. It is important to note that the sequencing resulted from applying Ξ does not conflict with the parallel nature of *MoTif-Core*. This is because all transformation-specific operations (operations on the *T-Core* elements) are performed in the external transition function rather than in the internal transition function. Concurrent internal transition functions are serialized, but external transition functions are executed in parallel (in a parallel setting).

7.2.4 Examples

MoTif-Core is a modelling language for designing model transformation. As defined by Harel and Rhumpe [HR00], a modelling language is defined by an abstract syntax and represented with a concrete syntax; the semantics of the language is defined by a semantic mapping function from the abstract syntax to a semantic domain. The abstract syntax of *MoTif-Core* is defined by the meta-model in Figure 7.1. Its concrete syntax is described graphically by rounded rectangular shapes as illustrated in Figure 7.3. The semantic domain of *MoTif-Core* is the *DEVS* formalism and the semantic mapping function was described in Sections 7.2.2 and 7.2.3, mapping each element from the *T-Core* meta-model onto *DEVS* models. In fact, a *MoTif-Core* model is a *DEVS* model. To illustrate this transformation language, we propose to first re-construct a simple graph transformation rule. Then, we re-construct the case where one transformation rule out of a choice of two is applied in a non-deterministic way.

A simple graph transformation rule

The simplest rule-based model transformation unit is a rule. In graph transformation, a rule behaves as follows. Given a transformation rule pattern, the rule first looks for an occurrence of its LHS pattern to match in the input graph. If a match is found, the rule transforms the graph by rewriting the matched sub-graph, resulting in the RHS pattern. Figure 7.3 shows how a graph transformation rule can be defined in *MoTif-Core*. The graphical syntax of *MoTif-Core* primitives is a rounded rectangle labelled on the top right by its type (M for Matcher, I for Iterator, W for Rewriter, R for Resolver, B for Rollbacker, S for Selector, and Y for Synchronizer). Inside it, the optional `alias` followed by a column and then a *name* identify the primitive. A Composer is represented by a double-lined rounded rectangle. A line depicts a channel connecting ports. In this figure, the composition of the different primitive *MoTif-Core* elements are encapsulated in a Composer named SimpleRule.

The Composer has three inner models: a Matcher, an Iterator, and a Rewriter. M is the set of these three inner models. N is the set of labels which identifies each component m by its unique identifier $id(m)$. The connection topology is given by the influencee sets: $I_{\text{SimpleRule}} = \{\text{:M1:M}, \text{:I1:I}, \text{:W1:W}\}$, $I_{\text{:M1:M}} = \{\text{:I1:I}, \text{SimpleRule}\}$, $I_{\text{:I1:I}} = \{\text{:W1:W}, \text{SimpleRule}\}$, and $I_{\text{:W1:W}} = \{\text{SimpleRule}\}$. For clarity of the figure, the channels from the CCancelIn inport to the ACacenlIn inport of each sub-model were omitted. This is why the influencee set of SimpleRule includes I1 and W1.

The behaviour goes as follows. The packet the SimpleRule receives via its CPacketIn inport is first sent to the Matcher. When the Matcher receives the packet, any[8] occurrence in the graph of its pre-condition pattern c (the LHS) is stored in the packet. After a certain delay specified by Δ, if a match is found, the Iterator receives the modified packet output from the Matcher and selects one match (in this case the only one). Then, the Rewriter receives the packet output from the Iterator and transforms the graph according to its post-condition pattern c (the RHS) applied on the selected match (specified in the packet). After a certain delay specified by Δ, the selected match is removed form the packet and the resulting packet is sent to an outport of the Composer. In the case of a successful application, the newly modified packet is sent through the success outport CSuccessOut. If the Matcher was unable to find any matches, or if the Iterator has exceeded the number of iterations (to select a match), the packet is sent through the fail port CFailOut, depicting that the SimpleRule was not applied.

Non-deterministic selection

MoTif-Core is not only a timed language, but also allows parallel composition of models. Since the different primitive models are designed such that they wait for an input event to arrive at an inport, they can inherently run in parallel inside a Composer. *GReAT* [AKK+06] is a well-known graph transformation language with asynchronous behaviour. For example, Figure 7.4(a) presents a Test block where two Cases (atomic or composite rules) can be applied. When a Test block receives a packet in *GReAT*, the packet is tested on all the Cases. If multiple Cases succeed, only one will be applied non-deterministically.

Figure 7.4 shows how a Test block can be re-constructed using *MoTif-Core* primitives. The *MoTif-*

[8] For this simple rule, the *max* parameter of the Matcher is set to 1 since the rule is applied only once.

Core model consists of a Composer composing two SimpleRules—one for each Case, assuming they each consist of a simple rule described above. However, a Selector (Cut) ensures that at most one of them will get applied by sending the appropriate cancel event to the SimpleRule as soon as a packet is received. Since *GReAT* is not a timed model transformation language, we assume that the Matcher and the Rewriter of both rules consume the same time. The Composer has the following inner models: two Matchers, two Iterators, two Rewriters, and one Selector. As before, M is the set of these three inner models and N is the set of labels. The connection topology is given by the influencee sets: $I_{:\mathsf{Case1:M}} = I_{:\mathsf{Case2:M}} = \{:\mathsf{Cut:S}\}$, $I_{:\mathsf{Cut:S}} = \{:\mathsf{Case1:M}, :\mathsf{Case2:M}, :\mathsf{Case1:I}, :\mathsf{Case2:I}\}$, $I_{:\mathsf{Case1:I}} = \{:\mathsf{Test}, :\mathsf{Case1:W}\}$, $I_{:\mathsf{Case2:I}} = \{:\mathsf{Test}, :\mathsf{Case2:W}\}$, $I_{:\mathsf{Case1:W}} = I_{:\mathsf{Case2:W}} = \{:\mathsf{Test}\}$, and $I_{:\mathsf{Test}} = \{:\mathsf{Case1:M}, :\mathsf{Case2:M}, :\mathsf{Cut:S}, :\mathsf{Case1:I}, :\mathsf{Case2:I}, :\mathsf{Case1:W}, :\mathsf{Case2:W}\}$. Again, the Composer influences all its inner models because of the cancel ports connections.

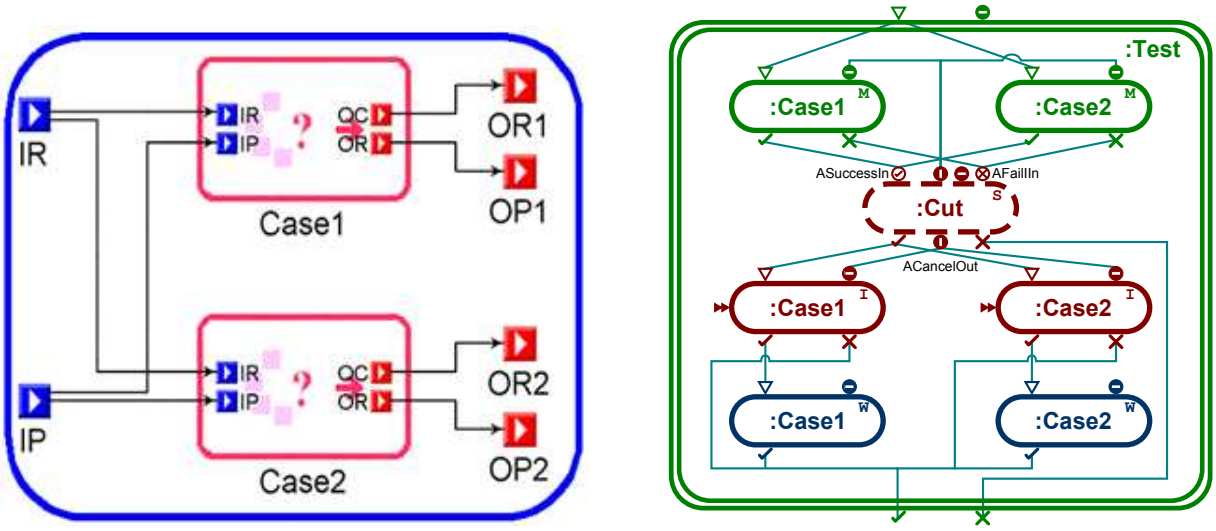
The behaviour goes as follows. First, the two Matchers each receive a clone[9] of the packet that the Composer received. Then, assuming both Matchers are in success mode, the select function Ξ of the Composer will choose one of them to output. Without loss of generality, suppose Case1 is chosen. Consequently the Selector receives the packet from ASuccessIn and immediately sends it through ASuccessOut which yields to both Iterators receiving the packet. But only Case1 elements must be activated. To solve this problem, the Selector also sends a cancel event through ACancelOut which both Iterators receive and only the one mentioned in the `exclusion` list (in this case Case1) does not cancel its activity. The same behaviour as previously described for the SimpleRule follows from there. For the case where one of the Matchers fails, the same scenario applies since the Composer will still select the successful one. Finally, in the case where both Matchers fail the Composer selects one of them and the Selector sends the packet it receives through AFailOut. Therefore the *MoTif-Core* model is indeed semantically equivalent to the asynchronous Test block in *GReAT*.

The reason why cancel events have been added to the *MoTif-Core* language is because *DEVS* lacks of dynamic re-structuring. That is, once modelled, the atomic and coupled DEVS models as well as the connection topology can no longer be modified. Dynamic structure DEVS (DSDEVS) [Bar95] overcomes this limitation. It allows the structure of a model to change dynamically. Structural changes include changes in connection topology and the creation and deletion of components in a coupled DEVS model. The changes are triggered by particular state-conditions. We have not chosen DSDEVS variant as a semantic because we do not believe the expressive power offered by DSDEVS is useful for us. At the (transformed) model level, we have dynamic structure thanks to the nature of graph transformation. At the level of the rule scheduling, the control/data flow logic which classic DEVS offers is more than sufficient. Using DSDEVS would allow for the rule scheduling to change dynamically which hardly seems useful.

7.3 Properties of MoTif-Core

The following outlines some of the most remarkable properties of *MoTif-Core*.

[9] Cloning the events is necessary to preserve *DEVS* properties, as two atomic DEVS models shall not share the same (sub-)state. In this case, modifying the packet in one Matcher (*e.g.*, storing the matches found) should not affect the packet of the other Matcher.



(a) (b)

Figure 7.4: A Test block in *GReAT* showing two cases in (a) and its equivalent model in *MoTif-Core* in (b).

7.3.1 Soundness

When connecting *MoTif-Core* elements, through port channelling, the resulting model defines an execution flow of a graph transformation. In order to ensure a **proper flow**, Lemma 1 states that whenever a packet is received by a *MoTif-Core* primitive, a packet will be output from that entity. This is true as long as no cancel event is received.

Lemma 1. For any AtomicPrimitive $\langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$, the following must hold:

$$\forall s \in S, \forall e \in T, \forall x \in X \setminus \texttt{INSTANCESOF}(\text{Cancel}), \lambda(\delta_{ext}((s,e),x)) \neq \phi \Leftrightarrow \tau(\delta_{ext}((s,e),x)) < +\infty$$

Proof. Since by construction, $\forall s \in S, \lambda(s) \neq \phi$, we need to show that $\tau(s') < +\infty \Rightarrow s' = \delta_{ext}(s)$. That is, if the time advance of a state s' is finite then s' was computed from the external transition function. Note that for all atomic primitives except the Synchronizer, $\tau(s) < +\infty \Leftrightarrow \alpha = \texttt{true}$ if and only if a packet is received without a cancel event.

For the Synchronizer, we have $\tau(s') < +\infty \Rightarrow |TC.\texttt{success}| + |TC.\texttt{fail}| = threads$. Recall that the size of `success` is only incremented by the `successIn` operation and the size of `fail` is only incremented by the `failIn` operation. Both operations are applied in the context of δ_{ext} only, when a packet is received without a cancel event. Hence the condition $|TC.\texttt{success}| + |TC.\texttt{fail}| = threads$ is satisfied only after the application of δ_{ext}.

On the other hand, note that for all atomic primitives, neither s_0 nor the state produced by δ_{int} lead to an output as the time advance is infinite. □

7.3.2 Complete and Autonomous System

A *MoTif-Core* model is a *DEVS* model specialized for transformation purposes. It can therefore be combined with **arbitrary AtomicDEVS and CoupledDEVS models**. The implementation of *MoTif-Core* makes use of this to specify the interface for the input of the host model in a similar way as in [SV07]. A *user* AtomicDEVS sends the initial packet to a *transformation* CoupledDEVS which encapsulates the *MoTif-Core* transformation model. The interface for outputting the result of the transformation is also modelled, as it is sent back to the user block. Hence, *MoTif-Core* is a **complete system** since all the components are modelled in *DEVS*. It is also **autonomous** as it does not require interaction with the outside world.

7.3.3 Time and Asynchrony

MoTif-Core is inherently a **timed** event-based language. This allows transformation languages to specify the time duration for the matching phase (Matcher) and for the rewriting phase (Rewriter) separately. Furthermore, the remaining AtomicDEVS elements do not consume time (their time advance is 0). Thus the transformation modeller has control over timing at the transformation unit level. This allows a transformation engineer to conveniently specify the behaviour of languages modelling reactive systems, *e.g.*, modelling lag time or delays. Another application of transformations enriched with the dimension of time is in simulation-based design. In this case, a simulated time system can later be replaced with a real-time clock. Chapter 9 shows such an application.

Being discrete-event based, *MoTif-Core* allows one to specify **asynchronous** transformation languages and can provide a data flow of execution. That is, several packets can be processed simultaneously, and hence the transformation can potentially be parallelized. Nevertheless, the timing specified by the time advances may induce delays which implicitly give a causal dependency between concurrent transformation units.

Traditionally in graph transformation, dependency between rules was only based on the input model and the specification of the rule patterns (LHS and RHS). This leads to **causal relationships** between rules at run-time. In *MoTif-Core*, rule causality depends on several factors. Orthogonal to the aforementioned factors, the duration for executing a Matcher or a Rewriter also affects the flow of the transformation. Furthermore, pivot exchange may prevent a match from occurring or let the matching focus only on part of the input model. Together with how ports are connected, this may affect the sequence of rules application in the overall transformation.

7.3.4 Well-formed transformation language

Lemma 2 ensures that the execution of a *MoTif-Core* model is well-formed. Model transformation is inherently non-deterministic. The non-determinism appears in two places: the location of application in the host model and the choice of which rule to apply. The former is because the pre-condition pattern of a rule may have multiple matches in the host model. The latter is because rule-based model transformation is declarative, allowing rules to be partially ordered. Therefore, *MoTif-Core* models must have a non-deterministic execution, but each choice must be repeatable. This is what we call a

well-formed execution.

Lemma 2. Every *MoTif-Core* model has a well-formed model transformation execution.

Proof. A *DEVS* model is deterministic if:

- all atomic models have their δ_{ext}, δ_{int}, and λ functions deterministic, and
- all coupled models have their Ξ function deterministic.

It is trivial to see that δ_{ext}, δ_{int}, and λ are all deterministic for all AtomicPrimitives. However, this imposes that the matching algorithm of the TCMatcher must be deterministic, as well as the selection algorithm of `packetIn` operation of the TCIterator, the resolution function of the TCResolver, and the merge function of the TCSynchronizer. In our implementation, we ensure that the abovementioned operations are deterministic by enforcing all choices to be chosen randomly in a Monte-Carlo sense, repeatable using sampling from a uniform distribution to provide a reproducible, fair sampling. However, it is the responsibility of the user to define deterministic (or random repeatable) custom resolution and merge functions. As for the select function of the Composer, random choices are repeatable as mentioned above. □

7.3.5 Modular Execution

MoTif-Core models can be executed by a **DEVS simulator** as described in Chapter 6. This allows the DEVS model to behave as a graph transformation engine. Other types of simulators can be used to execute a DEVS model, such as real-time simulators or distributed simulators. These are attractive since they do not require to modify the DEVS model (*i.e., plug-and-play*).

The time advances specified in a DEVS model evolve the time modelled explicitly. The standard DEVS simulator runs the *MoTif-Core* model in simulated time. The transformation model can also be executed by a **real-time DEVS simulator** (RT-DEVS) [HSKP97]. Our Python implementation of RT-DEVS based on `pythonDEVS` will be used to run transformation models in real-time in Chapter 9.

MoTif-Core primitives are independent from one another as they do not share any information. Furthermore, the packets they exchange contain enough information for each primitive to process them. The ControlPrimitives provide join points to parallel execution of other *MoTif-Core* primitives. Therefore, a distributed (and thus partially parallel) execution of *MoTif-Core* models is possible with an appropriate **distributed DEVS simulator** (c.f., Section 6.6).

A potential overhead of running DEVS models in parallel in a distributed environment is the select function of coupled models. Recall that it only happens when multiple internal transitions can occur simultaneously. However in *MoTif-Core*, all the work performed by the primitives is encapsulated in the external transition functions. The internal transition functions do very little work. For the Matcher, Rewriter, Resolver, Selector, and Synchronizer, the internal transition function simply resets the current state to the initial state. For the Iterator and Rollbacker it simply changes a boolean value and dereferences the temporary pointer to the packet. Also, before the internal transition is triggered, an

atomic DEVS model may output an event computed by the output function. Except for the Synchronizer, the computation of a the output function is minimal as it either sends a packet pre-computed in the external transition function, or creates a cancel event or an exception event. As for the Synchronizer, the packets are merged into a single packet in the output function. Recall that the default `merge` function first verifies whether the received packets have overlapping matches. If not, then the match set of the "merged" packet consists of the union of all the match sets from all the packets received. Therefore, the computation time of the default merge function is linear in terms of the size of the host graph. However, if a custom merge function is specified, the computation time depends on its specification.

7.4 The MoTif Language

MoTif-Core is a language for modelling transformations. From a syntactical point of view, it is a domain-specific language for modelling with *T-Core* primitives. However, its semantics relies entirely on *DEVS*. Thus the transformation engineer is required to have expertise in the *DEVS* formalism, which is rarely the case. It is hence not a trivial transformation language to design transformation models with. Instead, *MoTif-Core* is intended to offer a common platform for model transformation languages that are at a more appropriate level of abstraction. In this section, we therefore present a transformation-specific language *MoTif* whose syntax abstracts away *T-Core* constructs and whose semantics is defined in terms of *MoTif-Core*.

7.4.1 A Domain-Specific Language for General-Purpose Transformations

MoTif is a modelling language for designing model transformations. This language is engineered following MPM principles where everything is explicitly modelled at the most appropriate level of abstraction using the most appropriate formalism. Therefore its meta-model, depicted in Figure 7.5, consists of three parts: the pattern language, the scheduling language, and the transformation units. This is compatible with the transformation language engineering methodology proposed in Section 5.4.

The pattern language is automatically adapted to the domain of application of each transformation following the RAM process of Section 5.3.4. It is integrated in the meta-model of *MoTif* by extending the Pre- and PostConditionPattern classes through inheritance, the same way as in Figures 5.4 and 5.5. *MoTif* is a controlled graph transformation language. It offers a clean separation of the transformation units from the structure and flow of execution of the transformation. There are two types of transformation units. A **query** is composed of solely pre-condition patterns. They consist of a LHS and optionally NACs. A **rule** is composed of pre- and post-condition patterns, with a LHS, a RHS, and optional NACs. Transformation units are embedded in AtomicRuleBlocks that are part of the scheduling language. This language mainly consists of RuleBlocks. A RuleBlock can thus be either atomic or composite (CompositeRuleBlock). In the atomic rule blocks, an ARule ("Atomic Rule") encodes a rule and a QRule ("Query Rule") encodes a query. Composite rule blocks are used to modularly encapsulate other RuleBlocks (atomic or composite). Some express advanced control flow structures such as branching, looping, and parallelism. RuleBlocks have ports that can be connected via channels. Model is the only input port and Success, Fail, and Exception represent output ports. Exceptions

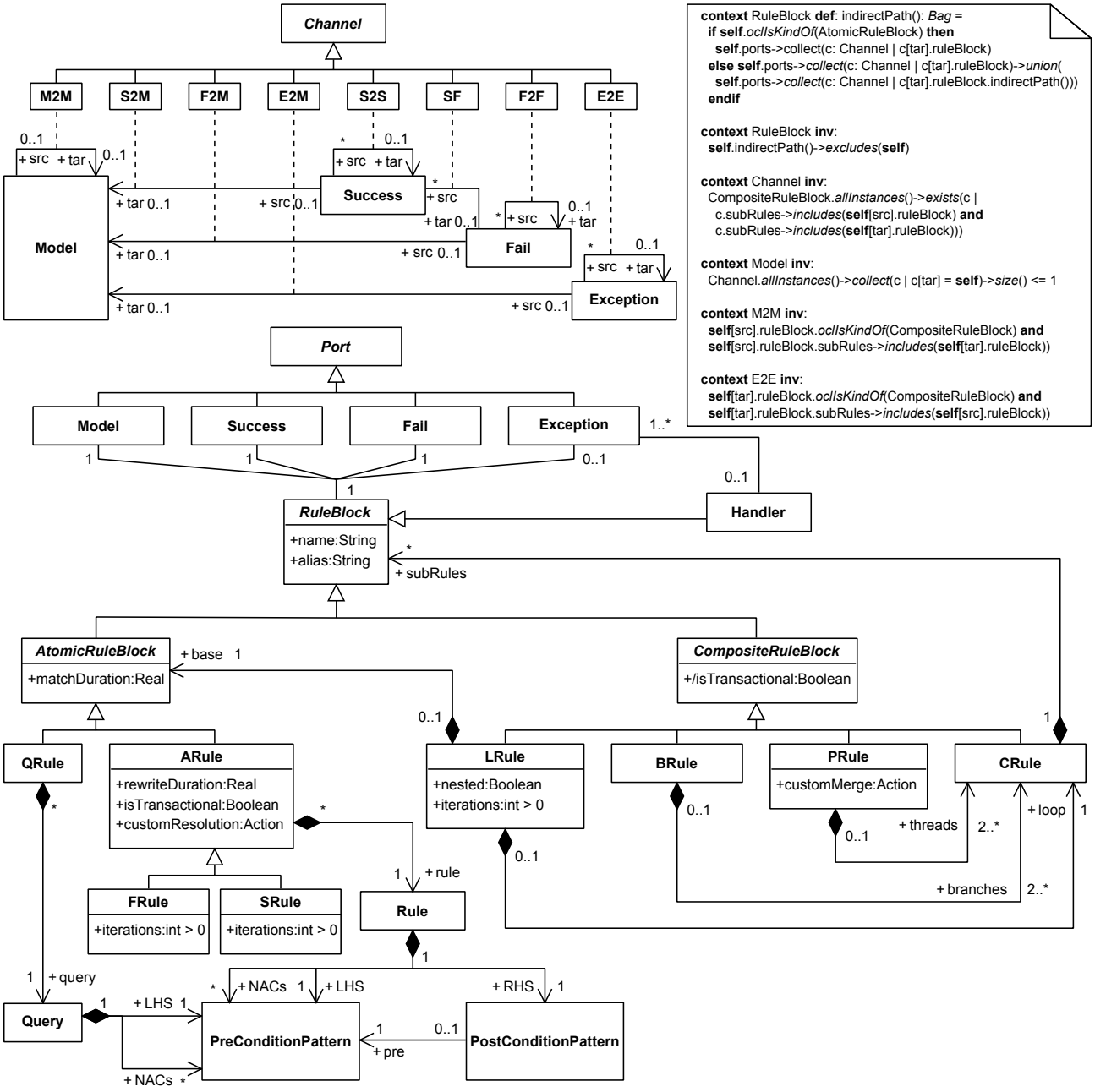


Figure 7.5: The meta-model of *MoTif*.

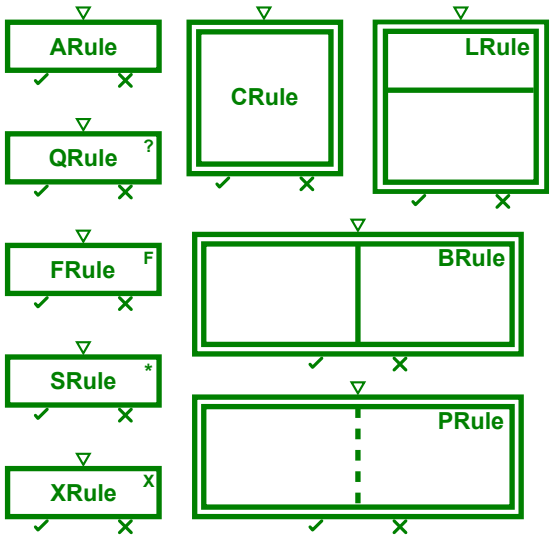


Figure 7.6: The different rule blocks in *MoTif*.

and Handlers will be discussed in the next chapter. Port channelling induces an ordered **sequence** of application of rule blocks. The OCL constraints included in the meta-model specify that:

- No path defined by a sequence of port connections shall induce a cycle;
- The source and target ports of any channel shall be within the same scope defined by composite rule blocks;
- A model port can be the target of at most one channel;
- The source port of a model to model channel shall be attached to the parent of the source's rule block;
- The target port of an exception to exception channel shall be attached to the parent of the target's rule block.

In the *MoTif* visual modelling language, the concrete syntax of an ARule is a single rectangle frame as depicted in Figure 7.6. The top triangle on a rule block is the Model input port. The bottom left tick symbol is the Success output port and the bottom right "X" symbol is the Fail output port. Conceptually, the input model is received from the Model port and, if the application of the rule is successful, the resulting model is output through the Success port. However, if the pre-condition pattern is not satisfied, the original model is output from the Fail port. The QRule has a similar graphical syntax with a question mark symbol on the top right of the rectangle. The transformation engineer can specify a duration for the matching phase for both atomic rule blocks. In the case of an ARule, the duration for the rewriting phase can also be specified.

A *MoTif* sub-model encoding transformation units can be part of a CRule ("Composite Rule"). CRules are visually depicted by a double rectangle frame. The same ports appear on both atomic and composite rule transformation models which implies that they can be used interchangeably to modularly build complex **hierarchical** transformation models.

Iterative rule application is possible with variants of an ARule. The FRule ("For all Rule") applies the transformation rule on *all* matches of the pre-condition pattern (in arbitrary, but deterministic and repeatable order). The matches are assumed to be parallel independent [EEKR97]. Two matches are parallel independent if no overlapping matched element is modified (node deletion or attribute modification) by the rule when applied. If they are not, the transformation designer can specify a resolution function to resolve the conflicts. The purpose of the FRule is purely syntactic. It is syntactic sugar for applying the rule iteratively over all matched sub-graphs. The maximum number of iterations is parametrizable. The FRule is represented using the same concrete visual syntax as an ARule, annotated with an "F" in the top right corner. If the maximum number of iterations is not infinite, the positive integer appears in the top left corner.

Another variant of the ARule is the SRule ("Star Rule"). It is applied sequentially as long as the pre-condition pattern is satisfied in the model. That is, after the received graph is matched and transformed, the resulting model is then matched by the same rule. This continues until no more matches can be found in the resulting packet. Care should be taken when using this construct as it may result in an infinite loop. When combined with pivot passing, the SRule applies itself **recursively**[10]. The SRule is represented using the same concrete visual syntax as an ARule, annotated with an asterisk in the top right corner. If the maximum number of iterations is not infinite, the positive integer appears in the top left corner.

While iteration involves a single rule block, **looping** allows one to iterate over multiple rule blocks. This is possible with the LRule ("Loop Rule"). It consists of an atomic rule block as base and a CRule as loop body. It allows applying several rules iteratively. The LRule has different variants depending on the type of the base rule block and whether pivots are used in the patterns, such as rule nesting and indirect recursion. Its concrete syntax is the same as a CRule but a horizontal solid line separates the base compartment from the loop compartment.

In graph transformation, it is sometimes desirable to have many rules match, but let only one be applied. *MoTif* introduces the BRule ("Branch Rule") block which makes of this feature for **branching**. Its purpose is to receive a model, through its Model port, and send it to each branch. However, only one branch is selected to continue executing the transformation. Visually, a BRule is very similar to a CRule but the rectangle is partitioned by vertical filled lines to separate the branches, each branch being a CRule in its own right.

MoTif allows rules to be applied in **parallel** with the PRule ("Parallel Rule"). This leads to what we call "threads" of rule applications. Each thread is applied concurrently, independently from one another. The output of a PRule is a single model "merged" from the result of each thread. The threads are assumed to be sequential independent [EEKR97]. That is, any order of application of the rules in each thread leads to the same output. If they are not, the transformation designer can specify a merge function to merge the graphs in conflict. The PRule's parallel execution requires special care. Visually, a PRule is very similar to a BRule but vertical lines that separate the threads are dashed. Each thread is a CRule in its own right.

[10] This is similar to direct recursion in procedural programming languages, where a procedure invokes itself.

When the *isTransactional* flag of a rule block is activated, its behaviour is extended with memory capacity, which provides **back-tracking**. We denote such a rule block by XRule ("Transactional Rule"). For composite rule blocks, *isTransactional* is set to true if and only if the first sub-rule is transactional. We will see in Chapter 9 that, through transactional rules, *MoTif* also allows for **recursion**. A transactional rule block has the same concrete visual syntax as the rule block, annotated with an "X".

Note how the meta-model of *MoTif* does not include any information about the data processed. This is because *MoTif* is a language that constrains the transformation modeller to only focus on describing a model transformation. In fact, the semantics of *MoTif* is defined in terms of *MoTif-Core*. The behaviour of each of its constructs is detailed in the following subsections.

7.4.2 From MoTif to MoTif-Core

Now that the abstract and concrete syntax of *MoTif* have been described, we define its semantics by mapping it onto *MoTif-Core* (*i.e.*, *DEVS*). The semantic mapping is injective since every *MoTif* construct corresponds to a unique *MoTif-Core* model. We describe this mapping as a model transformation specified in *MoTif*. It is therefore a **higher-order transformation** transforming one transformation language into another. Following the methodology described in Chapter 5, the meta-model of the patterns of this transformation consists of the RAM version of the meta-models of *MoTif* and *MoTif-Core*. In this subsection, we outline the main transformation steps of the semantic mapping.

Every RuleBlock in *MoTif* is mapped onto a Composer in *MoTif-Core*. The *alias* and *name* parameters of the Composer are the same as those of its RuleBlock counterpart. The Model port is mapped to the CPacketIn port of the Composer, the Success port to the CSuccessOut port and the Fail port to the CFailOut port. Exceptions are not considered here.

QRule

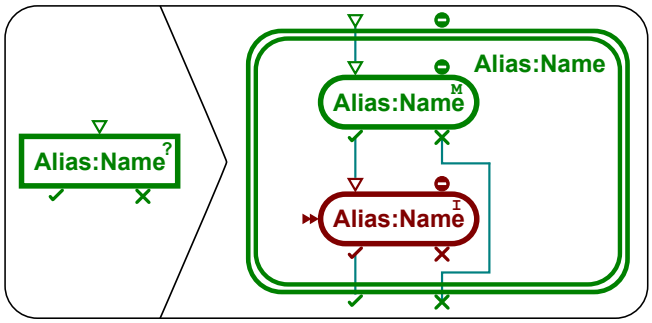


Figure 7.7: The QRule in *MoTif* and its equivalent *MoTif-Core* model.

The QRule, is the simplest transformation unit with no side effect: it only searches for one match of a pattern in the host model. Figure 7.7 illustrates how a QRule is mapped onto a *MoTif-Core* Composer. We use the following notation to refer to a QRule: $QRule_{name,alias,pre,\Delta_m}$. The rule in the figure

describes the connection topology of the Composer's sub-models[11]. We denote M_{QRule} the set of sub-models of the Composer of type QRule. For readability purposes, we will omit the *alias* and *name* parameters of *MoTif-Core*'s AtomicPrimitives which will always be set to those of *MoTif* RuleBlock, unless stated otherwise.

$$M_{QRule} = \{\text{Matcher}_{pre,\Delta_m,1}, \text{Iterator}_1\}$$

The QRule has a very simple behaviour. Given an input packet, if a match is found, the resulting packet is output via the success port. Otherwise, the original packet is sent from the fail port.

ARule

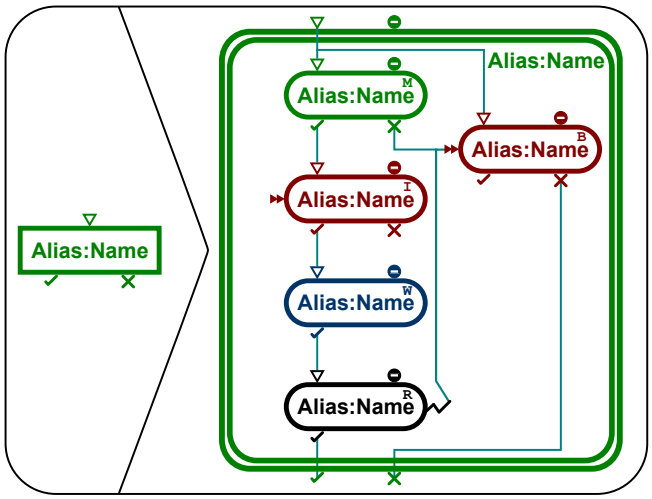


Figure 7.8: The ARule in *MoTif* and its equivalent *MoTif-Core* model.

The ARule, is the simplest transformation unit with side effect. When an ARule receives a model input from the Model port, it searches for one occurrence of its LHS in the input model. If a match is found, it is transformed according to the RHS of the rule. Figure 7.8 illustrates how an ARule is mapped onto a *MoTif-Core* Composer. We use the following notation to refer to an ARule: $ARule_{name,alias,pre,post,\Delta_m,\Delta_w,res}$. The rule in the figure describes the connection topology of the Composer's sub-models. We denote M_{ARule} the set of sub-models of the corresponding *MoTif-Core* Composer:

$$M_{ARule} = \{\text{Matcher}_{pre,\Delta_m,1}, \text{Iterator}_1, \text{Rewriter}_{post,\Delta_w}, \text{Resolver}_{res}, \text{Rollbacker}_1\}$$

An ARule behaves similarly to the simple rule described in Section 7.2.4. However, a Resolver is added in case a pending match in the packet conflicts with the current rule application. Visually, the zigzag on its right depicts the AExceptionOut port from which an exception event encapsulating the packet is output if the Resolver cannot resolve the conflicts[12]. To ensure the atomicity of the graph

[11] Once again, the connections between the CCancelIn port of the Composer and the ACancelIn ports of all its sub-models are omitted for clarity.

[12] More details on exceptional situations will be provided in the following chapter.

transformation rule, a Rollbacker is added to the *MoTif-Core* model. It ensures that if the rule is not applied, the packet will be restored to the state it was before entering the Composer.

FRule

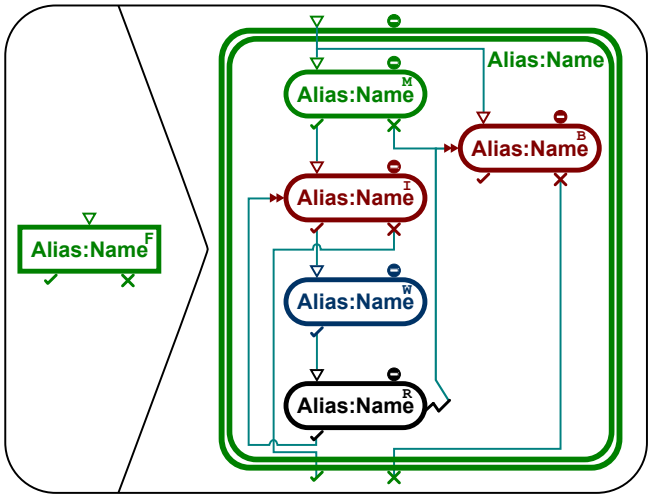


Figure 7.9: The FRule in *MoTif* and its equivalent *MoTif-Core* model.

The FRule is an ARule that applies its transformation phase on *all* the matches found before the new model is output. As shown in Figure 7.9, the matching phase is performed only once and, after the match is rewritten and validated, the packet is sent back to the Iterator that will select another match to process. Note that the Iterator failing (*i.e.*, outputs a packet from AFailOut) means that the Matcher has successfully found matches in the host graph and that there are no more matches left to process. In this case, the FRule will successfully output the new packet. If, however, the Rewriter or the Resolver fails during one of the iterations, all the modifications that had been performed in this Composer are discarded through the Rollbacker. Given an FRule defined as $FRule_{name,alias,pre,post,\Delta_m,\Delta_w,res,max}$, the sub-models of the corresponding *MoTif-Core* Composer are:

$$M_{FRule} = \left\{ \mathsf{Matcher}_{pre,\Delta_m,max}, \mathsf{Iterator}_{max}, \mathsf{Rewriter}_{post,\Delta_w}, \mathsf{Resolver}_{res}, \mathsf{Rollbacker}_1 \right\}$$

The user can control the number of times the rule encoded in the FRule is applied. If $max = \infty$, it will be applied on all possible matches. The order in which matches are processed is non-deterministic as it relies on the behaviour of *T-Core*'s TCIterator. Also, the TCResolver will by default fail if any two matches overlap. This will result in discarding all previous transformations performed by this FRule. This seems like an overhead as confluence of the matches could have been detected prior to the execution, through the computation of critical pairs [HKT02] for example. However, the latter approach may sometimes be too conservative leading to false positives (an example is given in [HHT02]). This is overcome in *MoTif* by letting the user override the validation criteria *res*.

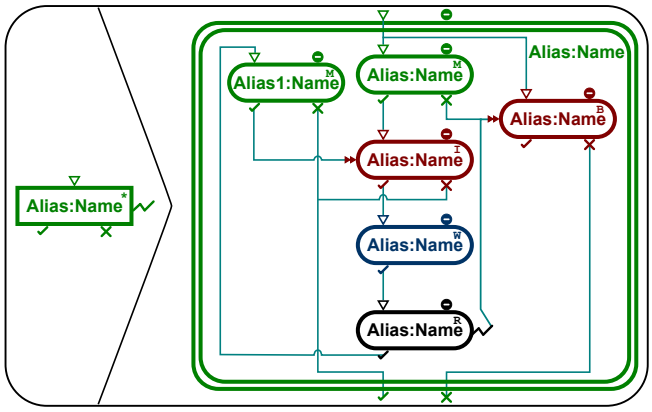


Figure 7.10: The SRule in *MoTif* and its equivalent *MoTif-Core* model.

SRule

The SRule, is an ARule that applies the rule *as long as possible*. That is, after the received model is matched and transformed, the resulting packet is then matched again by the same rule. This continues until no more matches can be found in the resulting packet. As shown in Figure 7.10, the application of the SRule is considered successful if at least one match is found and transformed. For this purpose, after the packet has been matched, transformed, and validated a first time, it is sent to a different Matcher that will ensure the loop. A clone of the original Matcher is needed to distinguish the first non-occurrence of matches (in which case the SRule fails) from subsequent ones (in which case the SRule succeeds). The second Matcher sends a packet to the ANextIn port of the Iterator rather than to its APacketIn port to keep track of the number of iterations remaining. This then requires that both Matchers refer to the same pre-condition pattern. Given an SRule defined as $SRule_{name,alias,pre,post,\Delta_m,\Delta_w,res,max}$, the sub-models of the corresponding *MoTif-Core* Composer are:

$$M_{SRule} = \{\ \mathsf{Matcher}_{pre,\Delta_m,1}, \mathsf{Matcher}_{name,alias+'1',pre,\Delta_m,1}, \mathsf{Iterator}_{max}, \mathsf{Rewriter}_{post,\Delta_w}, \mathsf{Resolver}_{res}, \mathsf{Rollbacker}_1\}$$

XRule and its variants

The XARule, is the transactional version of an ARule. It is an atomic rule with the capability of rolling-back the packet to the state before the last application of this rule. As shown in Figure 7.11, the application of the XARule has a similar topology to the ARule with the addition of a second Rollbacker. Given an XARule defined as $XARule_{name,alias,pre,post,\Delta_m,\Delta_w,res}$, the sub-models of the corresponding *MoTif-Core* Composer are:

$$M_{XARule} = \{\ \mathsf{Matcher}_{pre,\Delta_m,\infty}, \mathsf{Iterator}_1, \mathsf{Rewriter}_{post,\Delta_w}, \mathsf{Resolver}_{res}, \mathsf{Rollbacker}_{name,alias,1}, \mathsf{Rollbacker}_{name,alias+'1'}\}$$

The Matcher now needs to search for all matches as they may all be processed due to roll-backing. It then sends the resulting packet to the Rollbacker which checkpoints the packet. Recall that its

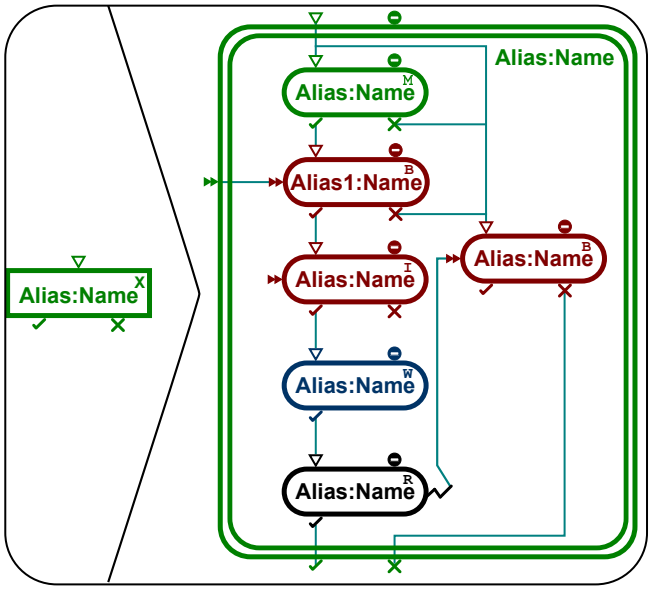


Figure 7.11: The XARule in *MoTif* and its equivalent *MoTif-Core* model.

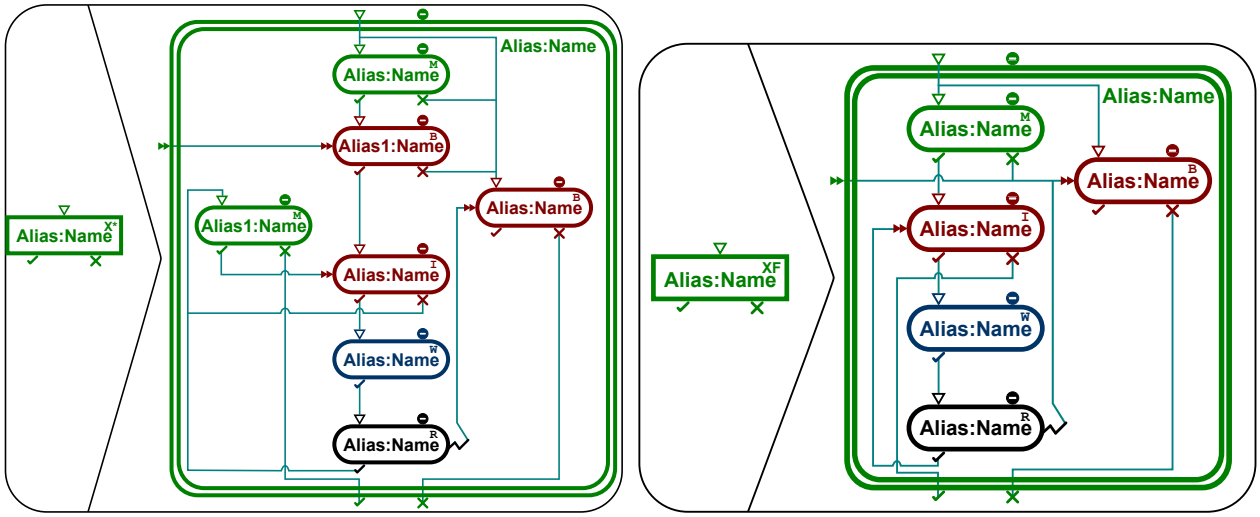
maximum number of iterations is set to the number of matches found. Then, the Iterator receives the packet output by the Rollbacker (which is the same as the one output from the Matcher). It selects a match and outputs it to the Rewriter. The same behaviour as for the ARule follows from there. After some time, the Composer may receive an event from its CNextIn port and forwards it to the Rollbacker to undo the previous effect of this rule and attempt to apply it on a different match. When all matches have been tried, the packet is restored to its original state and output via the fail port.

The QRule also has a transactional variant. Although it does not modify the input model, it nevertheless adds match sets to the packet. The topology of the *MoTif-Core* model equivalent to the XRule is the same as for the XARule, but without Rewriter and Resolver.

Figure 7.12(a) shows the XSRule, the transactional version of an SRule. The XSRule is the transactional version of an SRule. When the Composer receives a packet from the CPacketIn port, the XSRule applies the rule first on the original model received. Subsequent transformations are applied on the previously transformed model. However, the order in which the matches are selected may lead to different final results. Therefore when a roll-back is required, all previous transformations of the rule are discarded and the following sequence of transformations starts with a different initial match selected. This is why the first Matcher must find all matches. Given an SRule defined as $SRule_{name,alias,pre,post,\Delta_m,\Delta_w,res,max}$, the sub-models of the corresponding *MoTif-Core* Composer are:

$$M_{SRule} = \{\ \mathsf{Matcher}_{pre,\Delta_m,\infty}, \mathsf{Matcher}_{name,alias+'1',pre,\Delta_m,1}, \mathsf{Iterator}_{max}, \mathsf{Rewriter}_{post,\Delta_w}, \mathsf{Resolver}_{res}, \mathsf{Rollbacker}_{name,alias,1}, \mathsf{Rollbacker}_{name,alias+'1'}\}$$

Figure 7.12(b) shows the XFRule, the transactional version of an FRule. In this case, only one Rollbacker is needed since the FRule consumes all the matches. Therefore, when the Composer receives



(a) (b)

Figure 7.12: The XSRule in (a) and the XFRule in (b) and their equivalent *MoTif-Core* models.

an event from the CNextIn port, all changes affected by the XFRule are discarded and the original packet is output. Given an XFRule defined as $XFRule_{name,alias,pre,post,\Delta_m,\Delta_w,res,max}$, the sub-models of the corresponding *MoTif-Core* Composer are:

$$M_{XFRule} = \left\{ \mathsf{Matcher}_{pre,\Delta_m,max}, \mathsf{Iterator}_{max}, \mathsf{Rewriter}_{post,\Delta_w}, \mathsf{Resolver}_{res}, \mathsf{Rollbacker}_1 \right\}$$

CRule

The CRule, is a composite rule that allows one to group rule blocks modularly. It provides abstraction of a phase or concern of the transformation. Semantically, a CRule is mapped to a *MoTif-Core* Composer which has the same purpose. According to the meta-model of *MoTif*, the Model port of the CRule can be connected to at most one sub-rule. However, several sub-rules may be connected to its Success or Fail ports.

A CRule is transactional if its first sub-rule is. Consider the example in Figure 7.13 which shows a *MoTif* model on the left consisting of a CRule, named C, with four sub-rules. R1 and R3 are XRules and R2 and R4 are ARules. In case they all succeed, the channel connections indicate that the rule blocks are applied in alphabetical order. If R1 fails, then the transformation terminates in failure. In case any other rule block fails, the transformation is halted indefinitely. However, since C is transactional, the channels are implicit. If a rule block with an unconnected port fails, the transformation rolls-back to the previous XRule recursively, until the first rule block. In the example, if R4 fails then the transformation rolls-back to R3. But if R3 fails, then it rolls-back to R1 since R2 is not an XRule. We denote such a composite rule block an XCRule.

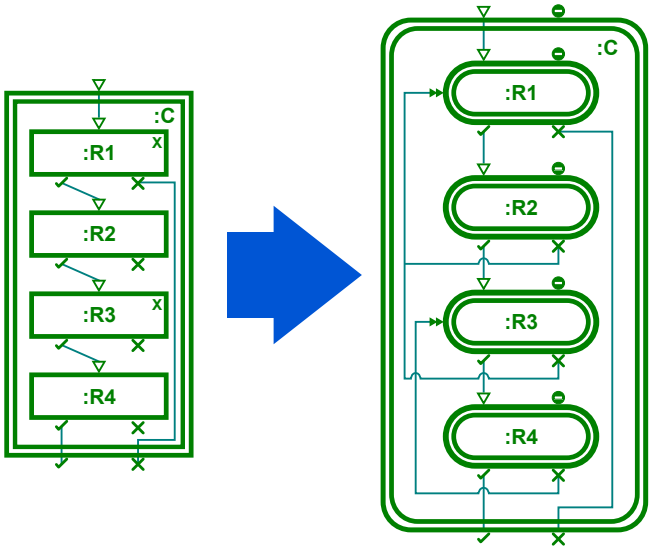


Figure 7.13: A CRule with transactional behaviour and the equivalent *MoTif-Core* model.

LRule and its variants

The LRule, is a composite rule block whose primary purpose is to loop over several rule applications, since channel connections cannot induce cycles. As depicted in Figure 7.14(a), it consists of a QRule as base and a rule block in the loop part. The inner rule block R is applied for each match encountered in the Matcher of the query. If the success and fail ports of R are not connected, then looping continues regardless of its output. Otherwise, the loop is interrupted in a similar way as a break statement interrupts a loop in programming languages. Given an LRule defined as $LRule_{name,alias,pre,post,\Delta_m,\Delta_w,max,R}$, the sub-models of the corresponding *MoTif-Core* Composer are:

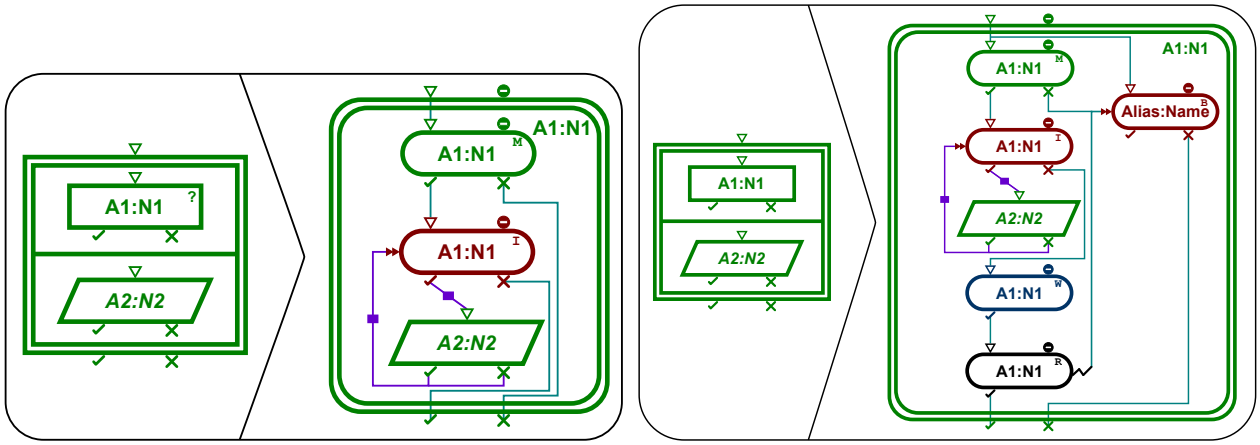
$$M_{LRule} = \{\mathsf{Matcher}_{pre,\Delta_m,max}, \mathsf{Iterator}_{max}, R\}$$

If the base rule block is an ARule, then the Rewriter in the corresponding *MoTif-Core* model is applied at the end of the loop. As Figure 7.14(a) shows, the equivalent *MoTif-Core* model of an ARule is constructed by interleaving the inner rule block between the Iterator and the Rewriter. Such a construct, called an LARule, allows one to interleave the matching and the rewriting phases of several rule blocks thereby providing *rule nesting*[13]. The LARule is defined as $LARule_{name,alias,pre,post,\Delta_m,\Delta_w,max,R}$, the sub-models of the corresponding *MoTif-Core* Composer are:

$$M_{LARule} = \{\mathsf{Matcher}_{pre,\Delta_m,max}, \mathsf{Iterator}_{max}, \mathsf{Rewriter}_{post,\Delta_w}, \mathsf{Resolver}_{res}, \mathsf{Rollbacker}_1, R\}$$

The LFRule is constructed in a similar way, replacing the ARule by an FRule. In this case, the Rewriter is applied at each iteration before the inner rule block. In its nested version, the LNFRule, the Rewriter is applied at each iteration after the inner rule block. Both constructs are illustrated in

[13] Section 3.4.2 shows how rule nesting allows one to amalgamate transformation rules.



(a) (b)

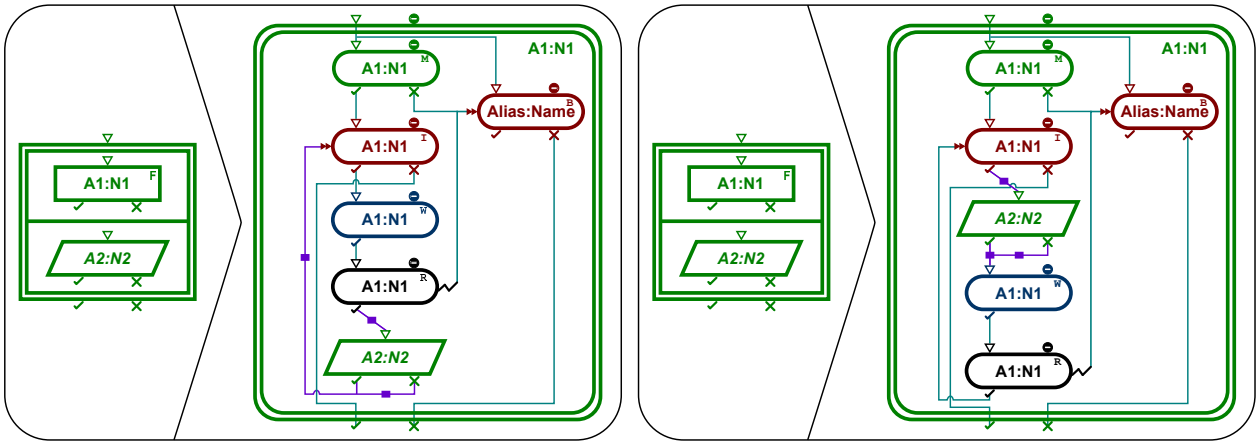
Figure 7.14: The LRule in (a) and the LARule in (b) and their equivalent *MoTif-Core* models.

Figure 7.15(a) and 7.15(b). The LSRule and LNSRule are constructed in a similar way, replacing the FRule by an SRule. If the base rule block is transactional then the LRule becomes transactional. Since all variants of the XRule either add a Rollbacker between the Matcher and the Iterator or a channel to the existing Rollbacker, all the applications of both the base and the inner rule blocks will be undone upon roll-back.

BRule

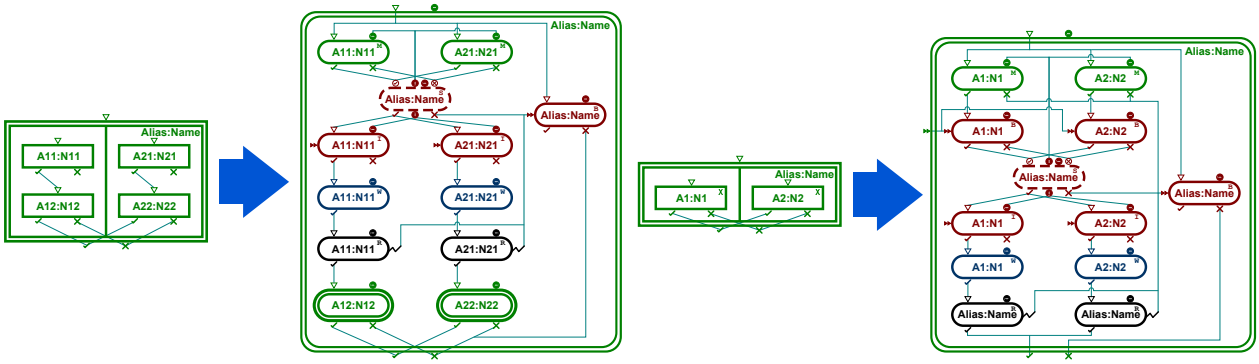
The BRule, is a composite rule block that non-deterministically selects one successful branch of execution. The selection is applied only on the first matchings of each branch. Thus when a branch starts with a composite rule block, the selection is applied only on the first atomic rule block found. Figure 7.16(a) shows the *MoTif-Core* model corresponding to a BRule with two branches, each consisting of a sequence of two ARules. The channels the Matchers output from are re-routed to a single Selector. The behaviour is similar to the `Test` rule block in Section 7.2.4. However, to preserve the atomicity property of the transformation, a Rollbacker is added to undo any changes if a Resolver does not succeed. Note that if there were only one branch in the BRule consisting of an ARule, the BRule would have the exact same behaviour as the ARule. The only difference is the addition of the Selector, but it does not consume time. Recall that the FRule, SRule and the LRule only differ from the ARule starting from the Iterator. Therefore if one replaces the ARules in Figure 7.16(a) by an FRule or an SRule or an LRule, only that part following the Iterator would have to be adapted. In case of an XRule, Rollbackers are added between the Matchers and the Selector as illustrated in Figure 7.16(b). Then the success/fail connections are delegated to the Rollbackers. This has for effect to undo all modifications performed within the BRule.

A variant of the BRule is the BSRule. It allows one to apply recursively each branch, giving the chance to both branches of re-applying at each iteration. As illustrated in Figure 7.17, the first rule block of each branch is converted into an SRule. On the first iteration, the BSRule behaves exactly



(a) (b)

Figure 7.15: The LFRule in (a) and the LNFRule in (b) and their equivalent *MoTif-Core* models.



(a) (b)

Figure 7.16: The BRule in (a) and the XBRule in (b) and their equivalent *MoTif-Core* models.

like a BRule. But on the following iterations, the cloned Matcher outputs a packet to a clone of the Selector and the normal behaviour of a BRule follows.

Since *MoTif* is a timed transformation language, matching time and rewriting time may differ from one rule block to another. The Selector will always choose the first branch that finds matches and cancel all the other branches. However, if two or more branches have their first Matcher output a packet at the same time, one of them is chosen non-deterministically by the select function of the Composer and the other one is cancelled. The output time of the BRule is the output time of the selected branch.

PRule

The PRule, is a composite rule block that provides deterministic execution of several threads of sub-transformations in parallel. The PRule succeeds if and only if all threads succeed. The output time of

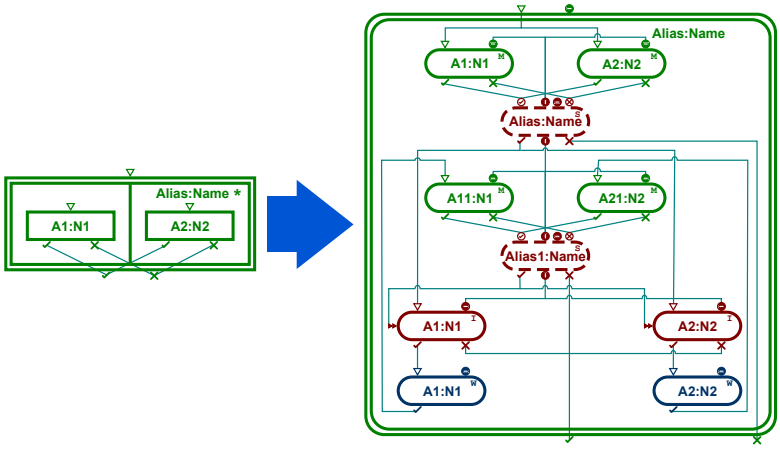


Figure 7.17: A BSRule and its equivalent *MoTif-Core* model.

the PRule is the output time of the slowest thread. We propose two alternatives for applying threads sub-transformations in parallel.

The first alternative synchronizes the threads at the end of each sub-transformation. Each thread is mapped to a Composer encapsulating the *MoTif-Core* model equivalent to the transformation of the corresponding thread, following the transformations presented in this subsection. A Synchronizer is added to the *MoTif-Core* model such that the CSuccessOut port of all the Composers are connected to the ASuccessIn port of the Synchronizer and their CFailOut port is connected to the Synchronizer's AFailIn port. The top part of Figure 7.18 illustrates such a PRule with two threads, each consisting of two ARules. The advantage of this approach is that the content of each thread can be arbitrarily complex and with arbitrary match and rewrite durations, since the synchronization and merge of packets only happens once at the end. The disadvantage is that the burden is on the transformation engineer who has to provide a merge function (if needed) that takes as input as many packets as there are threads.

The second alternative takes advantage of the fact that the transformation language has been decomposed in primitive operations. The primary purpose of a PRule is to provide optimization points in a transformation allowing concurrent operations. The critical operations in a transformation are the matching and the rewriting. Since matching is in general significantly more time consuming than rewriting (c.f. Chapter 4), it is highly desirable to parallelize this operation, on rules that are parallel independent. The right part of Figure 7.18 illustrates how the previous PRule is mapped to *MoTif-Core* in this approach. Suppose the PRule has θ threads. In *MoTif-Core*, first a clone of the received packet is fed to the θ Matchers. When a Matcher finishes processing the packet, it sends it to Synchronizers according to the success or fail output. There are 2^{θ} Synchronizers, each representing a possible combination of the Matchers' output. Since a Matcher outputs a packet exclusively from either of its outports, there is exactly one Synchronizer that performs the merge of the packets and outputs the result. Recall from Chapter 3 that if the received packets have non-overlapping match sets, the merged packet consists of the union of these match sets. Otherwise, the transformation engineer must define

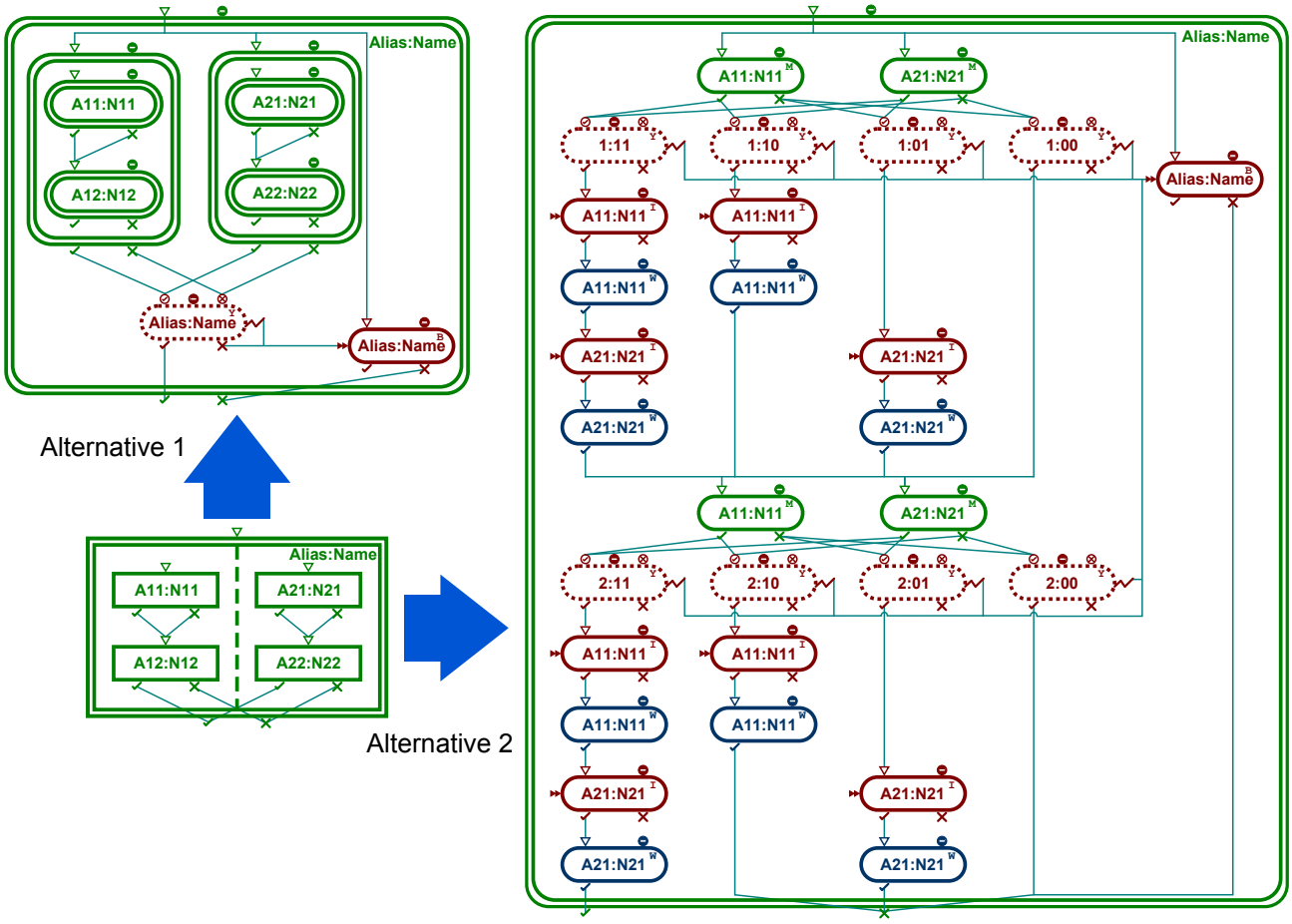


Figure 7.18: A PRule mapped to a *MoTif-Core* model according to each alternative.

the merge operation. Depending on the activated Synchronizer, the Iterator and Rewriter corresponding to the rule blocks whose Matcher succeeded are applied. After the first rule of each thread has been applied, the resulting packet is sent to the following Matchers and the same behaviour continues. The main disadvantage of this alternative is that the presented solution is well-defined for threads consisting sequences of ARule applications as depicted on the bottom left of Figure 7.18, but the number of ARules in each sequence may differ. The main advantage, however, is that synchronization and packet merging happens at a finer granularity than the first alternative, which enables the transformation engineer to (1) specify simpler merge functions and (2) know exactly where the packets are not mergeable.

In our implementation, if each thread of a PRule only consists of a sequence of ARules and QRules, then the PRule is transformed into a *MoTif-Core* model according to the second alternative; otherwise according to the first alternative. Here, we set the stage for future work to implement a possibly parallel or distributed transformation based on these alternatives.

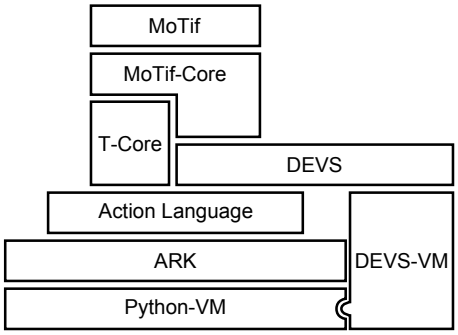


Figure 7.19: The architecture of the *MoTif* language.

7.5 Running MoTif models

MoTif is a completely modelled model transformation language. Its syntax is convenient to use for a transformation engineer in the sense that it only contains artefacts specific to transformations (unlike *MoTif-Core*). Its semantics is also modelled since it is mapped onto the *MoTif-Core* modelling language and this mapping is modelled as a transformation. Figure 7.19 illustrates the different language layers *MoTif* relies on. *MoTif* is a syntactic sugar language of *MoTif-Core*, which consists of the core elements of the language. The former language simply defines a more user-friendly syntax encapsulating the different transformation operators provided in the latter language. *MoTif-Core* combines *T-Core* and *DEVS*, both running on a model-aware virtual machine. They are expressed in a neutral target language as defined by the AToM^3 Redux Kernel (ARK), which represents the meta-meta-modelling layer in *AToM*^{3}. The tool is implemented in Python. The *DEVS* virtual machine allows executing *MoTif* transformations.

Figure 7.20 represents the framework in which *MoTif* transformation models are executed. *MoTif* is a formalism defined in *AToM*^{3} as a domain-specific language. To define a transformation, the transformation engineer transforms the source and target meta-models with the semi-automatic RAM process. The result is combined with the transformation unit part of the meta-model of *MoTif* and produces a customized meta-model for the patterns of the transformation. On the one hand, the transformation engineer defines rules, queries, and their patterns in the modelling environment described in Section 5.3.5. They are then automatically compiled into *T-Core* patterns. On the other hand, the transformation engineer specifies the control flow of the transformation by designing a *MoTif* model. The atomic rule blocks refer to the transformation units already created. Then the *MoTif* model is transformed into an equivalent *MoTif-Core* model as outlined in Section 7.4.2. The resulting model is further compiled into a *PythonDEVS* model following the mapping defined in Section 7.2. The generated *T-Core* patterns are integrated in the DEVS model through package imports. A *PythonDEVS* environment is also generated from the *MoTif-Core* model. It allows one to interact, execute, and debug the *PythonDEVS* model.

After a *MoTif-Core* model is generated from the *MoTif* model, some optimization are performed to reduce unnecessary overhead of DEVS artefacts. For example, the transformation systematically

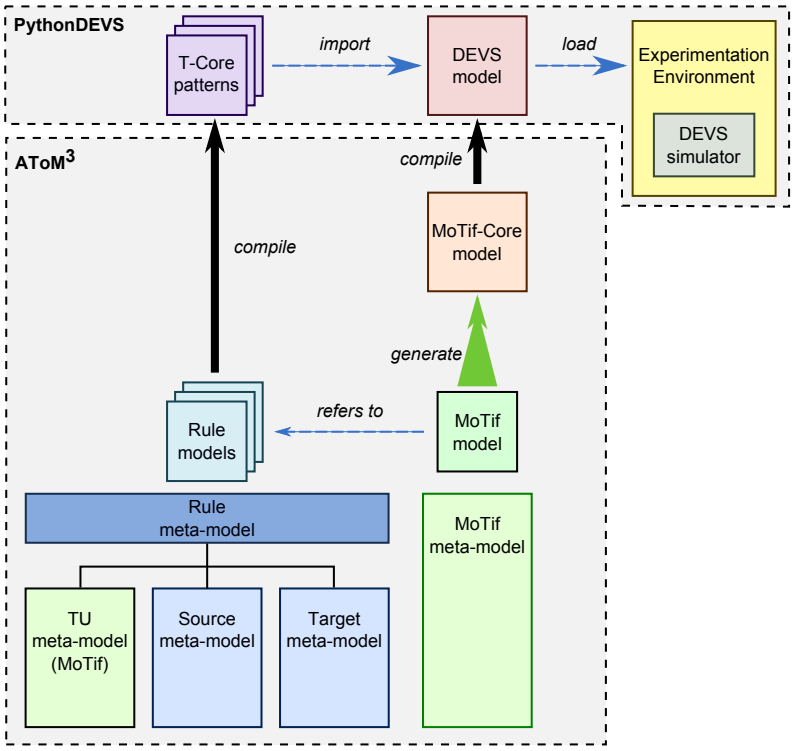


Figure 7.20: The *MoTif* execution framework.

includes a Resolver in an ARule. Its purpose is to verify if the application of the rule conflicts with any pending matches. Consider one of the branches of the SRule in Figure 7.16(a). The ARule `A12:N12` is applied sequentially after the ARule `A11:N11`. Therefore the packet will not contain any match set after `A11:N11` thus the application of `A12:N12` cannot be in conflict with any other rule. In fact, the only cases where an ARule can conflict with other matches is if it is part of an LRule (in the loop) or a PRule. Hence the resulting *MoTif-Core* model of an ARule is the simple rule depicted in Figure 7.3, since no Rollbacker is needed anymore. A similar reasoning for the SRule can be followed. If it is not part of the loop of an LRule variant or the base an LNSRule or part of a PRule, then no Resolver and Rollbacker are required.

7.6 Enabling Higher-Order Transformation

To illustrate *MoTif*, we extend the example presented in Chapter 5 in which a *MoTif* transformation maps a finite state automaton to a Petri net model. We will design a Petri net simulator in *MoTif* that defines the operational semantics of the Petri net model and, by transitivity, the operational semantics of the finite state automaton it was translated from. Then we show how, using both transformations, one can specify a higher-order transformation to animate the finite state automaton while the Petri net model is simulated.

7.6.1 Petri Net Semantics

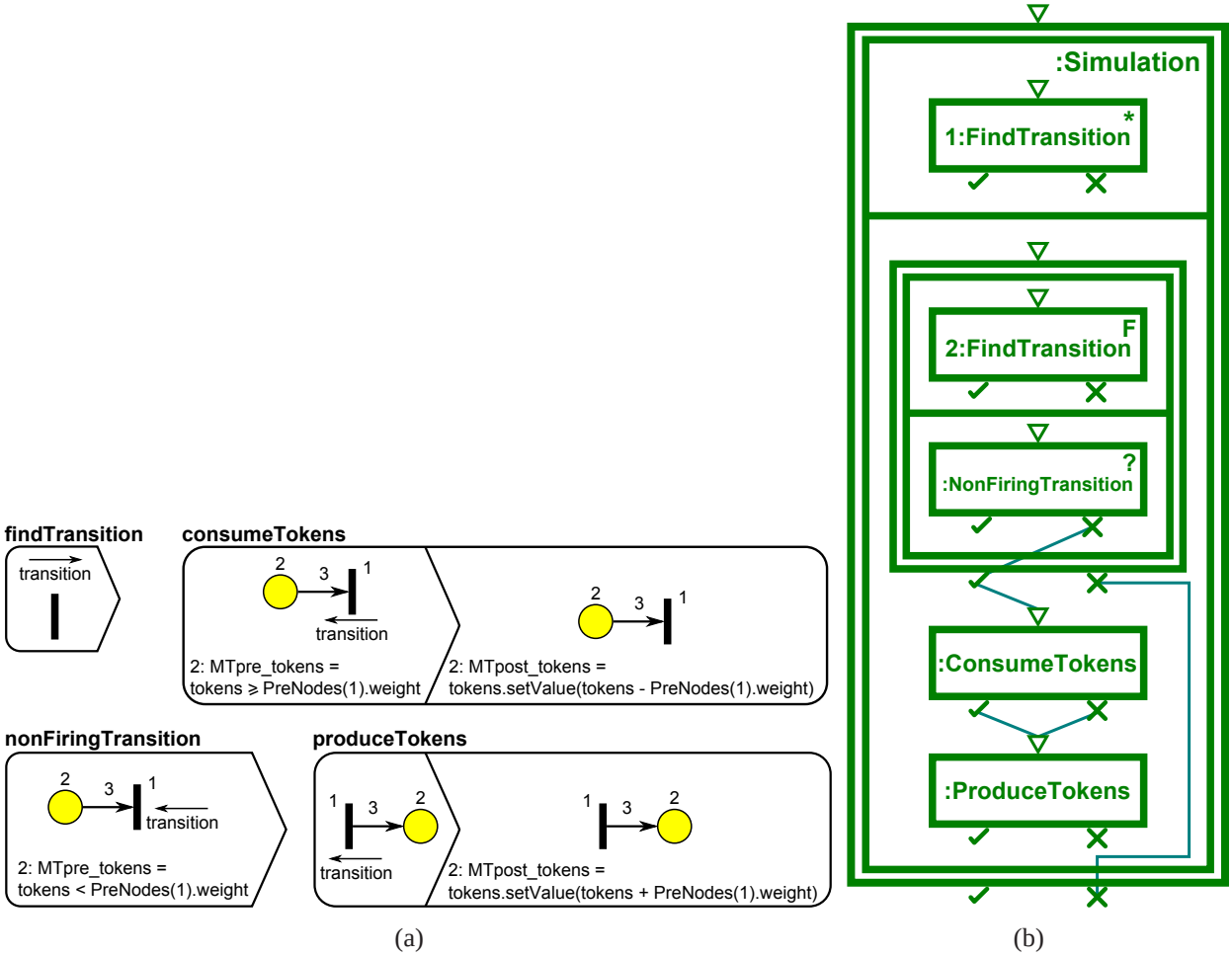


Figure 7.21: The operational semantics for Petri nets: the rules in (a) and the control flow in (b).

We simulate the execution of a Petri net execution by using a small set of transformation rules, transforming Petri nets to Petri nets. This realizes an operational semantics of Petri nets (see Figure 7.21(a)). We are able to express the operational semantics in just four simple rules thanks to the expressiveness of *MoTif* control structures. The respective control structure is shown in Figure 7.21(b). The control structure makes it particularly easy to find an enabled Petri net transition, *i.e.*, one which can fire. Such a transition needs sufficiently many tokens at *each* of its incoming transitions. One naive solution for finding enabled transitions is to just specify all possible patterns to be found as sub-graph isomorphisms. Alternatively, this can be solved provided that the pattern specification language uses intentional specifications to allow referring to sub-graphs of arbitrary size. However, the most elegant solution is to iterate through all transitions until one has been found that does *not* satisfy the pattern of a *non*-firing transition.

The behaviour of the transformation model goes as follows. The outer-most rule block is an

LSRule called Simulation—since the base rule block of this LRule is an SRule. Although the base is a query, it is nevertheless encapsulated in an SRule (with no rewriting phase) to recursively execute the transformation. First, 1:FindTransition looks for one transition. The transition found is assigned to a pivot called *transition*. In the loop part of the LSRule, an LFRule ensures that only firing transitions will be processed. This is done by iterating over every transition in the model and, if one of them can not fire, it is assigned to the pivot in order to fire the transition. This interruption in the loop is represented by the channel from the fail port of the NonFiringTransition QRule to the success port of the enclosing LFRule. When a firing transition is found, it is assigned to a pivot called *transition*, replacing the former transition. Then, tokens are transferred along this transition as depicted by rules ConsumeTokens and ProduceTokens. After that, the first rule block Simulation is applied again recursively, by re-matching the new model looking for a transition. This control flow goes on until no more transitions are fire-able.

Next we will use a higher-order transformation to extend the control structure shown in Figure 7.21(b) to include an animation component.

7.6.2 Background on Higher-Order Transformation

Tisi *et al.* [TJF+09] define higher-order transformation (HOT) as "a model transformation such that its input and/or output models are themselves transformation models". A HOT is a model transformation and is therefore an automatic manipulation of a model with a specific intention (c.f., Section 1.1). The difference with a (first-order) model transformation is that at least one of the input or output artefacts of the HOT must be a transformation model:

- A HOT transforms a transformation model into a transformation model;
- A HOT transforms a non-transformation model into a transformation model;
- A HOT transforms a transformation model into a non-transformation model;
- A HOT generates a transformation model from nothing.

The authors use the term "transformation model" which was first introduced by Bézivin *et al.* in [BBG+06]. The meaning is that transformations must be modelled, which matches the philosophy adopted in this thesis.

There is ample motivation for transforming transformations by means of higher-order transformations. Promising application areas include:

Transformation synthesis is the category of HOTs where the output is a transformation model. On the one hand, this type of HOT is used to reduce the abstraction level such as mapping from *QVT-Relations* to *QVT-Core* [Obj08] (with respect to traces) or the mapping from *MoTif* to *MoTif-Core* (with respect to DEVS). The implementation of triple graph grammars (TGGs) transforms declarative specifications (TGG rules) to operational specifications (TGG operational rules) as described in [KKS07]. On the other hand, a HOT can synthesize generic transformations. That is, the input or output meta-models of a transformation is not known a priori: the HOT generates a transformation on-the-fly specific to the input or output meta-models. For

example in [GvD07], a HOT takes as input a meta-model *MM* and outputs a transformation model M_T. M_T takes as input two models that conform to *MM* and outputs the difference model between them: the union of both models annotated with "similar", "from left", or "from right" stereotypes.

Transformation analysis is the category of HOTs where the input is a transformation model and the output is not. On the one hand, it can produce a view on the transformation model by querying it. On the other hand, the HOT generates data information from a transformation for analysis purposes. For example in [dLV10], de Lara and Vangheluwe transform the operational semantics of a language defined as a graph transformation, together with an instance model of the language, into a Petri-Net. This Petri net can then be used to analyze properties of the model transformation.

Transformation composition is the category of HOTs where the input consists of several transformation models and the output is a single transformation model. This kind of HOT allows one to compose two transformations into one. For instance in his thesis [VG08], Van Gorp has applied HOT to desugar (convert into primitive constructs) a "copy" operator for graph transformations. Another application is the separation of transformation concerns. Instead of putting all functionality into one transformation, one can split concerns over many transformations and integrate them by sequentially adding them with higher-order transformations to a base transformation. Often one may use multi-stage transformations for the same effect, but sometimes this is not a viable option (see next section). Separating transformation concerns from each other does not only reduce the complexity of an otherwise monolithic transformation but also opens up the opportunity to re-use (higher-order) transformations. Conversely, **transformation de-composition** takes as input one transformation model and outputs several ones.

Transformation migration is the category of HOTs where the input is a model and the output is a transformation model. Whenever the definition of a language evolves, any associated transformations have to be adapted. This adaptation process may sometimes be semi-automated by using a HOT generated from the modifications made to the language.

Transformation modification is the category of HOTs where the transformation is applied on a transformation model in-place. On the one hand, the semantics of the transformation engine can be enhanced. For example, adding a copy operator to transformation rules [VG08] or a grouping mechanism in the patterns [BNN+07]. On the other hand, the transformation can be optimized by improving the transformation definition for more efficient results or by refactoring it based on best practices.

7.6.3 Source-Level Animation

In conjunction, the "Finite State Automaton to Petri Net" and Petri net operational semantics transformations presented simulate a language recognizer but without visualizing the execution at the source

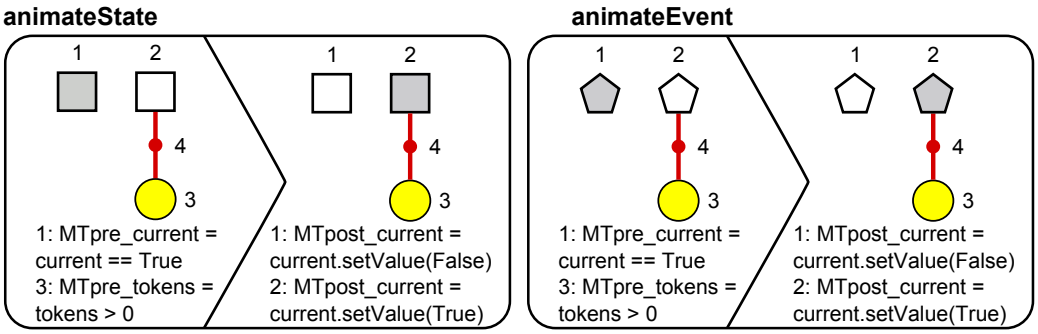


Figure 7.22: The animation rules.

level of state automata. In order to add source-level animation, we need to add update rules (see Figure 7.22) to the operational Petri net semantics. *AToM*^{3} then automatically takes care of updating the respective concrete syntax.

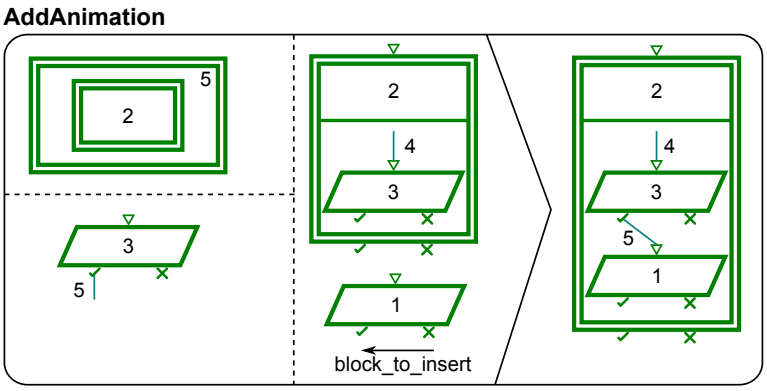


Figure 7.23: Higher-Order Transformation: Animation.

Having explicitly modelled all artefacts, we are now in a position to define a HOT that automatically adds the animation rules to the operational semantics transformation. The rule block labelled 1 at the bottom of the LHS pattern in Figure 7.23 is a parameter to the HOT `AddAnimation` and contains all the animation rules of Figure 7.22. We perform the parameter passing by using the block as a pivot model. The RHS of the HOT `AddAnimation` simply links the animation block into the main loop of the original semantics control structure (see Figure 7.21(b)), making sure this happens only once at the top level (hence the NAC).

Figure 7.24 shows the result of applying the HOT `AddAnimation` to the original control flow shown in Figure 7.21(b). Note that being able to modify an explicit representation of a transformation's control structure allowed us to design `AddAnimation` in a way that makes it reusable. As long as one designs other operational semantics definitions with a similar overall main loop and provides corresponding update rules as a parameter to `AddAnimation`, the latter can be re-used as is.

Had we designed the operational semantics transformation to perform a single step only, it would

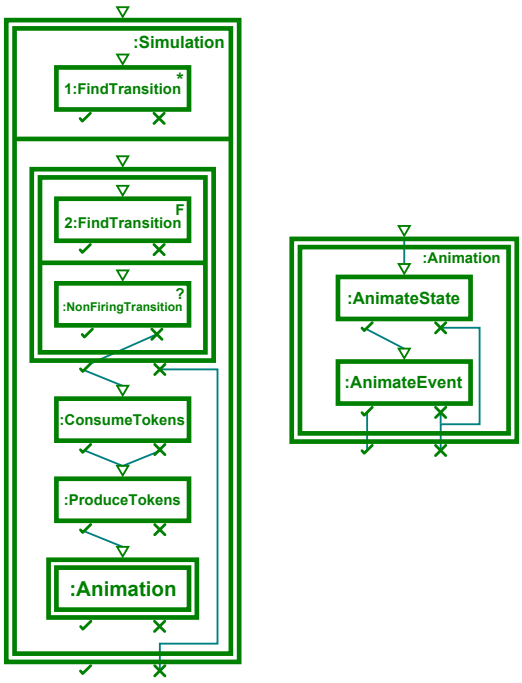


Figure 7.24: The final Petri net control flow.

have been possible to add animation using a multi-stage transformation approach as well (*i.e.*, sequential execution of transformations). The transformation we discuss next, however, cannot be easily expressed by a multi-stage approach.

7.6.4 Correspondence Links

The animation rules make use of correspondence links between finite state automata and Petri net models (see element labelled 4 of Figure 7.22). These correspondence links sometimes coincide with the intermediate generic links used in the "Finite State Automata to Petri nets" transformation, but the latter are not a reliable source for establishing correspondence. Furthermore, they are typically removed as part of the transformation in order not to waste memory.

We can, however, automate the insertion of relevant correspondence links by adding correspondence associations between certain language elements at the meta-model level, *i.e.*, by creating a "meta-triple" [GdL07b] (see Figure 7.25). With this additional information, a HOT that extends the "Finite State Automata to Petri nets" transformation with the feature of establishing correspondence links, can be defined with a simple, single rule (see Figure 7.26). This rule transforms translation rules such that they automatically insert corresponding links between respective input-output language element pairs. Note that Petri net places are linked to both state automaton states and events. It is therefore crucial to have the context of the original translation rule that creates an output language element based on the presence of an input language element. The rule in Figure 7.26 specifically matches such creation patterns. Establishing the correct links between corresponding input-output elements without

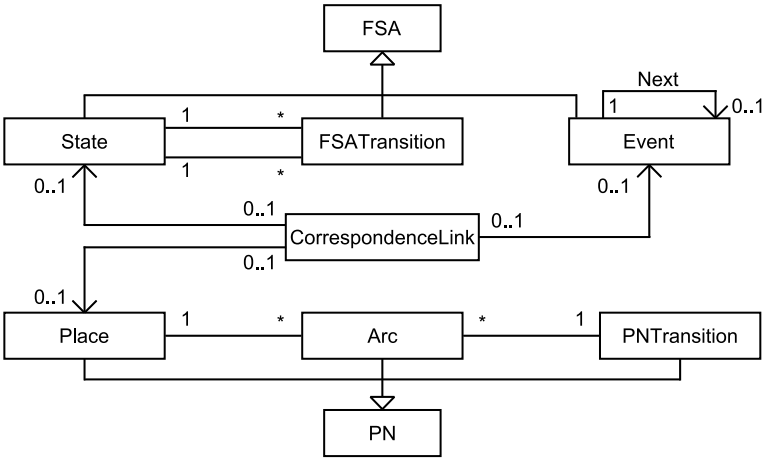


Figure 7.25: The correspondence meta-triple.

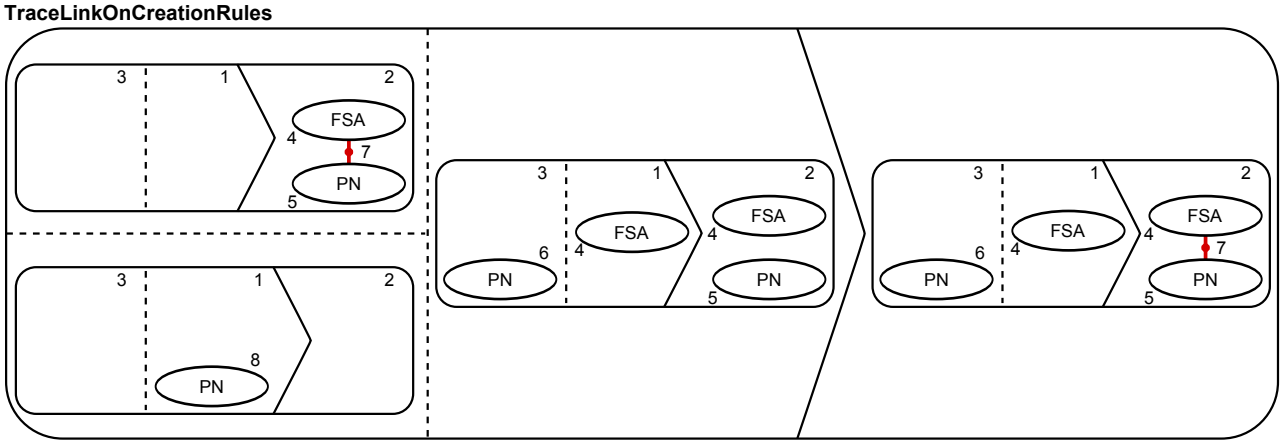


Figure 7.26: Higher-order Transformation: correspondence links.

this contextual knowledge would, in general, be impossible. This demonstrates that HOTs cannot be subsumed by multi-stage transformations.

Our correspondence HOT is not directly reusable because *MoTif* does not support parametrization of rules. Because of this limitation we could not formulate a generic version of the transformation that can be tailored to a particular application by passing in the names of meta-model elements of input/output language types. Instead, the pivot passing emulates parameter passing. Nevertheless, the transformation is reusable with respect to its structure. Another application can simply be obtained by a manual renaming of the input/output language types.

7.7 Related Work

This section summarizes some of the topics covered by *MoTif* and how they relate with existing approaches.

7.7.1 Explicit Use of Time

Timed Graph Transformation (TGT), as proposed by Gyapay, Heckel and Varró [GHV02], integrates time in the double push-out approach. They extend the definition of a production by introducing, in the model and rules, a "chronos" element that stores the notion of time. Rules can monotonically increase the time. *DEVS* is inherently a timed formalism, as explained previously. In contrast with TGT, it is the execution of a rule that can increase time and not the rule itself. That is, in TGT the designer of the rule can increment the chronos element in the RHS pattern. However, in *MoTif* (as a matter of fact in *MoTif-Core*) time is a property of a rule taken into consideration in the scheduling of the rules. Hence, the control flow (of the graph transformation) has full access to time.

This addition of time is similar to how Heckel et. al. define stochastic graph transformations [HLM04]. Every transformation rule is augmented by a probability indicating the delay of application of the rule. In fact, Heckel et. al.'s approach is complementary to ours: one can encode the application rate of individual *MoTif* rules in the time advance function, if the current simulation time is stored in the state of each rule. As pointed out in [GHV02], time can be used as a metric to express how many time units are consumed to execute a rule. Having time at the level of the block containing a rule rather that in the rule itself retains this expressiveness.

7.7.2 Higher-Order Transformation

Varró and Pataricza seem to have been the first to suggest higher-order transformations for improving the performance and maintainability of first-order transformations [VP03].

Schürr's triple graph grammars (TGGs) are designed to support correspondence links between models from declarative rules [Sch94]. In contrast to our higher-order transformations, a TGG transformation designer is more flexible in defining different correspondence links per rule. However, this also bears the risk that some correspondence links are forgotten or incorrectly established. A higher-order transformation like ours automates the process of establishing the required links and will be comparatively simple to correctly define, even for more advanced cases. Moreover, we may generate correspondence links with arbitrary amounts of additional information in contrast to the fixed format links of TGGs.

Jouault used *ATL* to define a higher-order transformation for automatically generating traceability links [Jou05]. Unlike our correspondence link higher-order transformation, however, Jouault's `TraceAdder` transformation adds *traceability* links between all elements rather than *correspondence* links. Traceability links can sometimes be used for tracking correspondences as well but not in general. Furthermore, while traceability links come for free as they do not need pattern specifications that only match relevant correspondences, they may use up a lot of memory even though the majority of

them is not being used in correspondence mapping applications. The main problem with using *ATL* for specifying higher-order transformations is that a higher-order *ATL* transformation has access to the transformation definition (*e.g.*, to the FSA-to-PN transformation), but not to the latter's respective input and output languages (the FSA and PN meta-models). All matching patterns and output patterns for the input/output languages of the transformation are therefore unchecked as they occur on a purely textual basis only. This results in even mundane syntactic errors going unnoticed. This problem is aggravated by the fact that the result of the higher-order transformation must be tested dynamically in order to detect the errors. Sometimes the only means to detect errors is to examine the first-order transformation's output. Having detected errors in the output, one then has to trace back the errors to the first-order transformation and from there back to the higher-order transformation. While there will always be a class of errors that will require this extended backward reasoning, our approach can avoid this complicated procedure for purely syntactical errors.

7.7.3 Comparison with other Model Transformation Approaches

The idea of defining the semantics of a model transformation language at a higher level of abstraction (such as *MoTif*) with respect to another model transformation language at a lower level of abstraction (such as *MoTif-Core*) is not new. For example, the semantics of *QVT-Relations* is defined in terms of *QVT-Core* [Obj08]. This semantic mapping is defined in plain English text, which lacks of formality. In the case of *MoTif*, the mapping is defined as a higher-order transformation which is, in turn, expressed in *MoTif*. Since *MoTif* is formally defined in terms of *DEVS*, *T-Core*, and graph transformation, the semantic mapping of *MoTif* to *MoTif-Core* is formally defined.

Nowadays, the Eclipse Modelling Framework (EMF) is gaining a lot of popularity. EMF allows one to design a transformation language by modelling its abstract and concrete syntaxes. *MoTif* is a completely modelled language. Its abstract syntax (defined by a meta-model) and its concrete (visual) syntax are specified in *AToM*^{3}. This can also be done in EMF. The semantics of *MoTif* is defined in terms of the DEVS formalism. The DEVS simulator ensures the execution of *MoTif* transformations. This would not be directly possible to implement in EMF, since this would require adding a virtual machine for simulating DEVS models on top E-Core virtual machine.

Finally, we complete the comparison of the different model transformation tools presented in Chapter 2.2. Recall that Table 2.1 categorized several tools according to different features. Table 7.1 completes this feature matrix for the *MoTif* language. This is based on the description of the language in Section 7.4.1. Although all the tools compared in Table 2.1 provide a control flow mechanism for graph transformations, many designed a new formalism for this purpose. Also, none of these exploit event-based transformations. *MoTif* not only allows event-triggered execution, but the user and his interaction with the executing transformation can be explicitly modelled, offering a user-centric approach to model transformations. On top of the novelties MoTif adds to control structures for model transformation, it is the only language introducing the notion of time and allowing the designer to *explicitly model* back-tracking and recursion.

Property	MoTif
Control Structure	DEVS
Atomicity	ARule
Sequencing	Yes
Branching	BRule
Looping	FRule, SRule, LRule
Non-determinism	BRule
Recursion	SRule, LSRule, BSRule
Parallelism	PRule
Back-tracking	XRule
Hierarchy	CRule

Table 7.1: Feature matrix of *MoTif*.

7.8 Conclusion

In this chapter, we have shown how the combination of *T-Core* with *DEVS* allows one to express a modular timed graph transformation language. This language, however, is typically not used directly by the modeller as it is a low-level transformation language. Nevertheless, *MoTif* offers a syntactic layer on top of *MoTif-Core* to abstract away from all *DEVS* artefacts. Therefore the semantics of *MoTif* is expressed in terms of *MoTif-Core*. This allowed us to enhance and introduce new model transformation features, such as the dimension of time. Being a completely modelled language, both at the syntax and at the semantics level, the *MoTif* language allows one to easily design higher-order transformations. In fact, the semantic mapping of *MoTif* to *MoTif-Core* is itself expressed in *MoTif* as higher-order transformation from the former to the latter.

Since expressiveness is the primary focus of this thesis, this chapter addressed the expressiveness of the novel language *MoTif*. From a performance point of view, each of the layers *MoTif* relies on is efficiently implemented: *T-Core*, *PythonDEVS*, *AToM*^{3}, and Python. In the future, we would like to investigate the performance overhead of having all the layers described in Figure 7.19 versus a more direct implementation such as *Py-T-Core*. One can foresee that the performance of *Py-T-Core* will be more efficient than that of *MoTif*. However, although *Py-T-Core* is more efficient and more easily integrable in programmed software, there are two disadvantages of choosing *Py-T-Core* over *MoTif*:

1. *Py-T-Core* does not rely on *DEVS* hence the modular execution property cannot be used anymore, *i.e.*, the execution cannot easily be ported from a sequential to a distributed environment.
2. In model-driven engineering terms, *MoTif* abides to the MPM principles and the transformation engineer or the domain-specific engineer can design transformations for his DSL at a level of abstraction as he used to model DSMs.

As mentioned in Section 7.4.2, an extension of *MoTif* is to provide full support for parallel and distributed transformations. One possibility is to enhance the language with the notion of scoping such

as in hierarchical graph transformations [DHP02, VJ04]. However, to facilitate the transformation engineer in the design of transformations using PRules that are currently supported, we have enhanced *MoTif* with the notion of exception handling, thus allowing for forward error recovery at the synchronization points.

8

Transformation Exceptions

As model transformations are increasingly used in model-driven engineering, the dependability of model transformation systems becomes crucial to model-driven development deliverables. As any other software, model transformations can contain design faults, be used in inappropriate ways, or may be affected by problems arising in the transformation execution environment at run-time. We propose in this chapter to introduce exception handling into model transformation languages to increase the dependability of transformations. We first introduce a classification of different kinds of exceptions that can occur in the context of model transformations. We present an approach in which exceptions are modelled in the transformation language and the transformation designer is given constructs to define exception handlers to recover from exceptional situations. This facilitates the debugging of transformations at design time. It also enables the design of fault-tolerant transformations that continue to work reliably even in the context of design faults, misuse, or faults in the execution environment.

8.1 Introduction

Model transformation is at the heart of model-driven engineering approaches; it is therefore crucial to ensure that the transformations are safe to use: when a model transformation is requested to execute, any exceptional situation that prevents the transformation from executing successfully must be detected and the requester must be made aware of the problem. Informing the requester about the situation allows for possible reactions. What exactly needs to be done highly depends on the context in which the model transformation has been requested.

A model transformation can be seen as an operation on models, taking a model as input and producing a (possibly implicit) model as output. This is similar to operations in a programming language, which can have input and output parameters and, in addition, can affect the application state stored in objects or variables. In order to address exceptional situations that prevent the normal execution of an operation, modern programming languages introduced *exception handling* [Don90].

A programming language or system with support for exception handling allows users to signal exceptions and to define handlers [Don90]. To signal an exception amounts to detecting the exceptional situation, interrupting the usual processing sequence, looking for a relevant handler, and then invoking it. Handlers are defined on (or attached to) entities, such as data structures or contexts for

one or several exceptions. Depending on the language, a context may be a program, a process, a procedure, a statement, an expression, etc. Handlers are invoked when an exception is signalled during the execution or the use of the associated or nested context. Exception *handling* means to put the system into a coherent state, *i.e.*, to carry out forward error recovery and then to take one of these steps: transfer control to the statement following the signalling one (resumption model [Goo75]); or discard the context between the signalling statement and the one to which the handler is attached (termination model [Goo75]); or signal a new exception to the enclosing context.

In model transformation, the transformation units (or *rules* in rule-based transformations) that compose a transformation have the notion of *applicability* (of a rule). In contrast to an operation at the programming language level, the model transformation may or may not be applied depending on the applicability of its constituting rules. We must from the beginning clearly distinguish *transformation failure* from *transformation inapplicability*, as we consider these as two distinct outcomes. In graph transformation for example, a rule *r* is said to be *applicable* if and only if an occurrence of its LHS is found in the model (encoded as a typed attributed graph). When *r* also specifies a NAC, such a pattern shall *not* be found given the LHS match. In case of a successful match, the match is replaced by the RHS of *r*. Thus the result of a *successfully applied rule* is the (possible) modification of the graph it received. If no occurrences of the LHS were found in the input model, the rule is said to be *inapplicable* and the resulting graph is identical to the input graph. Both a successfully applied rule and a rule that did not match (inapplicable) describe the regular execution of a transformation rule. However, as in the case of the execution of an operation in a program, it is possible that during a model transformation an exceptional situation is encountered in which case it is impossible to continue normal execution. At run-time, there are situations in which neither an output model can be produced by applying the transformation in its entirety nor is it possible to determine the non-applicability of the transformation. In this case the rule is said to have *failed*. The definition of applicability, inapplicability, and failure of rules can also be extended to the level of the transformation. That is respectively, the transformation has at least one rule that was successfully applied, no rule in the transformation has been applied, and the last rule to be applied has failed.

Once the development of the transformation is completed and tested, it is critical to provide a *failure-free* model transformation that does not crash when, for example, an erroneous input model is provided. The model can be incorrect from a syntactic or a semantic point of view. The former is typically detected before the transformation starts executing, since it must conform to a specific meta-model. Therefore we only consider semantic incorrectness of models in this paper. In this case, the designer must be able to specify how to handle such errors at the transformation level directly. This is why proper handling of exceptions must be available to the designer.

Currently, no model transformation language offers means to reason about such exceptional situations encountered during model transformations (see related work section). This chapter is based on [SKV10] which is a first attempt to motivate and define the notion of exception and exception handling in model transformations. Some may argue that it is not needed in a transformation language and that it is a tool or system issue instead. This contribution claims however that there are many kinds of exceptional situations that can arise while transforming a model and that, following MPM principles, these should be modelled in the transformation language to give the modeller control over

how such a situation is to be handled. If applied rigorously, exception handling leads to the design of safe model transformations.

The remainder of the paper is structured as follows. In Section 8.2, we analyse what kind of exceptions can occur in model transformations. Then, in Section 8.3, we elaborate on possibilities for handling such exceptions. We also outline the implementation of transformation exceptions and their handling in *MoTif*. Finally, we put the presented work in perspective in Section 8.4.

8.2 Classification of Exceptions in Model Transformations

Similar to exception class hierarchies used in object-oriented programming languages to distinguish between different kinds of exceptions, we propose a classification of the exceptions that may arise during a model transformation. In a transformation model, faults may originate from:

1. the transformation design,
2. the model on which a transformation is applied,
3. or the context in which the transformation is executed.

This section provides a non-exhaustive classification of potential exceptions that may arise during a transformation.

8.2.1 Terminology

We shall define the terms *failure*, *error*, and *fault* that are used in fault-tolerant computing, in the context of model transformation. A *failure* is an observable deviation from the specification of a transformation. In other words, a failed transformation either produced a result that, according to the specification, is not a valid output model for the specified input, or produced no result at all. An *error* is a part of the transformation state that leads to a failure. The transformation state includes the input and output models, as well as potentially created temporary models and auxiliary variables. A *fault* is a defect or flaw that affects the execution of the transformation. A fault is thus typically present before the transformation execution, *e.g.*, when there is a flaw in the design of the transformation, or the fault arises from the fact that a transformation is applied to a model that it was not designed to work on, or finally the fault resides in the execution environment. At run-time, a fault can be activated and lead to an error, *i.e.*, an erroneous state in the transformation, which in turn may be detected if the transformation language supports it. If it is not handled, however, an error propagates through the system until the transformation fails. Note that the time between the error activation and the transformation failure is the only time frame for the transformation to handle the exception.

We define an *exception* in the context of model transformation as a description of a situation that, if encountered, requires something exceptional to be done in order to resolve it. An *exception occurrence* at run-time signals that such an exception was encountered.

Traditionally, exception handling has two purposes. It can prevent non-intentional behaviour when a fault occurs or it can serve as an intentional means to interrupt the current execution and go out of

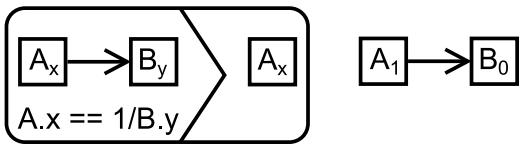


Figure 8.1: A rule with attribute constraints written in an action language (on the left) applied to a specific input model (on the right).

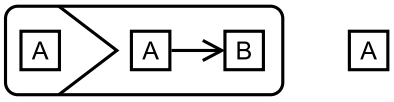


Figure 8.2: A monotonically increasing rule (on the left) applied to a specific input model (on the right).

scope. Therefore, from a design perspective, we distinguish between different uses of exceptions the model transformation designer can make.

8.2.2 Execution Environment Exceptions

`Execution Environment Exceptions` (EEEs) represent exceptional situations that typically originate from the transformation's virtual machine.

Action Language Exceptions

When the transformation language allows the use of an action language (which can contain a constraint language such as OCL), a complete exception tree may be provided for types of exceptions specific to the action language itself. Depending on the capabilities of the action language, these exceptions can come from arithmetic manipulations, list manipulations, de-referencing null references, etc. For example, Figure 8.1 illustrates a specific *`Action Language Exception`* (ALE), namely the case of a division by 0.

System Exceptions

During the execution of a transformation, the virtual machine executing the transformation can encounter exceptional situations, *e.g.*, it can run out of memory. There are many reasons that could lead to such a problem, one of which is a design fault in the transformation itself. Consider a transformation that contains an iteration over a monotonically increasing rule (that never deletes an element nor disables itself) as depicted in Figure 8.2. A memory overflow will eventually occur if an infinite loop or recursion (like a recursive rule as described in [GdL07a]) is executed.

Other kinds of *`System Exceptions`* (SEs) may arise, *e.g.*, I/O Exception when logging is used and the logging device is not writeable. Also, if the access to the model is provided via web-service functions, for example, the server may be down leading to communication or access errors.

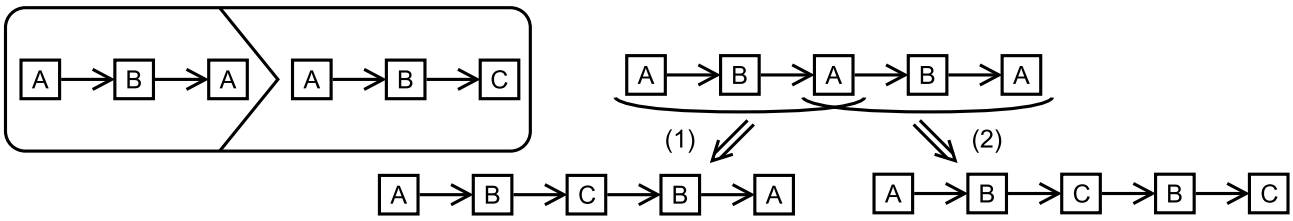


Figure 8.3: An inconsistent use of an *iterated rule* (on the left) with respect to a specific input model (on the right).

8.2.3 Transformation Language-Specific Exceptions (TLSEs)

Features specific to a particular transformation language can also be the source of exceptional situations. For example, *ProGReS* [ZS92], *QVT* [Obj08], and to a certain extent, *FUJABA* [FNTZ00] allow rules to be parametrized by specific model elements that may be bound to matches from previous rules. In these languages, executing a rule with unbound parameters results in an exceptional situation that needs to be resolved. A similar exceptional situation arises when a pivot node is passed from one rule to another by connecting input and output ports in *GreAT* [AKK+06] or even through nesting in *MoTif*, if the rules are not appropriately connected. In *MoTif* however, if a rule expects a pivot node that is not provided, the rule is inapplicable rather than in failure.

But there are other kinds of exceptional situations that can arise due to a specific transformation language design. In languages such as *QVT-R*, for example, the creation of duplicate elements is semantically avoided by the concept of *key* properties. Two elements are logically the same if and only if their key properties are equal. The *key* is used to locate a matched element of the model and a new element is created (with a new *key*) when a matched element does not exist. However, if multiple keys with the same value are found in a model, this indicates that the model is faulty[1].

Moreover, exceptions proper to the implementation of the scheduling language can also be considered. In *MoTif*, for instance, since the underlying execution engine allows for timed transformations by specifying the duration of application of a rule, bad timing synchronization may arise when *e.g.*, rules are evaluated at the same time (through conditional branching or parallel application). This typically happens due to the numerical error of floating point operations in the DEVS simulator.

8.2.4 Rule Design Exceptions

`Rule Design Exceptions` (RDEs) represent errors that stem from a fault in the design of the transformation model itself.

Inconsistent Use Exception

One class of design faults that may happen in a transformation is when rules are conflicting with one another. We distinguish the case when a rule conflicts with itself from when several rules conflict

[1] We assume in this chapter that the transformation engines are fault-free.

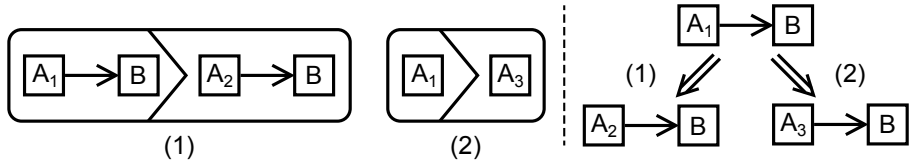


Figure 8.4: Two conflicting rules to be applied in parallel (on the left) with respect to a specific input model (on the right). The two rules are specified in a PRule depicting that they will be executed concurrently.

with each other. The former occurs when a rule finds multiple matches on a given input model and is executed several times in a row. This typically happens during an iteration; *e.g.*, a rule executed in a `loop` in *ProGReS*, iterated in a for-loop or a while-loop in *QVT-OM* [Obj08], or in case the rule is an SRule in *MoTif*. This is the case in the example of Figure 8.3, where the rule matches the input model twice, but depending on the order in which the matches are processed, two different output models are produced. Although the transformation itself is a valid transformation, the application of the transformation to this particular input model results in a non-deterministic result and as such is very likely to be incorrect. We consider such a situation as an *inconsistent use* of a transformation and propose that in these cases the transformation should be notified with an `Inconsistent Use Exception` (IUE).

Synchronization Exceptions

Another class of design fault can happen in the context of parallel execution of model transformations, a technique often used for efficiency reasons.

Semantically, if a transformation designer specifies that two rules should be executed in parallel, this implies that the order of execution of the transformation rules is irrelevant. This optimization can, however, only work if the two rules are independent from one another. In *MoTif*, FRule and the PRule both allow one to execute the rewriting part of rules concurrently. For example, the two rules in Figure 8.4 are clearly not independent, as the application of one disables the application of the other. In fact, executing both rules in parallel yields two different models that cannot be trivially merged without knowledge of the application domain. We propose to signal such situations by raising a `Synchronization Exception` (YE).

8.2.5 Transformation-Specific Exceptions

Finally, we believe that a dependable transformation language should also support user-defined exceptions. Almost all programming languages with support for exception handling support user-defined exceptions that allow the programmer to signal application-specific exceptional conditions to a calling context. Similarly, a transformation language that supports user-defined `Transformation-Specific Exceptions` (TSEs) makes it possible for the transformation engineer to check desired properties of the model being transformed at specific points during the transformation execution. These property checks can take the form of *assertions* as pre-/post-conditions on a (sub-)transformation by specifying

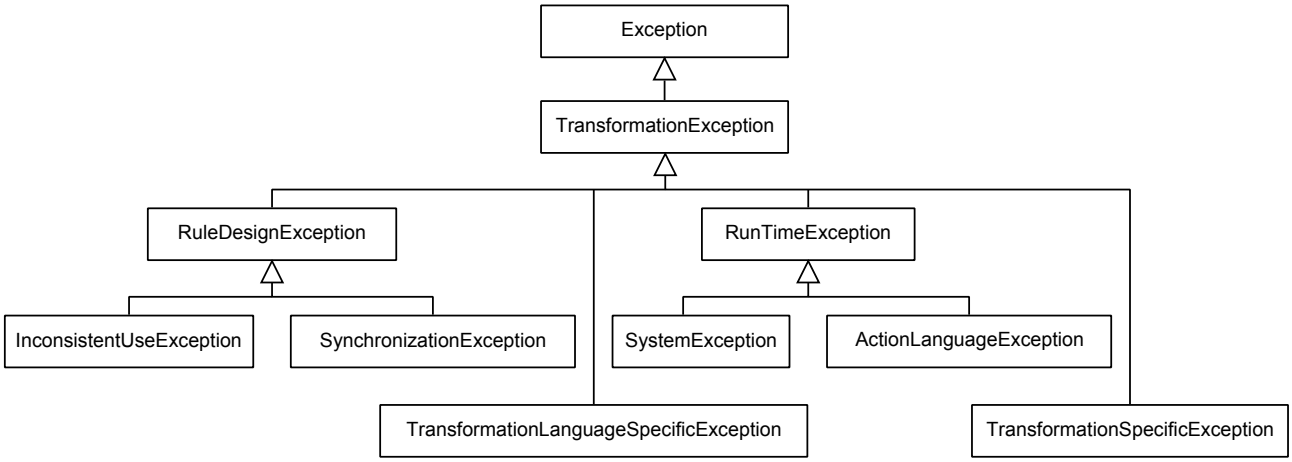


Figure 8.5: The proposed classification of model transformation exceptions in UML class diagrams.

a constraint on the current state of the model. In case the assertion fails, the corresponding TSE can be explicitly raised by the transformation model and signalled to the calling context.

8.2.6 Using Exceptions in Model Transformations

Figure 8.5 summarizes the classification of potential exceptions that may arise during the execution of a transformation. Some classes of exceptions like ALEs and TLSEs can be empty for certain model transformation environments, if the design of the transformation language and action language allows the corresponding problems to be detected statically. In the domain of programming languages, for example, dynamically typed languages such as Python define certain types of exceptions (*e.g.,* NoSuchField Exception) that strongly typed languages do not need to provide. In C++, for instance, a compiler can always statically determine that the programmer was erroneously trying to access a field of a class that has not been declared.

We foresee that exceptions are going to be used in two different ways in the context of model transformation: during transformation development to help eliminate design faults (*debug mode*) and when the transformation is applied to different models in order to increase dependability of the transformation at run-time (*release mode*). Some exceptions are more likely to occur in debug mode while others are relevant only in release mode.

Debug Mode

When running a transformation in debug mode, the goal of applying the transformation to an input model is not so much to obtain an output model that is subsequently used for other purposes, but to validate that the transformation design is correct. Debugging a transformation is not trivial and exceptions are very helpful for *debugging*, namely to detect logical errors in the design of a transformation.

If the generated output model does not correspond to what the transformation designer expects, then there must be a flaw in the transformation design that has to be found. In this case, the modeller

can debug the transformation by adding “assertion” rules at intermediate points in the transformation that check that the previous rule achieved the desired effect. If not, a user-defined TSE is thrown.

If unhandled, the exception halts the transformation execution and the transformation modelling environment informs the modeller of the exception kind and point of occurrence. Using this information, the modeller can more easily locate the rules that contain design faults.

When transformation rules are run distributed or in parallel to increase performance, a YE indicates a merging problem of the different output models. The problem occurs if the rules that are executed concurrently are not independent, *i.e.*, the intersection of the model elements modified by the rules is not empty. No transformation tool can provide an automated general merge operation, not only because general graph merging is undecidable, but also because the correct merging algorithm depends on the specifics of the transformation and its domain(s). Most likely a YE indicates that the modeller incorrectly assumed rule independence when he decided to instruct the transformation engine to use parallel execution.

The occurrence of an IUE on an input model, that the transformation under development should be able to handle, indicates that the iterated rule in which the exception was detected was incorrectly specified. The modeller needs to inspect the information carried with the exception such as the faulty matched model elements as well as the context of execution to then correct the faulty rule or revise the transformation design.

An EEE in debug mode can signal various problems to the modeller. It can signal design flaws, including flaws that are due to the incorrect use of a specific action language feature (*e.g.*, UnboundParameter Exception), incorrect expressions specified by the modeller using the action language (DivisionByZero Exception), or faulty transformation designs that result in infinite recursion or loops (MemoryOverflow Exception).

Release Mode

In release mode, a transformation that is assumed to work correctly is applied to an input model with the goal of producing an output model that is used for a specific purpose. Most likely it is essential that the transformation was applied successfully and did indeed produce the expected result; otherwise the output model is unusable. It is therefore important to design reliable transformations that can recover from exceptional situations and still provide a useful output.

In release mode, a SE such as `IOException` could signal that the device used for logging transformation related information is currently not writeable, for instance because the communication link broke down. Instead of immediately halting the transformation process, a reliable transformation could try to handle this situation. For instance, if the fault is assumed to be transient, the exception could be handled simply by waiting for some time and restarting the failed transformation. Alternatively, a different device could be used to store the log information.

The occurrence of an IUE in release mode signals that the transformation is being applied to an input model that the transformation was not designed to handle. An example of such a situation is given in Figure 8.3. This does not mean, however, that the rule cannot produce a correct output model. Both

possible outputs shown in Figure 8.3 might be correct, or maybe only one of them is. The problem is that the transformation system cannot guess what the correct behaviour of the transformation should be. One way of handling the exception could be to obtain *user (or external) input* from the transformation environment, *i.e.*, halt the transformation, prompt the user to designate the correct match or output, and continue with the transformation. Another transparent way of handling could be to apply a different set of rules instead that can produce an appropriate output model using different rules.

Finally, by incorporating exception handling into model transformation, it is possible to design fault-tolerant model transformations that can even tolerate transformation design faults. The idea is to employ design diversity in a way similar to what is done in recovery blocks [RX95] or N-version programming [CA78] at the programming language level: when a software is supposed to implement a critical functionality, several versions of the software are developed that achieve the specified functionality in different ways, *e.g.*, by using different algorithms and different data structures, sometimes even different programming languages and paradigms. The more diverse the implementations, the more likely it is that they do not fail simultaneously on the same input. At run-time, if one of the versions fails to deliver the expected result, the chances are high that one of the other versions produced a correct result. In the context of model transformation, it can be envisioned that a complex transformation can be designed in different ways by composing multiple rules in different ways, thus creating several versions of the same transformation. At run-time, if one of the transformation versions throws an exception, an alternate version can be executed to tolerate the fault and attempt to produce the desired result in a different way.

8.3 Exception Handling in Model Transformation

The previous classification identifies the exceptional situations that can occur during a model transformation. In order for a transformation to be dependable, the transformation designer should think of potential exceptions that could occur at run-time and design a way of addressing them in order to recover. We must therefore define a way that allows the transformation engineer to reason about exceptions and express exception handling behaviour at the same level of abstraction as the model transformation itself. However, exceptions should not be an escape to programming.

8.3.1 Modelling Exceptions

In order to be able to reason about exceptions at the transformation level, exceptions should be treated as first-class entities, *i.e.*, just like any other model element that can itself be used as an input to a transformation. From a transformation language design point of view, a transformation exception can be considered as a model conforming to a distinct meta-model as shown in Figure 8.6. An exception is identified by a name and has a status which can be: `active` (*i.e.*, the exception instance has not been addressed yet), `handling` (*i.e.*, it is currently being handled), or `handled` (*i.e.*, it has been addressed by a handler).

In order to enable proper handling, an exception must hold relevant information regarding its activation point context: where it happened, what happened, and when it happened. The transforma-

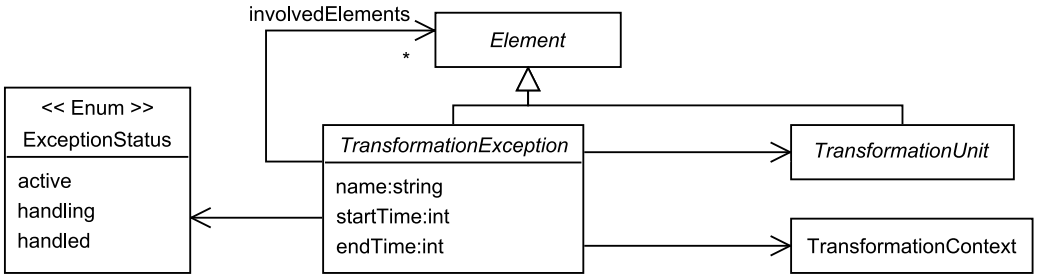


Figure 8.6: The transformation exception meta-model.

tion exception therefore references the transformation unit (the rule) that triggered its activation. The transformation context depicted in Figure 8.6 contains all the information needed to effectively investigate the origin of the exception occurrence and allow the designer to model an appropriate handler. In our implementation, for instance, the context contains the stack frame and the state of the packet (see Section 8.3.3) at the activation of the exception. In compositional or hierarchical transformation languages such as *MoTif*, *GReAT*, or *QVT*, knowing the exact path to the rule helps locating the fault in the transformation design, especially if the handler is not in the same scope as the activation point. To specify the activation point, several options can be considered:

- The most detailed information that can be provided is at the level of primitive transformation operations supported by the virtual machine instruction set (*e.g.*, CRUD operations). In this case, the modeller knows exactly which model element (node, edge, or attribute) was last accessed before the exception was activated. However, the transformation designer should not rely on the implementation of the transformation language when designing transformations.
- The context could simply indicate the transformation rule that triggered the exception. We believe that providing this little information severely limits the possible handling strategies that can be designed by the modeller, especially in the case of a SE or a TLSE.
- If the modelling language makes it possible to isolate the transformation operators (match, rewrite, iterate, etc.) from the virtual machine operations, such as in Chapter 3 (*i.e., T-Core*), then the activation point can be specified in terms of these operations. We believe that this is the right level of detail that allows the modeller to understand which part of the current rule generated the exception and to design an appropriate handler, if possible.

In addition to the point of activation, the transformation context should also indicate the state of the transformed (input) model at the time when the exception was thrown. For instance, in order to handle an RDE effectively, the input model elements involved in the matcher of the current rule should certainly be accessible to the handler.

In our proposed meta-model of an exception, we also included timing information, such as the timestamp at which the exception was generated (`active`) and has been handled (`handled`). This can be useful for profiling the transformation and gathering statistical measures on the handling policy. Moreover, in timed transformations such as in *MoTif*, the global (simulated) transformation time as well as the local time of the transformation rule operator may be useful.

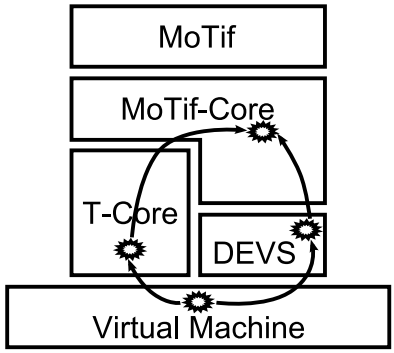


Figure 8.7: The *MoTif* framework and the propagation of exceptions across different layers.

8.3.2 Detection of Exceptions

When a transformation executes, the transformation run-time and the underlying virtual machine must monitor the transformation steps to detect the different kinds of exceptions presented in Section 8.2 and signal them appropriately. For example, the transformation run-time of the *MoTif* framework presented in Figure 7.19 is annotated with exception occurrences in Figure 8.7. The different classes of exceptions relevant to the modeller are detected at different layers, but must all be propagated to the *MoTif-Core* layer (and conceptually to the *MoTif* layer) in order to allow the modeller to handle them explicitly within the transformation, if unhandled in the meantime.

Detection of ALEs, such as *null de-reference* or *division by zero*, are typically detected at the level of the virtual machine in the *MoTif* framework. Depending on whether the action language is interpreted or compiled, certain design faults can even be detected at compile-time, in which case the corresponding exception never occurs at run-time. Similarly, the transformation language may prevent the action language from accessing model elements that are not explicitly part of the LHS, RHS, or NAC patterns, in which case null de-referencing can never occur. This may, however, be considered as a restriction on the expressiveness of the transformation language used and may lead to excessively large rules.

SEs are typically detected by the underlying operating system and the implementation language which is a Python interpreter in the *MoTif* case. To properly propagate the detected exception to the modeller, the exception needs to be caught at the virtual machine interface and transformed into the corresponding exception model instance shown in the previous section.

TLSEs are detectable at the level of *T-Core*, typically by checking pre-conditions before executing any language constructs. TLSEs are again an example of exceptions that can be rendered obsolete if the transformation language is compiled and strongly typed, in which case the compiler should be able to detect unbound parameters and similar situations. Bad timing synchronization of events can also be detected at the level of DEVS.

RDEs are also detected at the transformation language level. In algebraic graph transformation approaches, some RDEs can be detected statically. In grammar-like languages (a.k.a. unordered graph

transformation), rule non-confluence can be detected through critical pair analysis [HKT02]: verifying if a rule can disable another, *i.e.*, making it inapplicable. In such languages, this technique can assert parallel and sequential independence of the rules. Tools such as *AGG* detect these conflicts by overlapping the rules (all possible combination of the LHSs, taking NACs into consideration). However, their current approach is sometimes too conservative leading to false positives as it does not take into consideration the meta-model constraints (an example is given in [HHT02]). Moreover, although containing critical pairs of rules, a transformation may still be semantically correct and avoid the conflicts depending on the matches selected at runtime. The occurrence of an IUE can usually also not be checked statically since, most of the time, the input model to which a transformation is applied is not known at compile time.

Controlled graph transformation languages—which are more general than algebraic graph transformation approaches—consist of (partially) ordered rules, where rule scheduling is not implicit but modelled explicitly by the transformation designer. In this case, critical pair analysis is not directly applicable. It must first be adapted to controlled transformations as it may consider a pair of rules in conflict although the conflict does not occur at run-time because of a particular rule scheduling. For instance, let r_1, r_2, r_3 be a sequence of rules to be applied in this order such that the critical pair analysis test fails on (r_1, r_3) because r_1 deletes an element that can be matched by r_3. If r_2 re-creates those deleted elements, r_3 may still be applicable. In our framework, *T-Core* primitives such as a Resolver or a Synchronizer can be customized to detect IUEs and YEs.

Detection of TSEs cannot be done by the transformation framework automatically, since those situations depend on the semantics of the specific transformation. As mentioned in Section 8.2, they represent user-defined exceptions. Just like in programming languages that support user-defined exceptions where the programmer is responsible for detecting the exceptional situation using `if` statements or assertions, TSEs have to be detected by the transformation engineer. Fortunately, expressing a condition that needs to be satisfied by a model (or a condition that should never be satisfied by a model) is trivial in a transformation language: the condition can simply be expressed as a query on the input model, *e.g.*, using a QRule. Depending on the query, either a match being found or the fact that no matches are found depicts a violation of a constraint. To signal that, the rule must have the ability to `throw` an exception, which is described in the following subsection.

From a performance point of view, it is expected that a transformation running with exception detection mechanisms will be slower than one without it, as this is the case in programming languages such as Python, Java, and C#. Nevertheless, our prototype implementation showed that the closer an exception is detected to the virtual machine (see Figure 8.7), the smaller is the performance penalty. That is exceptions occurring at this level can be handled within the same layer or a layer above, whereas an exception occurring at the level of *MoTif* is needs to be processed by the underlying layers first before being rendered to the user at the transformation model level.

8.3.3 Extending Rules with Exceptions

In order to allow rules to signal exceptions to the enclosing transformation, we propose to add exceptional outcomes to rules. Therefore, such an *exceptional rule* receives a model as input and has

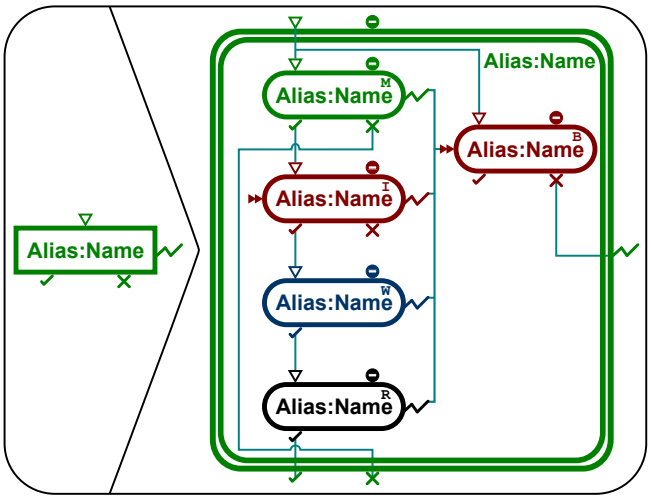


Figure 8.8: An ARule with the ability to detect exceptions.

three possible outcomes: a successfully transformed model (in case of a successful match and execution of the transformation), an unmodified model (in case the rule is inapplicable on the model), or an exception (when an exceptional situation occurred). If the rule outputs an exception, there are two possibilities: either (1) if the error took place in the matching phase, then the input model is not modified, or (2) if the error took place during the application phase of the rule, then the input model may have been partially modified. The latter outcome seems to defeat the *atomicity* property of a rule. However, as expressed in [GGKH03], a partial output may sometimes be desirable. This feature is certainly very helpful in debugging mode, as the modeller would like to see a partial result even if not complete to understand at what point the transformation execution failed in terms of the input model. Nevertheless, if backward recovery is desired, the transformation language should offer a mechanism to roll-back to the previous "safe" state of the model, *i.e.*, to the state that was valid before the rule was applied.

We have integrated the notion of exception in *MoTif* following the ideas mentioned above. At the level of *MoTif*, an Exception port can be added to a rule block. It is visualized by a zigzag line as shown in Figure 8.8. Informally, the semantics of an ARule with an exception port is that if the rule is applicable, the transformed graph is output through the Success port. Otherwise if the rule is inapplicable, the input graph is output through the Fail port. However if an error occurs, a transformation exception is output through Exception port at any point during the execution of the ARule. Therefore a rule has three possible outcomes: a new graph when the transformation is successful, the unmodified graph when the rule is not applicable, or an exception (modelled as in Section 8.3.1) if an exceptional situation is detected. In the latter case, the packet may have been partially modified. On the one hand in the matching phase, information concerning the matched elements may have already been stored in the packet. On the other hand, if the rewriting phase was already initiated, the input model may have been modified. In any case, the packet is in an *inconsistent state* with respect to the atomicity property of graph transformation rules application. The remaining rule blocks are adapted in a similar way.

To formally define this behaviour, we must adapt the semantic mapping of *MoTif* to *MoTif-Core* to take into account exceptional rule blocks. Being an event-based system, we model exceptions in *MoTif-Core* as events since they allow interruption. This was already present in its meta-model in Figure 7.1. Furthermore, the mapping of *MoTif-Core* elements onto *DEVS* also considered the output of exceptions in the definition of the output set Y and output function λ. An exception can be output by any atomic primitive. In the scope of a composite primitive, the Rollbacker receives an exception from its ANextIn inport and forwards the output of the TCRollbacker (its *T-Core* counter part) to the Composer as depicted in Figure 8.8. In this case, the TCRollbacker should output the same exception it received with the original packet stored in it. To ensure that an exception is received from the CExceptionOut outport of the Composer, the transfer function $Z_{i,C}$ from Section 7.2.3 verifies that the event is of type `TransformationException`.

From an implementation point of view, exceptions can be caught either directly at the level of an atomic DEVS (*e.g.,* for simulation errors) or at level of *T-Core* (*e.g.,* SE or algorithmic errors). In the latter case, the exception is propagated to level of the atomic DEVS. The generated exception is an instance of the class diagram in Figure 8.6 with the appropriate type according to Figure 8.5. It is initialized with a descriptive name, the time it occurred[2], and with the active status. The transformation unit is refers to the enclosing *MoTif-Core* atomic primitive, identified by its alias, name, and type. The `involvedElements` refer to the elements in the `match2rewrite` of the `current` condition pattern in the packet.

8.3.4 Modelling the Handler

Unlike current transformation tools such as *ATL* or *FUJABA* where exception handling is available only at the level of the code of the implementation of the transformation language, we believe again that the most appropriate level of abstraction at which exception handling behaviour should be expressed is at the transformation language level. This is similar to what is done in programming languages, *e.g.,* in Python, where exception constructs are provided in Python and not in C (the language in which Python is implemented). This could be achieved by specifying exception handlers either (1) at the level of transformation rules (in which case an exception handler would take the form of a transformation rule that is only applied in exceptional situations), (2) at the level of the transformation operators (*e.g.,* at the level of *T-Core* primitives such as the matcher or the rewriter), or (3) at the level of the primitive model manipulation implementation provided by the virtual machine level (CRUD). For the same reasons that we detailed in Section 8.3.1, we consider specifying exception handlers at the same level of abstraction as the transformation rules themselves as the optimal choice.

We propose two alternatives for how transformation exception handlers can be specified in a model transformation language. From a pure model transformation point of view, an exception can be seen as an ordinary model, although its semantics distinguishes it from a “normal” model. Hence when a rule emits an exception (model), this model can serve as input for other rules whose pre-condition looks for a specific exception type. Given an appropriate meta-model for modelling an exception (such as

[2]At the DEVS level, exceptions are caught in the external transition function δ_{ext}. The start time of a transformation exception is computed by taking the sum of the current simulation time with the elapsed time e.

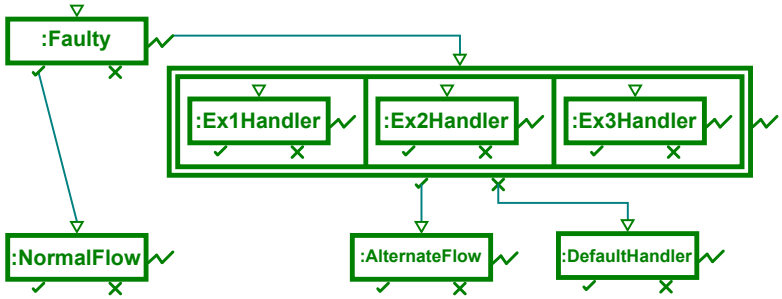


Figure 8.9: Explicitly modelling exception handling in the transformation model.

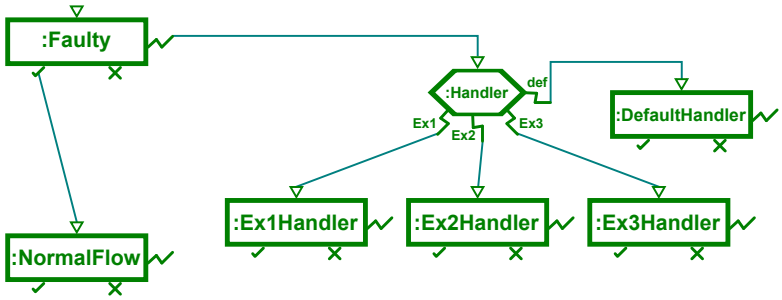


Figure 8.10: Handling exceptions in the transformation model.

partly given in Figure 8.6), the meta-model of the LHS pattern of an exception handling rule (*i.e.*, its domain) would need to involve multiple meta-models: the RAMified meta-model of the input model to transform as well as the meta-model of the transformation exception. This can be easily accomplished if the transformation language is itself meta-modelled as in the *MoTif* framework. Figure 8.9 shows an example of this approach based on the *MoTif* language. In a "normal" execution scenario, the rule `Faulty` is applied first, and in case of success, the rule `NormalFlow` is subsequently applied. However, if an exception occurs in `Faulty`, the rule outputs the corresponding transformation exception to the enclosing transformation. Then, the BRule containing the three rules `Ex1Handler`, `Ex2Handler`, and `Ex3Handler` receives the exception. The behaviour of this composite rule block is that the three rules simultaneously receive the exception and try to find a match. But at most one of the applicable rules is finally applied. Then, the transformation continues with the application of the rule `AlternateFlow`. Notice here that there is no priority precedence on the handling depending on the exception type. It is up to the modeller of the transformation to specify the order in which the rules catch the exception. For example, the rule `DefaultHandler` is only applied if none of the rules `Ex1Handler`, `Ex2Handler`, and `Ex3Handler` match.

Although the solution presented is elegant, our experience showed that it is not very practical to let the modeller specify rules that match elements from the transformation domain and simultaneously from the domain of exceptions. A more pragmatic solution is to let the exception produced by a rule be used to influence the control flow of the transformation, redirecting it to rules designed for handling specific exceptions. To support this, *MoTif* introduces an explicit Handler block where the modeller

may access the information of the exception model and specify subsequent rules to pursue the exception flow. The handler block acts as a dispatcher sending the packet contained in the transformation context of the exception through the outport corresponding to the exception name. Figure 8.10 shows the use of the handler block. The handler block associates the packet with the appropriate exception outport given a predefined exception tree. Since it is possible that not all the exceptions that can occur during a model transformation execution have a specific handler rule designed to address them, a default exception port is provided with the handler block (which is linked to the top-most exception class in the classification tree presented in Figure 8.5, *e.g.*, `TransformationException`).

Note that in both models the handling part may itself produce an exception which can be in turn handled. For example, since the handling process involves further pattern matching, memory overflows are likely to occur and hence it is necessary to properly handle such exceptions.

Formally, the *MoTif* Handler is mapped to an atomic DEVS, integrated in *MoTif-Core*[3]. Following the notation used in Section 7.2.2, a handler parametrized by a name and an alias, well as a set of strings Ψ each representing a pre-defined exception type. At initialization-time, the atomic DEVS will have as many outports as the size of Ψ. A Handler is defined by the following structure:

$$Handler_{name,alias,\Psi} = \langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

The state S is defined as:

$$S = \{(\chi, map) \,|\, \chi \in \texttt{INSTANCESOF}(\text{Exception}) \cup \{\texttt{nil}\}, map : \Psi \rightarrow Y\}$$

In this notation, map is a bijective function that maps an outport for each string in Ψ. Also, the Handler temporarily holds a an exception, from the time it receives it to the time it outputs it. We denote by $s_0 = (\texttt{nil}, map)$ the initial value of S. It receives an exception as input, hence the input set is:

$$X = \{\texttt{INSTANCESOF}(\text{Exception})\} \cup \{\phi\}$$

It outputs a packet, from one of the $|\Psi|$ outports, hence the output set is:

$$Y = \{\texttt{INSTANCESOF}(\text{Packet})\}^{|\Psi|} \cup \{\phi\}$$

The Handler does not consume time, hence the time advance is:

$$\tau(s) = 0$$

The internal transition function simply removes the exception stored in the atomic DEVS. Thus:

$$\delta_{int}(s) = s_0$$

The external transition function simply stores the exception in the atomic DEVS. Thus:

$$\delta_{ext}((s, e), \chi) = (\chi', map)$$

[3]Recall that a property of *MoTif-Core* is that a model can be extended with custom DEVS models (c.f., Section 7.3).

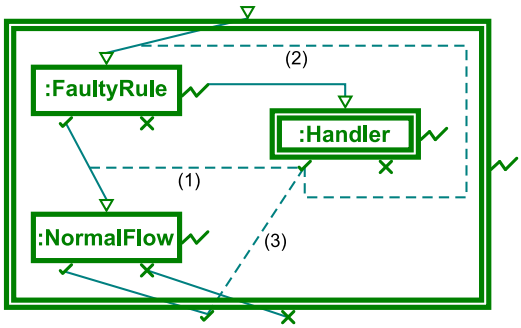


Figure 8.11: Modelling possible control flows after handling an exception locally at a sub-transformation level: (1) resume after the activation point, (2) restart at the beginning of the enclosing context, and (3) terminate the enclosing context.

where χ' is the updated version of the exception with the status set to `handling`.

Finally, for the output function, the Handler outputs the exception stored in its state via the outport in $Y = (y_1, y_2, \ldots, y_{|\Psi|})$ corresponding to the type of the exception. Hence:

$$\lambda((\chi, map)) = \begin{cases} \chi'' & \text{if } map(\chi.\texttt{type}) = y_i \\ \phi & \text{otherwise} \end{cases}, \forall y_i \in Y$$

where χ'' is the updated version of the exception with the status set to `handled`.

8.3.5 Control Flow Concerns

Once an exception is generated, it should either be handled right away or propagated to a higher scope. When a rule emits an exception, the control flow is redirected to a handler component which, in release mode at least, handles the exception with the goal of continuing the transformation. After the exception is handled, there are three options: the enclosing transformation may *resume*, *restart*, or *terminate*.

Resuming the transformation means to return the flow of control to the place where it was interrupted by the exception. As depicted by channel (1) in Figure 8.11, the transformation continues in the "normal" flow *after* the rule that activated the exception. Such a resumption model allows one to express an alternative execution of the transformation. However, care should be taken if the input model (or even the packet) was modified. As outlined in Section 8.3.3, the modeller may choose to recover the model to a state that was valid before the rule started applying its modifications, if desired.

Restarting the transformation means to re-run the enclosing transformation from the beginning. As depicted by channel (2) in Figure 8.11, the transformation restarts the "normal" flow *before* the rule that activated the exception. This is certainly an interesting way to tolerate transient faults. However, restarting the transformation induces a loop in the control flow which may lead to dead-locks, in case the fault is of a permanent nature.

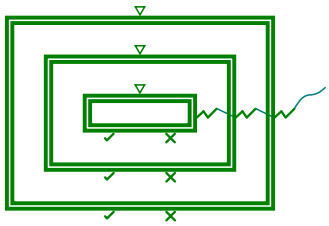


Figure 8.12: Propagating the exception to enclosing contexts.

Terminating the transformation means to skip the entire flow of the transformation in which a rule raised the exception. As depicted by channel (3) in Figure 8.11, the enclosing transformation exits the scope of the occurrence of the exception seamlessly. It ends in a "normal" flow, *i.e.*, in success or not applied mode, and not in generating an exception. As a result, the outer scope is not aware that an exception occurred.

Exception Propagation

Up to now, we have considered that it is at the level of a transformation rule that an exception is generated and subsequently handled at the level of the enclosing transformation. As mentioned previously, *MoTif* transformation models are hierarchical, in the sense that transformations can be nested. Constructs such as a CRule modularly define scoped sub-transformations, allowing to compose transformations models.

Just like in programming languages, it is recommended to handle exceptions in a scope that is as close as possible to the point of activation. In other words, local exception handling is preferred. However, it is possible that handling an exception locally is not possible, because the necessary context information that is needed to define a useful handler is only available in a more global context. Therefore, unhandled exceptions must be propagated up the transformation hierarchy as long as no corresponding handler can be found. Only if an exception propagates unhandled out of the topmost CRule, the transformation execution must be halted and debugging information displayed.

Figure 8.12 illustrates how, if an exception occurs, it is propagated[4] through three transformation encapsulation units (CRules). Nevertheless, at each level, a handler could be specified to "clean up" any local state before the exception is propagated outside. Note that once an exception is handled, it can no longer be propagated. If propagation is needed, the handler must create a new user-defined exception which can refer to the previously handled exception in its `TransformationContext`.

8.4 Related work

The concept of exceptions exists in programming languages since the 1970s [Goo75]. Several approaches have been proposed for modelling exceptions in workflow languages [BCCT05] and event-

[4]Note that in *MoTif*, even if the exception ports of sub-models are not explicitly connected to the exception port of their enclosing block, the generated *MoTif-Core* model will have such links.

driven languages [PM05]. However, there has not been any work on modelling exceptions in model transformation languages. Current tools mostly rely on exceptions triggered from the underlying virtual machine. As a matter of fact, debugging in tools such as ATL [JABK08], Fujaba [FNTZ00], GReAT [AKK+06], QVTo [Dvo08], SmartQVT [Fra08], or VIATRA [VB07] is specific to their respective integrated development editors (IDE).

Exception handling provides dynamic forward error recovery. Back-tracking mechanisms such as in *ProGReS* [Zün92] are considered as backward recovery mechanisms, although not intended for error recovery but rather for searching for matches to apply a rule.

QVT Operational Mappings (*QVT-OM*) supports exception handling at the action language level, an imperative extension of OCL 2.0 [Obj08]. The language allows one to handle exceptions in a *try ... except* statement in the same way as in modern programming languages such as Java, It is however unclear where, when, or how an exception occurs in *QVT-OM*. User-defined exceptions can be declared and raised arbitrarily[5] in the *main* operation of a transformation. Moreover, an exception is also raised when a *fatal assertion* is not satisfied. However, it is unclear what information exceptions carry and whether they can be propagated outside the scope of the transformation. Implementations of *QVT-OM* such as SmartQVT and QVTo (an Eclipse plug-in into EMF) have different interpretations from the standard, *e.g.*, allowing *map* to raise an exception if the pre-condition is not fulfilled. The advantage of our approach is to (1) explicitly model the raising of exceptions and (2) explicitly model the control flow subsequent to the handling of an exception.

FUJABA is a model transformation tool based on graph transformation combined with Story diagrams [FNTZ00]. There, exceptions are also not modelled, although present at the code level. The *maybe* statements in Story diagrams can be used to handle exceptions in the transformation, but they are only available for statement activities (*i.e.*, Java code). The same argument can be used as for the choice of allowing exception handling at the level of *T-Core*.

8.5 Conclusion

In this chapter, we have motivated the need for providing the concepts of exception and exception handling at the level of transformations. We have outlined a classification of potential exceptions that can occur in the context of model transformation. Though having different uses at different steps of the development of a transformation model, these transformation exception must be handled by the transformation model itself. We have discussed the different issues related to the handling of these exceptions.

We have implemented the main concepts of this approach in *MoTif*. As the prototype is still in an early stage, we are working on a system which will allow for user friendly debugging of model transformation. The implementation of the detection mechanisms for some classes of exceptions (such as SE, ALE, and TSLE) relies on the exception detection mechanism of the underlying implementation language (*i.e.*, Python). A complete implementation is left for future work.

[5] This fits in the *Action Language Exceptions* category according to our classification.

The same exercise this paper presented for the *MoTif* framework can be done for the *ATL* or *QVT-OM* languages. We are confident that it is also applicable to transformation languages at different levels of abstraction such as relational transformations (*e.g.*, *QVT-R* or *Triple Graph Grammars*).

Exception handling can become handy when designing a higher-order transformation. For example in *ATL* [KMS+09], static verification of well-formed higher-order transformation rules is quite limited. In this case, with an exception handling mechanism at the transformation level, the designer may safely rely on the engine to design arbitrarily complex higher-order transformations.

Part IV

Applications

"In theory, there is no difference between theory and practice. In practice there is."

Jan L. A. van de Snepscheut

9

The Pacman Game Case-Study

In Part III, we introduced the novel model transformation language *MoTif*. It allows one to model a time-advance for every rule as well as to interrupt (pre-empt) rule execution. In this chapter, we design a case study to show how the explicit notion of time allows for the simulation-based design of reactive systems such as modern computer games. We use the well-known game of Pacman as an example and model its dynamics in *MoTif*. This also allows the modelling of player behaviour, incorporating data about human players' behaviour and reaction times. Thus, a model of both player and game is obtained which can be used to evaluate, through simulation, the playability of a game design.

9.1 Introduction

Modelling and Simulation-Based Design MSBD uses mathematical, visual and executable methods to address problems of specification, verification, validation, integration and deployment of designing complex hardware and software systems. This includes techniques of abstraction, compositionality and virtualization. The *Discrete Event system Specification* (*DEVS*) formalism [Zei84] is primarily intended for modelling and simulation of complex systems.

MoTif is a model transformation language whose semantics is based on *DEVS*. Since *DEVS* inherently allows one to build hierarchical models, the transformation language becomes highly modular, allowing re-use of specific components of a transformation. Another side-effect of using *DEVS* is the explicit notion of time it introduces in model transformations. This allows one to model a time-advance for every rule as well as to interrupt (pre-empt) rule execution. Thanks to this notion of time, we implement a simulation-based design of reactive systems such as modern computer games. More precisely, the dynamics of the game is entirely modelled in the model transformation language.

In this chapter, we illustrate the application of *MoTif* to modelling and simulation-based design by means of the well-known Pacman game example, presented in Section 9.2. Subsequently, the suitability of *MoTif* for Modelling and Simulation-Based Design is demonstrated. The Pacman game is entirely modelled in *MoTif* in Section 9.3, including the game semantics, the transformation environment as well as the player. The simulation and optimization experiments as well as the synthesis of a real-time Pacman web application are explained in Section 9.4.

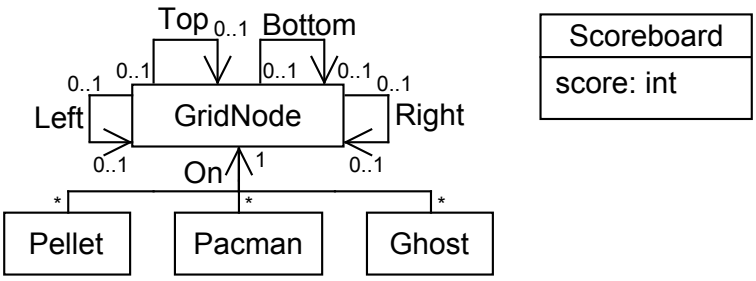


Figure 9.1: The Pacman Meta-Model

9.2 The Pacman Formalism

In order to illustrate the power of *MoTif* in the context of simulation-based design, we use a simplified version of the well-known video game Pacman throughout this chapter. The *Pacman* language syntax and semantics are inspired by Heckel's tutorial introduction to graph transformation [Hec06]. In what follows, we first synthesize a Pacman-specific visual modelling environment in the tool *AToM*^{3} [dLV02] by defining a meta-model of the *Pacman* language. Subsequently, we model the semantics of the Pacman language by means of graph transformation rules.

9.2.1 The Pacman Language (Abstract and Concrete Syntax)

The *Pacman* language has five distinct syntactic elements: Pacman, Ghost, Pellet, GridNode, and ScoreBoard. Figure 9.1 shows the meta-model of this modelling language. Pacman, Ghost, and Pellet objects can be linked to GridNode objects with an On association. This represents that these objects can be "on" a grid node. The four self-associations Left, Right, Top and Bottom between GridNode objects represent the geometric organization of the game area, similar to the classic Pacman video game. At a semantic level, these associations denote that Pacman and Ghost "may move" to a connected GridNode. A Scoreboard object holds an integer valued attribute `score`. For the concrete syntax of the *Pacman* language, an icon is associated with each of the meta-model's classes. For each of the meta-model's associations, a geometric/topological constraint relation is given. The On association between Pacman and Ghost entities for example is rendered as the Pacman icon being centered over the grid node icon[1]. Note how in this example there are no restrictions on the number of instances of each meta-model class, nor on the number of links to a GridNode instance.

9.2.2 The Pacman Semantics (Graph Transformation)

The operational semantics of the *Pacman* formalism is defined by means of a collection of (graph) transformation rules. These rules take as input a host graph (model) and produce as output the transformed graph. The resulting graph may be only partly modified, *e.g.*, the GridNode elements are preserved in the transformation. Concrete visual syntax is used in the rules in Figure 9.2(a)-(d). This feature of *AToM*^{3} is particularly useful for domain-specific modelling. The `kill` rule in Figure 9.2(a)

[1]Note that this is why links (instances of association) are not shown explicitly in Figures 9.2(a)-(d).

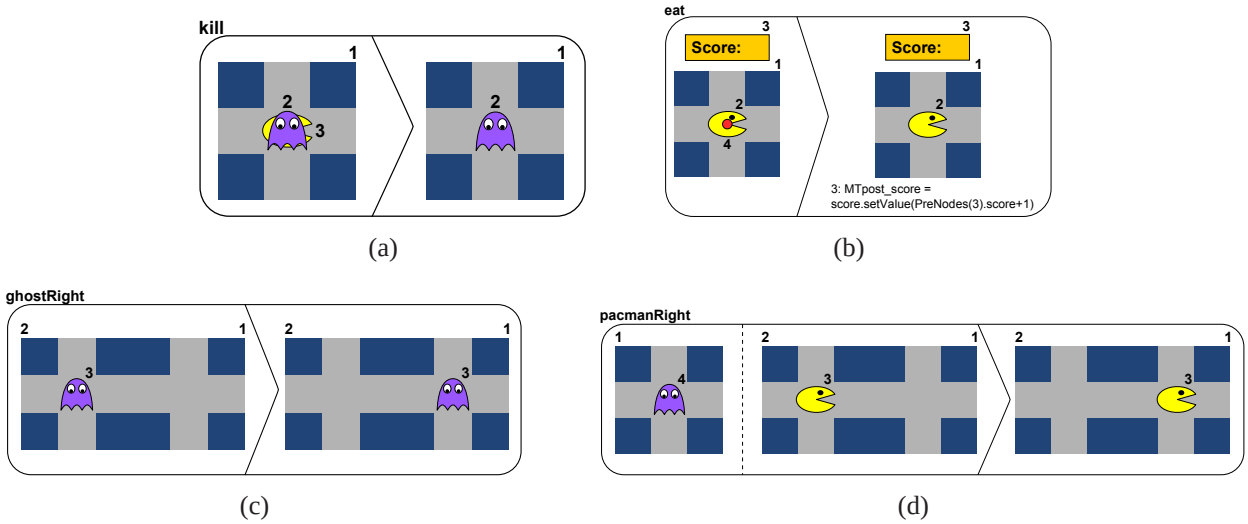


Figure 9.2: The Pacman semantics rules for (a) Ghost killing Pacman, (b) Pacman eating Pellet, (c) Ghost moving right, and (d) Pacman moving right.

shows killing: when a Ghost object is on a GridNode which has a Pacman object, that Pacman object is deleted. The `eat` rule in Figure 9.2(b) shows eating: when a Pacman object is on a GridNode which has a Pellet object, the Pellet object is deleted and the score gets updated (using an attribute update expression). The `ghostRight` rule in Figure 9.2(c) expresses the movement of a Ghost object to the right and the `PacmanRight` rule in Figure 9.2(d) the movement of a Pacman object to the right. Note the presence of a NAC in the last rule prevents the Pacman from moving to a GridNode that holds a Ghost (as this would imply certain death). Similar rules to move Ghost and Pacman objects up, down, and left are omitted.

9.3 Modelling the Pacman Case Study

At the heart of our approach lies the embedding of graphs in DEVS events and of individual transformation rules into atomic DEVS blocks as, at run-time, a *MoTif* transformation model is a *DEVS* model.

9.3.1 (Modelling) The Transformation Environment

The overall transformation model of the Pacman game is shown in Figure 9.3. The atomic DEVS block User is responsible for user (player) interventions. It can send the initial graph to be transformed, the number of rewriting steps to be performed (possibly infinite), and some control information. In a previous work [SV07], the control information was in the form of key press codes to model the user input to a game. All these events are received by the Controller, another atomic DEVS block. This block encapsulates the coordination logic between the external input and the transformation model. It sends the host graph through its outport to a rule set (the `Automatic` CRule) until the desired number

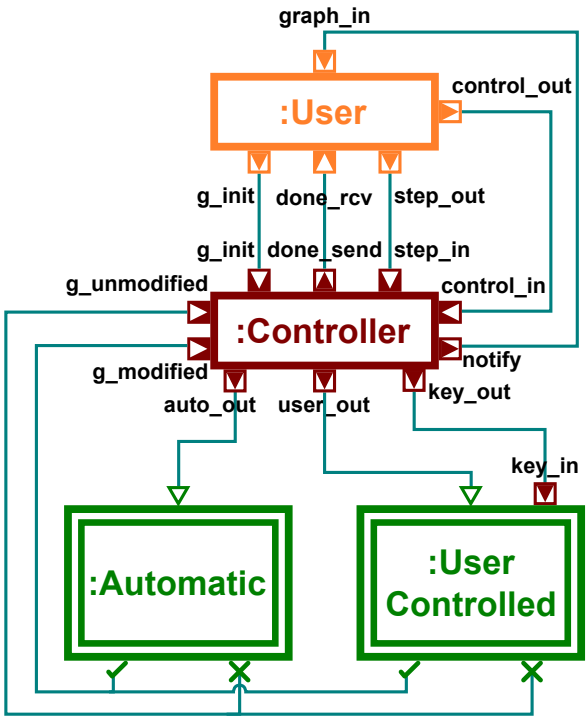


Figure 9.3: The overall transformation model

of steps is reached. If a control event is received however, the Controller sends the graph to another rule set (the `UserControlled` CRule). The RuleAutomatic CRule expects only a graph to perform the rewriting on, whereas the `UserControlled` CRule waits for a control, too. The details are omitted here to focus on the overall structure.

The model described in [SV07] does not model a realistic, playable game. When the user sends a key, the corresponding transformation rule is executed and the graph is sent to `Automatic` until another key is received or the Pacman entity has been deleted (note that the consumption of transformation steps is not a termination factor when infinity is initially sent). What prohibits this from being suitable for a playable game is:

- A rule consumes a fixed amount of time. From the graph rewriting perspective, this allows one to compute the time complexity of the transformation. From the input model perspective, it gives a way of quantifying the complexity of a model. However, this does not take into consideration any notion of game levels or real-time behaviour which a game such as Pacman should have.
- The user sends information to the rewriting system to (1) configure the transformation engine and (2) to control the transformation execution abstracted to the specific domain of interest (Pacman movements). This model does not take into account any playability issues, such as the Ghost moving too fast versus a user reacting too slowly.

In the sequel we present an extended model with focus on timing information. This will allow us, through simulation, to construct an optimally “playable” game.

9.3.2 Modelling the Player

In current graph transformation tools, the *interaction* between the user –the player in the current context– and the transformation engine is hard-coded rather than explicitly modelled. Examples of typical interaction events are requests to step through a transformation, run to completion, interrupt an ongoing transformation, or change parameters of the transformation. In the context of the Pacman game, typical examples are game-events such as Pacman move commands. Also, if animation of a transformation is supported, the time-delay between the display of subsequent transformation steps is encoded in the rewriting engine.

In contrast, in our *DEVS*-based approach, the interaction between the user and the game is explicitly modelled and encapsulated in the atomic DEVS block User (see Figure 9.3). Note that in this interaction model, time is explicitly present. Also performance analysis in the form of statistics is often neglected.

With the current setup, it is impossible to evaluate the *quality* (playability) of a particular game dynamics model without actually *interactively* playing the game. This is time-consuming and reproducibility of experiments is hard to achieve. To support automatic evaluation of playability, possibly for *different types* of players/users, it is desirable to explicitly model player behaviour. With such a model, a complete game between a modelled player and a modelled Pacman game –an experiment– can be run *autonomously*. Varying either player parameters (modelling different types of users) or Pacman game parameters (modelling for example different intelligence levels or speed in the behaviour of Ghosts) becomes straightforward and alternatives can easily be compared with respect to playability. For the purpose of the Pacman game, player behaviour parameters can be user reaction speed or levels of decision analysis (such as pathfinding). We have explored these two dimensions of behaviour. Section 9.4 will discuss reaction speeds more in-depth.

Obviously, evaluating quality (playability) will require a precise definition of a playability *performance metric*. Also, necessary data to calculate performance metrics needs to be automatically collected during experiments. Explicitly modelling player behaviour can be done without modifying the overall model described previously thanks to the modularity of *DEVS*. We simply need to replace the User block by a coupled DEVS block with the same ports as shown in Figure 9.4. We would like to cleanly separate the way a player interrupts autonomous game dynamics (*i.e.,* Ghost moving) on the one hand and the player's decision making on the other hand. To make this separation clear, we refine the User block into two sub-models: the User Interaction and the UserBehaviour atomic DEVS blocks. On the one hand, the User Interaction model is responsible for sending control information such as the number of transformation steps to perform next, or a direction key to move the Pacman. On the other hand, the UserBehaviour block models the actual behaviour of the player. This is often referred to as the "AI" of a Non Player Character (NPC) in the game community. It is this block which, after every transformation step, receives the new game state graph, analyzes it, and outputs a decision determining what the next game action (such as Pacman move up) will be. Also, since it is the UserInteraction block which keeps receiving the game state graph, we chose to give this block the responsibility of sending the initial host graph to the transformation subsystem.

An atomic DEVS block, Dispatch, receives the user action and branches the execution to the corre-

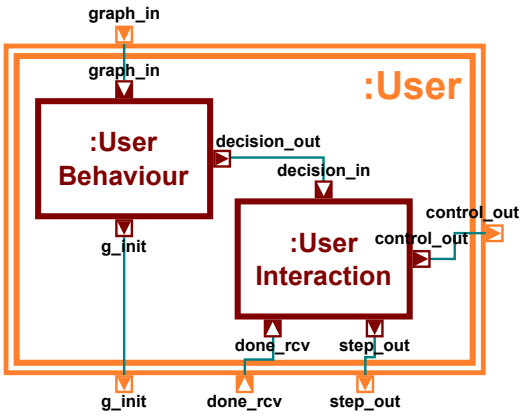


Figure 9.4: The enhanced User model

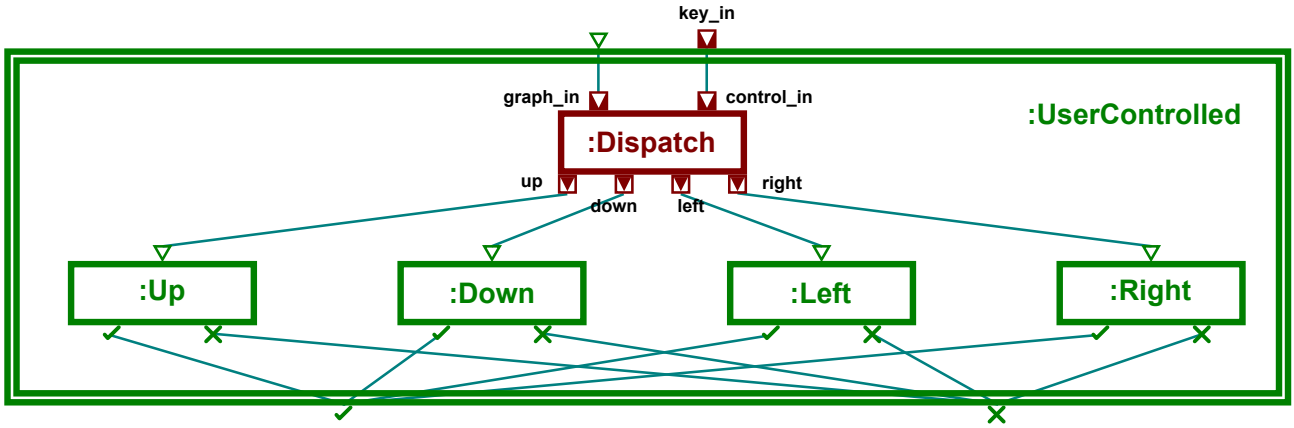


Figure 9.5: The UserControlled model

sponding rule to be triggered. Thus, in this approach, the event-driven execution of the transformation is embedded in the *transformation model* rather than in the rule itself such as in [GdL07b, GdL04]. There, the notion of Event-driven Graph Rewriting was proposed in the context of visual modelling: a graph rewriting rule is triggered in response to a user action. More precisely, the rule itself is augmented to behave according to the event it received. In *MoTif*, we have separated the event reaction from the rule itself. When the UserBehaviour coupled DEVS block emits the event, it is fed to the `UserControlled` model via the Controller as shown in detail in Figure 9.5. Also, in our approach, the user and user interaction itself have been modelled in the User coupled DEVS block.

Different players may use different *strategies*. Each strategy leads to a different model in the UserBehaviour block. We have modelled three types of players for our experiments: Random, Lazy, and Smart.

The Random user does not take the current game state graph into consideration but rather chooses the direction in which the Pacman will move in randomly. Note that this type of player may send direction keys requesting illegal Pacman moves such as crossing a boundary (wall). This

is taken care of by our Pacman behaviour rules: the particular rule that gets triggered by that key will not find a match in the graph, hence Pacman will not move. However, time is progressing and if Pacman does not move, the ghost will get closer to it which will eventually lead to Pacman death. Note that the rules used are similar to the move right of Figure 9.2(d): the NAC prevents movements towards a grid node already occupied by a Ghost.

The Lazy user does not make such mistakes. After querying the game state graph for the Pacman position, it moves to the adjacent grid node that has Food but not a Ghost on it. If no such adjacent grid node can be found, it randomly chooses a legal direction.

The Smart user is an improved version of the Lazy user. Whereas the Lazy user is restricted to making decisions based only on adjacent grid nodes, the Smart user has a "global" view of the board. The strategy is to compute the closest grid node with Food on it and move the Pacman towards it depending on the position of the Ghost. One way to implement this strategy is by using a path finding algorithm. Many solutions exist for such problems, including some efficient ones such as A* [HNR68]. It is possible to integrate this kind of path finding techniques in a *MoTif* model. Because it is not the main focus of this paper, we only outline two possibilities. One is to explicitly model the path finding with transformation entities. The depth-first search model in Figure 9.6 can be a starting point. The behaviour of the `SmartMove` transformation model is to find a sequences of path steps that, from a Pacman object, lead to a Pellet object. Since the `TryMove` BRule consists of transactional SRules, if `MakeMove` succeeds, `TryMove` is recursively applied on the new model. However, if `Eat` fails, `TryMove` rolls-back to the state before it was last applied and tries a different match or branch. It therefore models a recursive backtracking algorithm to find a correct path. Another abstraction is algorithmic, using an atomic DEVS block to encode the algorithm in its external transition function. In any case, whether modelled declaratively using backtracking rules or algorithmically, the path finding sub-model can be used as a "black-box" and integrated transparently in the transformation model. In this case-study, extending the *Pacman* meta-model with (x,y) coordinates on grid nodes allows a linear time solution to this particular path finding problem. *AToM*^{3} allows one to add actions to meta-model elements. Relative coordinates management is handled in the action of each of the four associations between GridNode objects: if a GridNode instance g_1 is associated with another instance g_2 by a Left association, then $g_1.x < g_2.x$ and $g_1.y = g_2.y$. Similar conditions are defined for Right, Top, and Bottom associations. Therefore, the pathfinding only needs to compute the shortest Manhattan distance from Pacman to Pellet as well as perform a simple check for the grid node coordinates of the Ghost.

We compare the performance of different user behaviour types in Section 9.4. Note that to match different user types, we need to model similar strategies for the Ghost to make the game fair. Indeed, a Smart user (controlling the Pacman) playing against a randomly moving Ghost will not be interesting nor will a Lazy user playing against a Smart Ghost. As players may become better at a game over time, game *levels* are introduced whereby the game adapts to the player's aptitude. This obviously increases game playability.

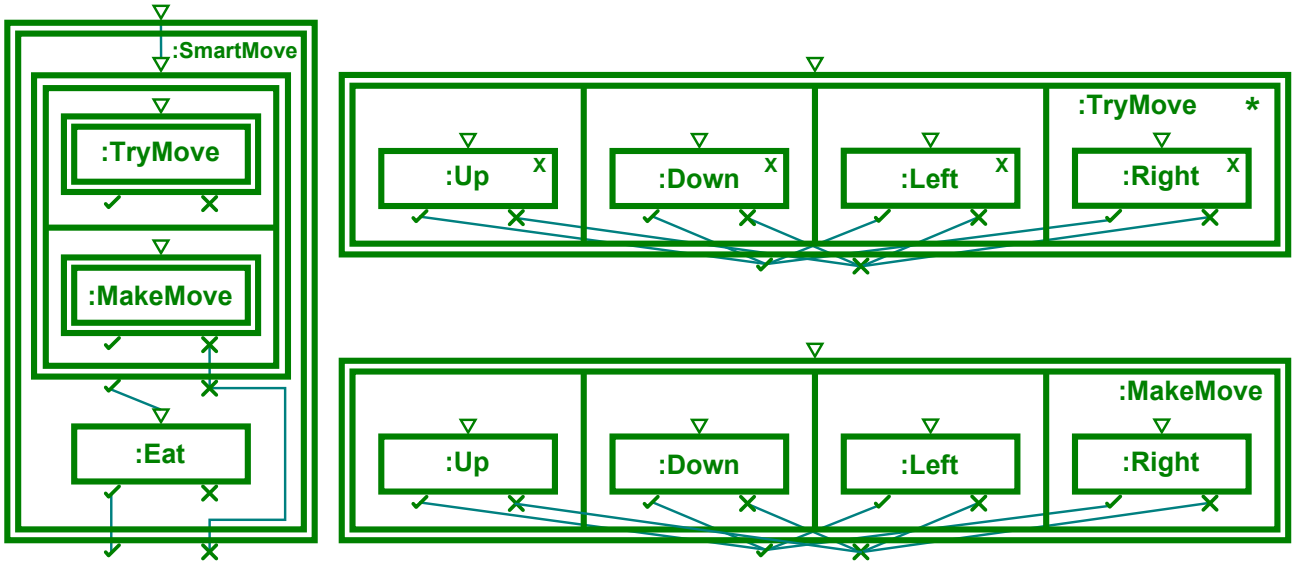


Figure 9.6: The *MoTif* model performing a recursively back-tracking transformation. The `SmartMove` CRule encapsulates the sequence of an XLSRule followed by the `Eat` ARule. `TryMove` encapsulates the XBSRule sub-model. `MakeMove` is the BRule version of `TryMove`.

9.3.3 Modelling the Game

As long as the (modelled) player does not send a decision key to move the Pacman (thus changing the game state graph) the game state graph continues to loop between the Controller block and the `Automatic` block depicted in Figure 9.7. If no instantaneous rule (`Kill` or `Eat`) matches, then it is the lower priority `GhostMove` block that modifies the graph. The Ghost movement model in Figure 9.8 supports different strategies. The game state graph is received by a Decider atomic DEVS block. Similar to the UserBehaviour block, it emits a direction that drives the movement of the Ghost. The Random, Lazy and Smart strategies are analogous to those of the player. The Random Ghost will randomly choose a direction, the Lazy Ghost will look for a Pacman among the grid nodes adjacent to the

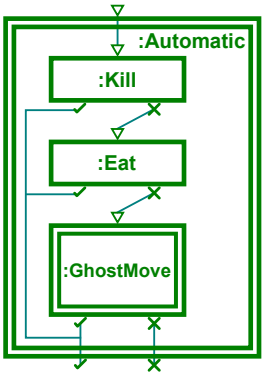


Figure 9.7: The `Automatic` CRule encoding rule priorities.

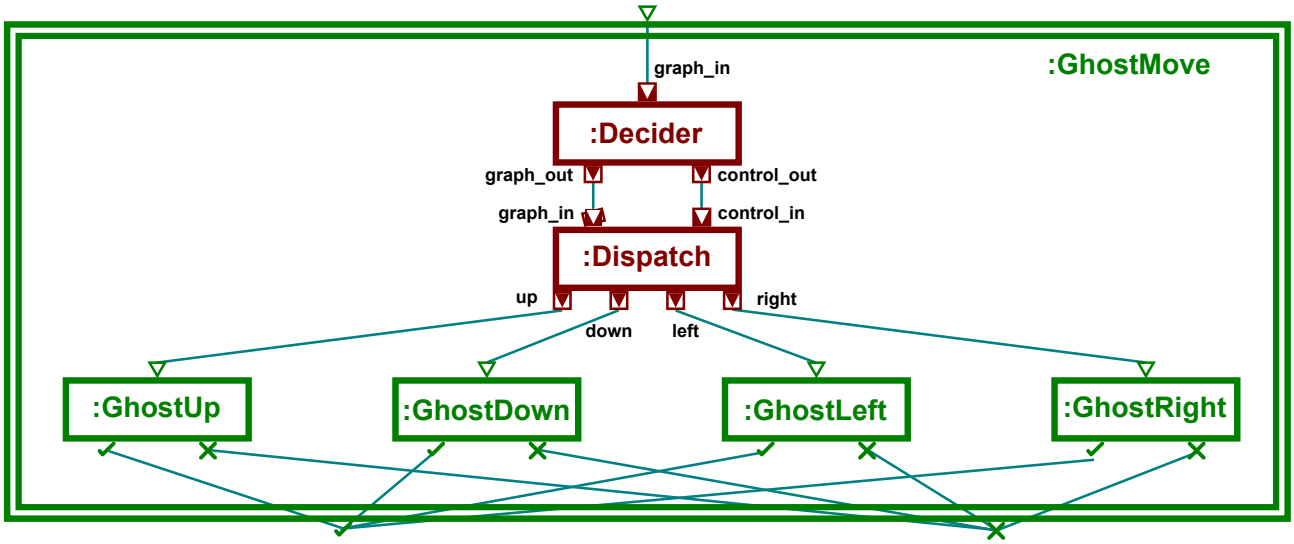


Figure 9.8: The enhanced `GhostMove` Model

one the Ghost is on, and the Smart Ghost has "global" vision and always decides to move towards the Pacman[2]. The same argument previously made about optimal path finding and backtracking applies. The Decider sends the graph and the decision (in the form of a key) to a Dispatch block and the rest of the behaviour is identical to that in the `UserControlled` CRule.

9.3.4 Explicit Use of Time

We have now modelled both game and player, and the behaviour of both can use Random, Lazy, or Smart strategies. However, one crucial aspect has been omitted up to now: the notion of time. Time is critical for this case study since game playability depends heavily on the relative speed of player (controlling the Pacman) and game (Ghost). The speed is determined by both decision (thinking) and reaction (observation and keypress) times.

We will now show how the notion of time from the *DEVS* formalism integrated in a graph transformation system can be used for realistic modelling of both player and game. We consider a game to be unplayable if the user consistently either wins or loses. The main parameter we have control over during the design of a Pacman game is the speed of the Ghost.

Each atomic DEVS block has a state-dependent time advance that determines how long the block stays in a particular state. `Kill` and `Eat` rules should happen instantaneously, thus their time advance is 0 whenever they receive a graph. In fact, all rules involved in the transformation have time advance 0. What consumes time is the decision making of both the player (deciding where to move the Pacman) and the game (deciding where to move the Ghost). For this reason, only the Decider and the UserBehaviour blocks have strictly positive time advance.

[2]In the original Pacman video game, these different Ghost types are referred to as "Clyde" for Random, "Pinky" for Lazy, and "Inky" for Smart.

To provide a consistent playing experience, the time for the Ghost to make a decision should remain almost identical across multiple game plays. The player's decision time may vary from one game to another and even within the same game. We have chosen a time advance for the Decider that is sampled from a uniform distribution with a small variance (interval radius of 5ms). What remains is to determine a reasonable average of the distribution. To make the game playable, this average should not differ significantly from the player's reaction time. If they are too far apart, a player will consistently lose or win making the game uninteresting.

9.4 Simulation experiments

In the previous section, we determined that the playability of the Pacman game depends on the right choice of the average time advance of the Decider block, *i.e.,* the response time of the Ghost. We will now perform multiple simulation experiments, each with a different average time advance of the Decider block. For each of the experiments, a playability performance metric (based on the outcome and duration of a game) will be calculated. The value of the Decider block's average time advance which maximizes this playability performance metric will be the one retained for game deployment. Obviously, the optimal results will depend on the type of player.

9.4.1 Modelling User Reaction Time

First of all, a model for player reaction time is needed. Different psychophysiology controlled experiments [ZS02] give human reaction times:

- the time of simple visuomotor reaction induced by the presentation of various geometrical figures on a monitor screen with a dark background;
- the time of reaction induced by the onset of movement of a white point along one of eight directions on a monitor screen with a dark background.

The reaction time distribution can be described by an asymmetric normal-like distribution. The cumulative distribution function[3] for sensorimotor human reaction time is:

$$F(x) = e^{-e^{\frac{b-x}{a}}} \tag{9.1}$$

where a characterizes data scatter relative to the attention stability of the subject: the larger a is, the more attentive the subject; b characterizes the reaction speed of the subject. For simulation purposes, sampling from such a distribution is done by using the Inverse Cumulative Method [Dev86]: a value sampled from the initial distribution function is computed by sampling a random number from a uniform distribution in the interval $[0,1]$ and subsequently evaluating the inverse cumulative function at that value.

[3] The type of the function for describing the reaction time distribution was chosen on the basis of the results obtained in two adult subjects (400 reaction time values of each type were measured in the dominant hand). The function was tested on a group of 25 university students between 17 to 20 years of age.

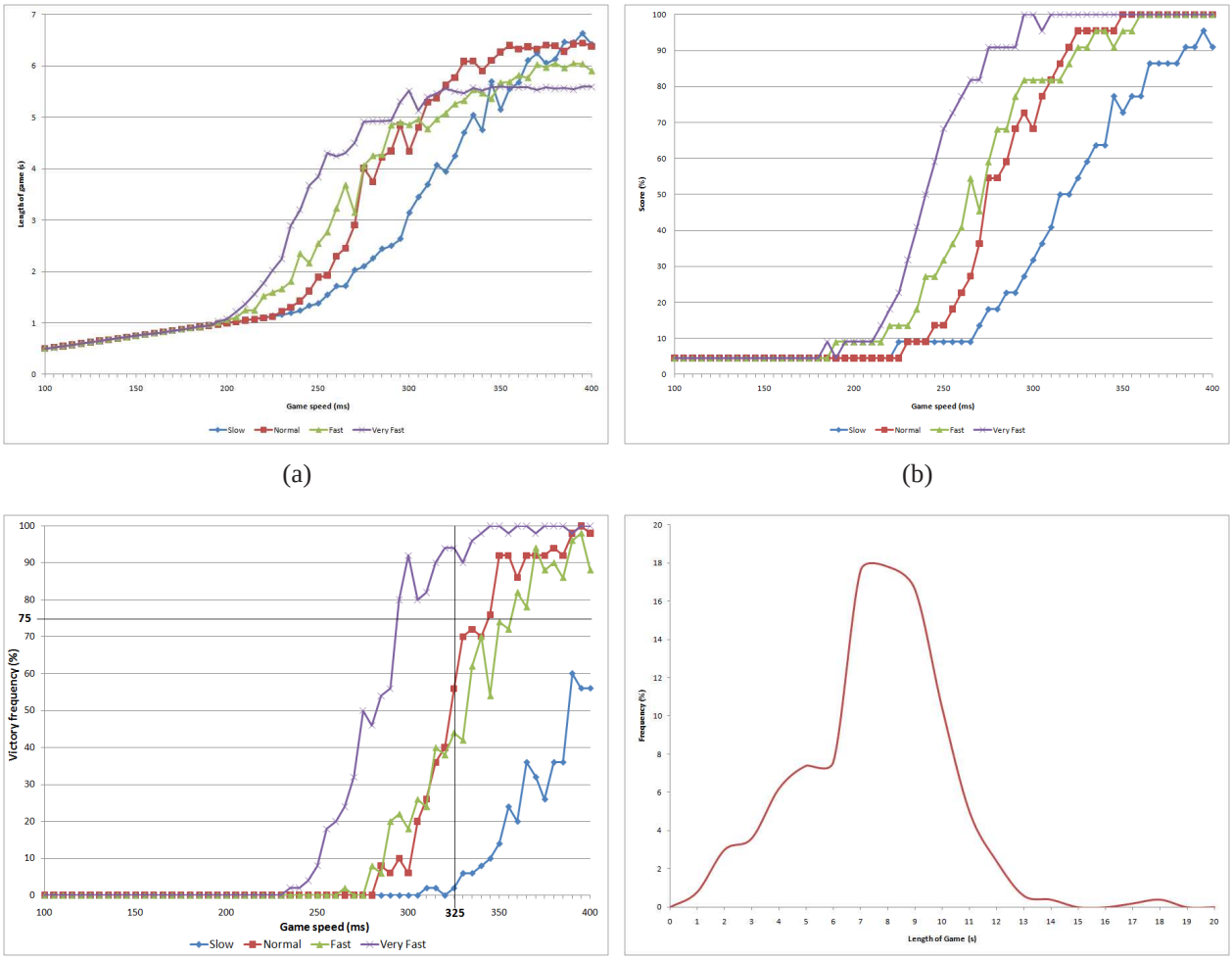


Figure 9.9: The simulation results: (a) the time till end, (b) the score at the end of game, (c) the victory frequency, and (d) the game length distribution with a normal user and a game time advance of 325ms.

For our simulation, four types of users were tested: Slow with $a = 33.3$ and $b = 284$, Normal with $a = 19.9$ and $b = 257$, Fast with $a = 28.4$ and $b = 237$, VeryFast with $a = 17.7$ and $b = 222$. The parameters used are those of four example subjects in [ZS02].

9.4.2 Simulation Results

For the simulations, we only consider the Smart user strategy. For each type of user (Slow, Normal, Fast and VeryFast), the *length of the simulated game* is measured: the time until Pacman is killed (loss) or no Pellet is left on the board (victory). To appreciate the need for these results, the score is also measured for each run. Simulations were run for a game configuration with 24 GridNodes, 22 Pellets, 1 Ghost and 1 Pacman. The game speed (ghost decision time) was varied from 100ms to 400ms. Each value is the result of an average over 100 samples simulated with different seeds.

The following presents the simulation results obtained by means of the DEVS simulations of our game and player model. All figures show results for the four types of users (Slow, Normal, Fast and VeryFast). Figure 9.9(a) shows the *time until the game ends* as a function of the time spent on the Ghost's decision. The increasing shape of the curves imply that the slower the ghost, the longer the game lasts. This is because the user has more time to move the Pacman away from the Ghost. One should note that after a certain limit (about 310ms for the VeryFast user and 350ms for the Normal user), the curves tend to plateau. To further investigate this behaviour, Figure 9.9(b) shows the score (relative percentage of number of Pellets eaten) for the different game speeds. Not surprisingly, these curves and the previous ones have the same shape: increasing up to a certain limit and then remain constant. These limits even coincide at sensibly the same values and happen when the score reaches 100%. An explanation for this behaviour is simply that after a certain point, the Ghost decision time is too low and the user always wins. Therefore, the optimal average *time advance* value we are looking for is found in the middle of the steep slope of the plots.

Figure 9.9(c) depicts the *frequency* with which a player will *win* a game (when playing a large number of games) as a function of the time spent on the Ghost's decision. We decided that we want to deploy a game where the user should be able to win with a probability of 75%. Thus, the optimal average Ghost time advance (decision time) was found to be 325ms (taking into account fairness among the different types of users). Note that the focus of this case study is not on the reasons for such decisions/assumptions but rather on how the integration of graph rewriting systems with the DEVS formalism gives the modeller the right level of abstraction for simulation and performance analysis.

To give further insight in the variability of the game experience, Figure 9.9(d) shows the *game length distribution* at the optimal *time advance* value. It is an unimodal distribution with a peak at 7.5s. This average is quite low, but not surprising given the small game board. Experience with the deployed real-time game application is consistent with this value.

9.4.3 Game Deployment

Having found a prediction for the optimal time the Decider block should spend on the choice of the next movement of the ghost entity, we can now test the simulated game with real users, in real-time. From the *AToM*^{3} *Pacman* formalism/meta-model we have synthesized –yet another model transformation– an Ajax/SVG-based application. The web application's dynamics is defined by a JavaScript compiled Statechart handling the events. The *MoTif* model is slightly modified to get the decision event from an external source. The User coupled DEVS now has an inport *"interrupt"* waiting for external input. The User Interaction block is then linked to that port to receive the decisions. The player behaviour model in the User Behaviour block is thus discarded. The transformation model is executed by a real-time version of our `pythonDEVS` simulator. An intermediate layer for event management and communication between the web client and the transformation model is used.

The most expensive operation in a graph transformation tool is the matching phase. *MoTif*'s current implementation of the rule matching turned out to be significantly fast enough to play the deployed game. Figure 9.10 shows a snapshot of the deployed game with the same initial state and conditions

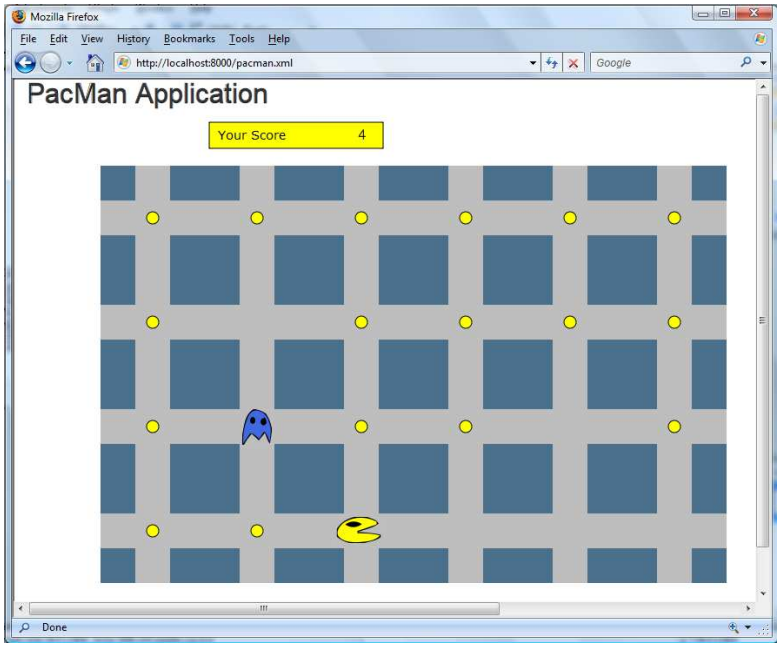


Figure 9.10: Snapshot of the deployed Pacman game running in a web browser

as the simulated game.

9.5 Conclusion

In this chapter we have shown how the explicit notion of time in *MoTif* allows for the simulation-based design of reactive systems such as modern computer games. We used the well-known game of Pacman as an example and modelled its dynamics with programmed graph transformation based on *DEVS*. This allowed the modelling of player behaviour, incorporating data about human players' behaviour and reaction times. We used the models of both player and game to evaluate, through simulation, the playability of a game design. In particular, we proposed a playability performance metric and varied parameters of the Pacman game. This led to an "optimal" (from a playability point of view) game configuration. The user model was subsequently replaced by a web-based visual interface to a real player and the game model was executed using a real-time DEVS simulator.

The use of graph transformation at the heart of this approach allows non-software-experts to specify all aspects of the design in an intuitive fashion. The resulting simulations give quantitative insight into optimal parameter choices. This is an example of modelling and simulation-based design, where the graph transformation rules and the timed transformation system are modelled, as well as the user (player) and the context. Having modelled all these aspects in the same model transformation framework, *MoTif*, allows for simulation-based design.

The transformation language used in the Pacman example emulates *AToM*^{3}'s rewriting semantics. In fact, we could have used another graph transformation semantics (such as unordered or layered

graph rewriting). We could even have combined different transformation specification languages. As such, *DEVS* acts as a "glue" language.

Using the *DEVS* formalism as a control flow language for graph rewriting enabled us not only to model the *AToM*^{3} semantics for graph transformation execution but also to model continuous execution and user interaction. Note that we are thus modelling control structures supporting step by step simulation, continuous simulation and user controlled simulation, which are not in the system under study, but rather in the execution environment.

The beauty of *DEVS* models lies in the modularity of its building blocks. In fact, each block performs an action given some input and can produce outputs. This modularity trivially supports the combination of building blocks specified using *heterogenous models* expressed in multiple formalisms. Hence, we may combine graph grammars with for example Statecharts and code. This is the key to scaling up (graph) transformation modelling to arbitrarily more complex models, far beyond the limits of pure rule-based graph transformation systems.

For future work we propose the following. The focus of this thesis is on expressiveness rather than performance. Although the transformation implementation is fast enough for this specific example of a Pacman game, performance analysis is needed for larger-scale games. At the model structure level, it is noted how topologically similar the `UserControlled` rules and `GhostMove` CRules are. Re-use and parametrization of transformation models deserves further investigation. For the presented Modelling and Simulation-based design application, we could also enhance the game with Dynamic Difficulty Adjustment techniques as outlined in [Hun05]. For example, the user speed could be measured in real-time and compared with the simulated user speeds. The speed of the ghost can then be adapted appropriately.

10

The Class Diagram To Relational Database Benchmark

In this chapter, we evaluate the expressiveness of *MoTif* by solving a standard benchmark in model transformation: the class diagram to relational database schema transformation. We also compare our solution to others.

10.1 Introduction

A plethora of model transformation languages co-exist nowadays. This diversity makes it hard for a customer to choose the most appropriate tool to solve his problem. Having realized this issue, the model transformation community proposed a common case study at the Model Transformations in Practice Workshop (MTiP) workshop [BRT05] co-located with the MoDELS conference in 2005. Various solutions were submitted to the workshop, mostly graph transformation based. Today, solutions have been constructed in all model transformation languages. It has thus become a standard benchmark for evaluating the expressiveness of the different languages.

In this chapter we propose to solve the case study in *MoTif* and thus compare its expressiveness with other common transformation languages. This comparison is a complement to the one drawn in Chapter 2. The following section outlines the specifications of the benchmark description. In Section 10.3, we solve the problem using *MoTif*. Finally in Section 10.4, we discuss similarities and differences of our solution with respect to others.

10.2 Description of the Benchmark

The case study was originally proposed by the OMG [Obj04] in 2004. However, the case study used in this chapter is based on the benchmark proposed at the 2005 MTiP workshop [BRT05]. There are several versions of this example—nearly every paper uses its own version. It is the most popular case study in model transformation. Its goal is to transform a class diagram model into an equivalent relational database schema. We will therefore call it the CD2RDBMS benchmark.

Figure 10.1(a) presents the meta-model of class diagrams considered in the benchmark. It consists

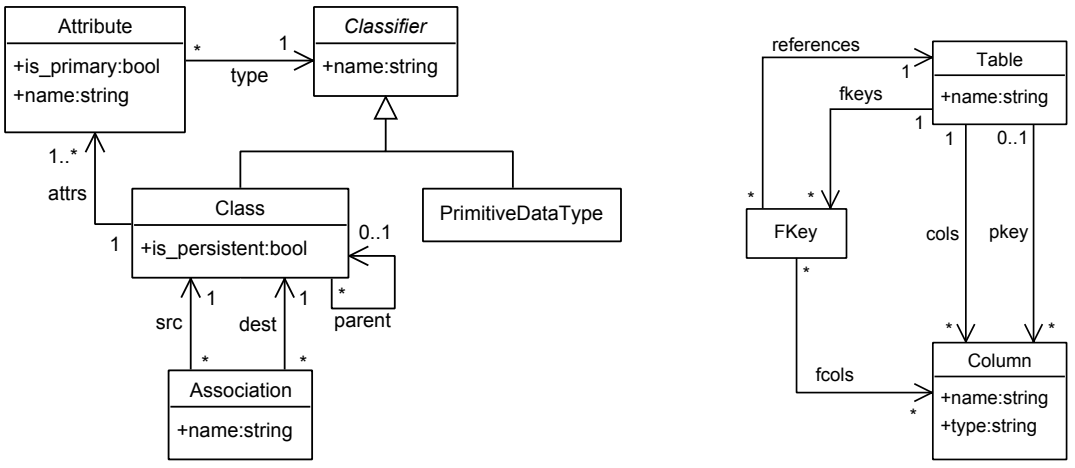


Figure 10.1: The class diagram meta-model in (a) and the relational database schema meta-model in (b).

of classes and directed associations. A class is either persistent or not. It can inherit from another class and contains at least one attribute. At least one attribute must constitute the class's primary key. The type of an attribute is either a primitive data type (simple) or another class (complex). Associations represent one-to-one relationships. Additional constraints where added to the FAQ page of the workshop [BRT05]: In an inheritance hierarchy, only the top-most class is persistent. Furthermore, sub-classes may not add new primary keys to the ones inherited from their parent.

Figure 10.1(b) presents the meta-model of relational database schemas considered in the benchmark. It consists of at least one table. A table contains at least one column; at least one of them must be assigned to the table's primary key. A table may contain a foreign key. Each of them refers to the particular table it identifies and denotes the columns of the table that are part of the foreign key. The transformation's directives are the following:

1. A class is transformed into a table if and only if it is persistent. The corresponding table shall have the same name and one or more column of every attribute of the class and every association for which the class is the source.
2. An attribute whose type is a primitive data type shall be transformed into a single column whose type is the same name as the primary data type's name.
3. An attribute whose type is a persistent class shall be transformed into one or more columns, created from the persistent class' primary key attributes. The columns name shall be prefixed by the name of the attribute followed by an underscore ("_"). The resulting columns must be part of a foreign key that refers to the persistent class. An association whose destination class is persistent must be transformed in a similar way.
4. An attribute whose type is not a persistent class shall be transformed into one or more columns. The columns name shall be prefixed by the name of the attribute followed by an underscore ("_"). The columns shall be placed in tables created from persistent classes. An association

whose destination class is not persistent must be transformed in a similar way. Primary and foreign keys of the translated non-persistent classes must be merged in appropriately, taking into consideration that the translated non-persistent class may contain primary and foreign keys from an arbitrary number of other translated classes.

5. All attributes of the (recursive) parent classes and all associations whose source is one of the parents classes shall be considered. The resulting columns are therefore all merged in the table corresponding to the top-most table. An attribute with the same name as an attribute in a parent class is considered to override it.
6. Associations are not directly transformed. However, each association which has a particular class as a source must be considered when transforming that class into a table and/or columns.
7. Transformations shall not create duplicate elements with the same names when merging attributes and associations as well as when foreign keys point to the same columns of a table.

The specifications imply three requirements with recursive nature: (1) the drill-down of the inheritance hierarchy, (2) the recursive propagation of the columns corresponding to non-persistent attributes and associations, and (3) the recursive propagation of the foreign keys involved with non-persistent classes.

10.3 The Solution in MoTif

Now we describe how to solve the CD2RDBMS transformation in *MoTif*. First we outline the implementation steps to design and run the transformation and then we present the details of the transformation specification.

10.3.1 Development of the Implementation

We follow the MPM transformation development methodology presented in Section 7.5. In our solution, we first define the class diagram (CD) and relational database schema (RDBMS) meta-models in the tool *AToM*^{3}. Then we synthesize a modelling environment for each language and the meta-model of the transformation model CD2RDBMS. Recall from Chapter 5 that the latter is obtained with the RAM process. Thanks to the CD2RDBMS meta-model, we generate a modelling environment for describing the pattern specifications of the transformation. All the patterns appear in Figure 10.2. The meta-model of the patterns allows one to create fragments of CD and RDBMS models augmented with constraints and action statements and an appropriate graphical concrete syntax (c.f. Figure 10.2). Furthermore, generic links allows one to connect elements within and across the CD and RDBMS languages. Then, based on the *MoTif* meta-model, we design a scheduling model that encapsulates the patterns such as the one in Figure 10.3. The CD2RDBMS patterns are compiled into Himesis patterns and the *MoTif* model is translated into a *MoTif-Core* model. The resulting transformation model is loaded in the common transformation framework to be executed. From the CD meta-model, we define a CD model to input to the transformation. At the end of the transformation execution, CD2RDBMS produces a multi-language model partitioned into two sub-models: one is the original CD model and the other is the produced RDBMS model, as it is an in-place transformation.

10.3.2 The Transformation Model

The core of the transformation resides in the specification of the patterns enclosed in rules and their scheduling in *MoTif*. The general idea is first to relate all classes to tables and simple attributes to columns, then complex attributes and associations are mapped to columns along with temporary structures, and finally all temporary artefacts are removed. The proposed solution consists of 21 rules, shown in Figure 10.2, organized as illustrated by the model in Figure 10.3.

First, the FRule `TopClass2Table` creates a table for each top-most class, even for non-persistent classes. This one-to-one mapping maintains generic links which act like traceability links. Then, `DrillDownInheritance` relates each sub-class to the table of its top-most parent. The rule is encapsulated in an SRule which allows one to recursively traverse each inheritance hierarchy. The NAC prevents ensures that the rule is executed only once per class. When every class is mapped to exactly one table, the rule `PDTAttribute` creates a column for every simple attribute in the table corresponding to the enclosing class. This is an SRule since we should prevent the creation of multiple columns with the same name, coming from a parent class' attribute. At this point all columns corresponding to attributes with a primitive data type have been created. Thus the FRule `SetPK` can safely create the primary keys of each table.

Next is to process complex attributes. Note that from the specifications, persistent attributes and associations with a persistent class as destination are transformed in a similar way. The same holds for non-persistent attributes and associations with a non-persistent class as destination. We will therefore describe the mapping for associations only, as mapping complex attributes is analogous. The next rule block is a BSRule with four branches, one for each case of (non-)persistent attribute or association. Only one applicable branch is executed. After its application, the BSRule `ProcessReferences` is re-applied until no branch is applicable. For the persistent case, the rule `PAssociation` creates a foreign key between the tables corresponding to the source and destination classes of the association. Additionally, a column is created in the table corresponding to the source class to be part of the foreign key. Note that although classes labelled 1 and 3 in the LHS are distinct elements, they may be matched to the same element in the input model as none of them is deleted in the RHS. Hence self-associations are also considered. The NAC prevents the application of the rule on the same association for each iteration of the BSRule. For the non-persistent case, each column of the table corresponding to the destination class is copied to the table corresponding to the source class. After each application of `NPAssociation`, the ARulePropagatePK ensures that if the original column was part of the primary key of its table, then the copied column is also part of the primary of its table. The BSRule processes each association recursively so that the columns and foreign keys correctly appear in the required tables. However, one should still take care of the foreign keys. On the one hand, the above process creates a separate foreign key for each column of each related table. The BSRule `MergeFK` therefore merges such foreign keys into a single one involving all columns referring to a same table. A BSRule is required in this case to handle alternation of associations and complex attributes recursively. On the other hand, the BSRule `PropagateFK` propagates all foreign keys of tables corresponding to non-persistent classes to the tables corresponding to persistent classes that refer to them. Note that for each propagated foreign key, the FRule `PropagateFKCol` ensures that the new foreign key refers to

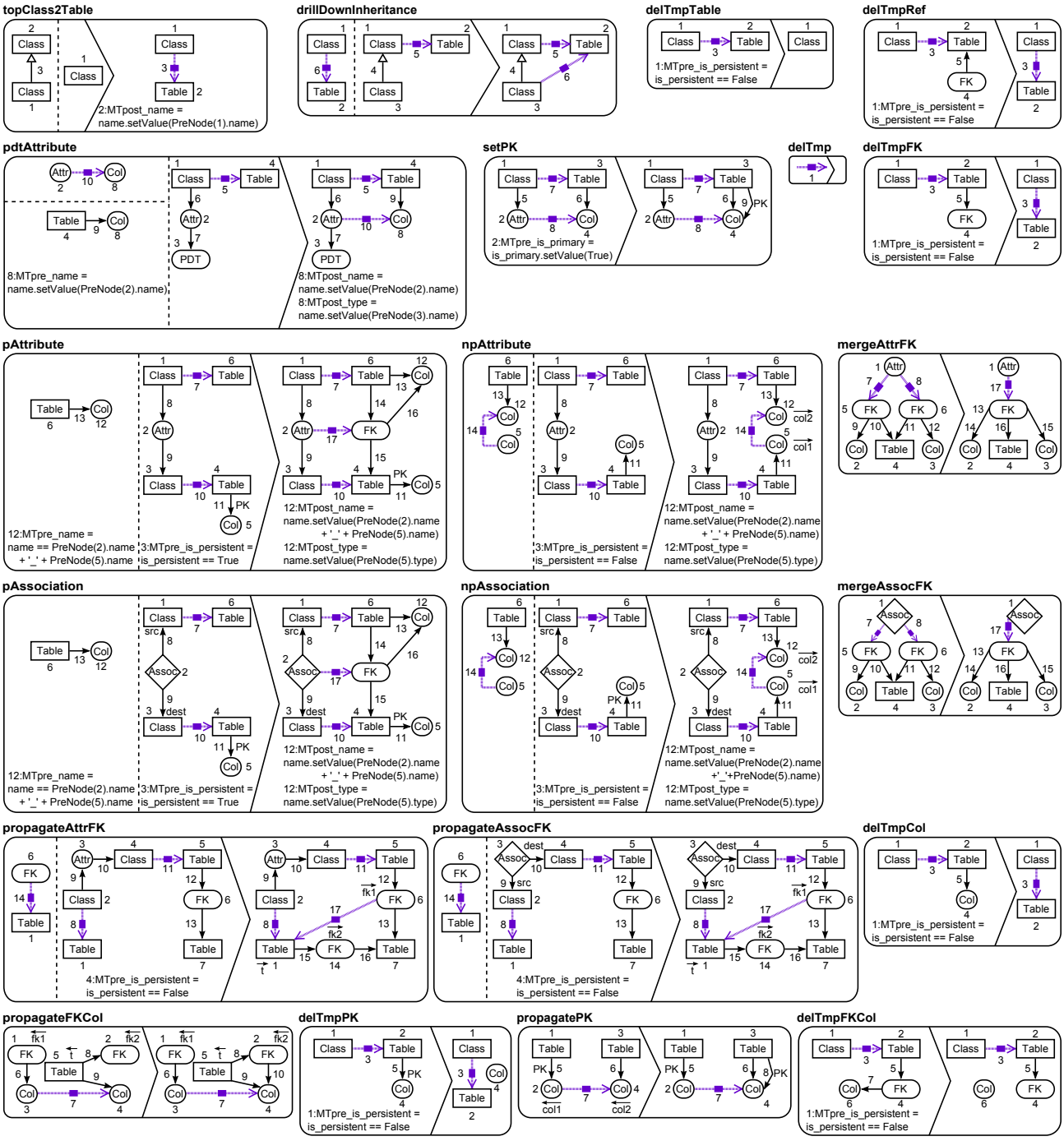


Figure 10.2: The CD2RDBMS transformation rules.

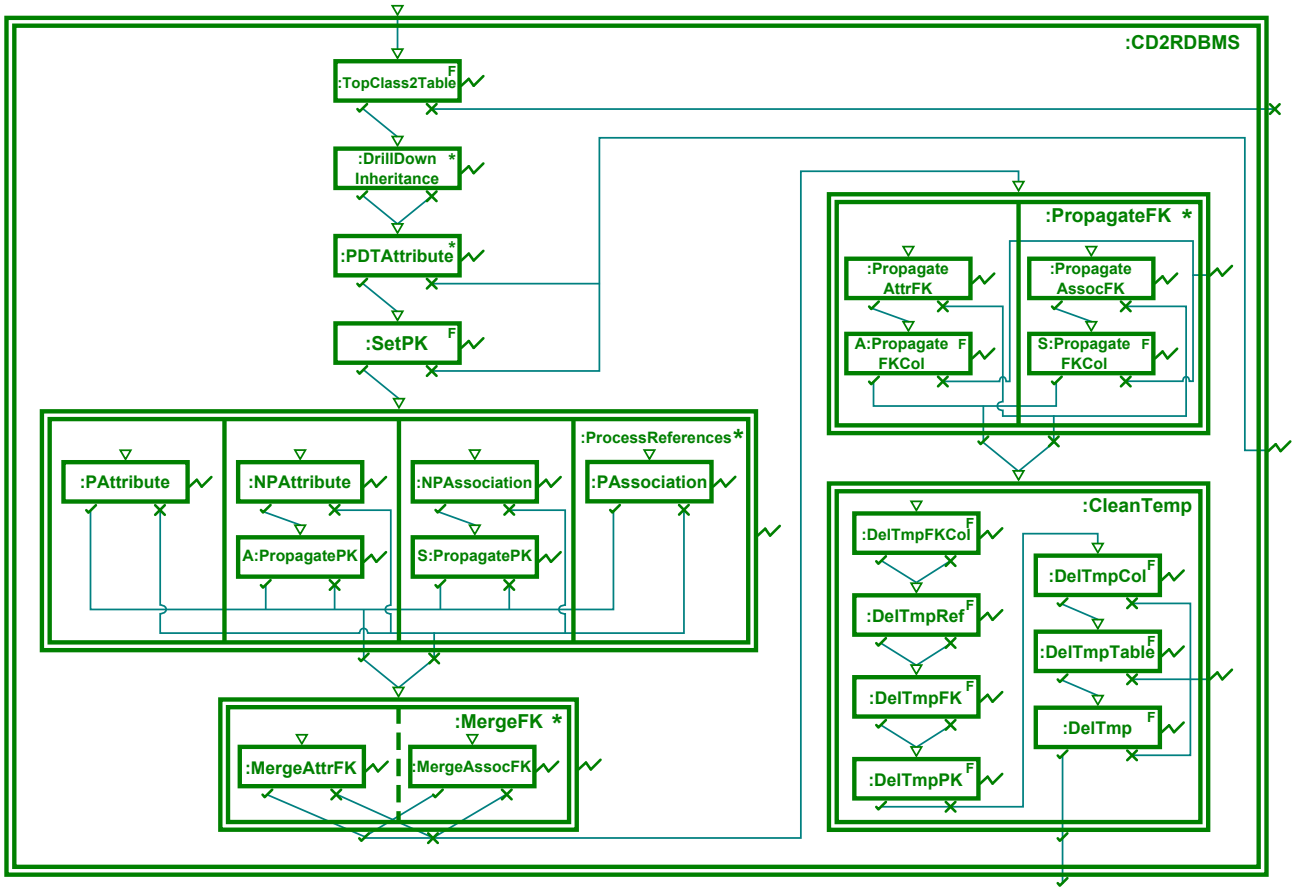


Figure 10.3: The CD2RDBMS transformation block.

the columns of its own table.

At this point, the mapping of all persistent classes is complete. It is thus possible to remove all temporary artefacts: columns, foreign keys, and tables (and the links between them) mapped from non-persistent classes, as well as all generic links. The sub-rule blocks in the `CleanTemp` CRule ensures this clean up. Here, the order in which all the FRule are applied is important to prevent the presence of dangling references.

10.4 Comparison With Other Solutions

The CD2RDBMS benchmark was first attempted by graph transformation tools. In [TEG+05], four of these tools solved the case study with different approaches. This lead to the identification of the different expressiveness of each. This was the first comparative study of contemporary model transformation tools along a common case study. In this section, we enrich this work by extending, on the one hand, the number of model transformation languages considered and, on the other hand, the properties of the comparison. A summary of the extended part of the comparison is given in Table 10.1.

The graph transformation languages involved in the former comparison were: *AGG*, *AToM*^{3}, *VIATRA2*, and *VMTS*. A solution with a preliminary version of *QVT-R* was also considered in the sample. Here, we compare the solution using *MoTif* with the same set of graph transformation languages. Furthermore, we have added other solutions using a non-graph transformation approach: *ATL* [ATL], *MOLA* [MOL], *QVT-R* (as it appears in [Obj08]), and *QVT-OM* [Obj08].

The former paper compared some features of each approach, the strategy and technique of each solution to the case study, as well as the tool accompanying each transformation language. Here, we focus on the following properties.

Meta-model modification. Some solutions needed to modify the original meta-models of CD and RDBMS to be able to correctly implement the transformation and simplify the rule specifications. For example, *AGG* duplicated the attribute and association elements in the CD meta-model to recursively compute the prefix of the corresponding column name, while *MOLA* and *VMTS* duplicated the links in the CD meta-model for the same purpose. The use of `keys` in the RDBMS meta-model of both solutions of QVT instruments the transformation to not create duplicate elements and thus does not require rules to merge them as `MergeAttrFK` and `MergeAssocFK` in Figure 10.2. *VIATRA2* introduced an ancestor link to easily drill-down the inheritance hierarchy and super-classed the attribute and association elements to avoid replicating rules for both elements as it is the case in *MoTif*. *ATL* simplified a major part of the benchmark by not considering non-persistent keys. *AToM*^{3} and *MoTif* are the only solutions that use the original meta-models without altering them.

Helper structures. Because of the recursive nature of the transformation, all graph transformation approaches require the use of helper structures. However, this often forces the modeller to manually extend the input and output meta-models with these elements. Nevertheless, *AToM*^{3} and *MoTif* are multi-paradigm modelling frameworks and thus offers constructs to connect elements from different meta-models natively. In the *QVT-OM* solution, the transformation temporarily creates an array of attributes for each persistent class to collect them from non-persistent ones. Both *QVT-R* and *ATL* do not require any helper structure.

Rule optimization. This property outlines the feature of each transformation language that reduces the number of rules to specify. For example, the order of the `FRules` in the `CleanTemp CRule` allow one to reduce the number of rules from 12 to 7 in the *MoTif* solution.

Number of rules. An quantitative way to compare the expressiveness of model transformation languages is in the number of rules. We have split this number in two parts: the rules required to produce the corresponding RDBMS model and those required to remove all temporary artefacts. *ATL* has the least number of rules since it only solves a subset of the transformation; it should therefore be ignored. The solution in *QVT-R* stands out with a total of eight rules each specified in less than 20 lines of code. This is mainly due to the ability to invoke other rules as method calls with parameters and recursion.

Incremental. A nice side-effect of *VIATRA2* is that it does not require one to reconstruct a completely new output model from scratch when modifying the input meta-model. The specification in

QVT state that the transformation can be executed incrementally, if properly implemented (this is still not the case in the tools MediniQVT and SmartQVT).

Bidirectional. Thanks to the declarative nature of relations in *QVT-R*, a single transformation model can be executed from the input to the output meta-model and vice-versa, following the check-/enforce semantics.

Syntax. *ATL*, *QVT-R* and *QVT-OM* transformation models are specified in textual syntax, while the remaining are in graphical visual concrete syntax. Only *AToM*^{3} and *MoTif* allow one to use a concrete syntax for the patterns in the rules. Furthermore, in the *MoTif* framework, a domain-specific language is automatically synthesized from the meta-models involved in the transformation.

Debugging. To facilitate the development of the CD2RDBMS transformation, the transformation language—or at least the supporting tool—should offer debugging facilities to the modeller. *VIATRA2* and *VMTS* only provide a log of the steps performed a run-time, although the latest version of the latter allows step-by-step animation of the input model similar to *AToM*^{3}. *AGG* allows a manual selection of the rules and the match of each rule. A critical-pair analysis can also be performed on the transformation to verify causality dependencies between rules. However, *MoTif* is the only transformation language that supports exception handling modelled at the level of the transformation model.

	AGG	ATL	ATOM^3	MOLA	MOTIF	QVT-R	QVT-OM	VIATRA2	VMTS
Meta-model modif.	Extra attributes + associations	No non-persistent classes	no	Intra-formalism links	no	keys	keys	Ancestor link + Property	Intra-formalism links
Helper structures	Inter-formalism links	no	Built-in generic links	Inter-formalism links	Built-in generic links	no	Extra attributes	Inter-formalism links	Inter-formalism links
Rule optimization	Layers	Simplification of CD meta-model	N/A	Recursion, parameters, method calls	FRule, BSRule	Recursion, parameters, method calls	Recursion, parameters, method calls	*OR-pattern*, *forAll*	Internal / external causalities
Number of rules	17 + ?	6 + 0	13 + 12	25 + 0	14 + 7	8 + 0	8 + 0	12 + ?	11 + 5
Incremental	no	no	no	no	no	yes	yes	yes	no
Bidirectional	no	no	no	no	no	yes	no	no	no
Syntax	Visual + abstract syntax	Textual + abstract syntax	Visual + concrete syntax	Visual + abstract syntax	Visual generated DSL + concrete syntax	Textual + abstract syntax	Textual + abstract syntax	Visual + abstract syntax	Visual + abstract syntax
Debugging	Manual selection of rule & match	Breakpointing & state inspection	Manual selection of match	N/A	Exception handling	N/A	N/A	Log window	Log window

Table 10.1: Comparison of model transformation tools for the CD2RDBMS case study

10.5 Conclusion

This chapter describes an MPM solution to the CD2RDBMS benchmark implemented in *MoTif*. It serves as an example on how to develop a transformation. The expressiveness of *MoTif* allows one to elegantly solve non-trivial case studies such as this one. Finally, this chapter demonstrates that developing a model transformation in a completely modelled framework alleviates the cognitive effort of the modeller.

The comparison proposed in Section 10.4 can be extended. For example, we considered the number of rules as a quantitative measurement for a qualitative feature such as expressiveness. In the future, we will investigate on how to improve the metrics for expressiveness of model transformation languages. Possible candidates are the distribution of the size of the rules, the total number of application points (matching points), the size of textual constraints or actions for graphical languages, or even the number of user clicks. What is also important to evaluate is the performance of transformation languages.

11

The AntWorld Benchmark

In this chapter, we evaluate the performance of model transformation languages developed in the framework presented in this thesis. This evaluation is based on a case-study proposed at the 2008 edition of the Graph-Based Tools (GraBaTs) workshop [SV08b]. The case-study constitutes a standard benchmark for graph transformation languages, where the focus is on local search performance. The solution presented here is designed, on the one hand, in *Py-T-Core* to compare its performance efficiency with other transformation tools and, on the other hand, in *MoTif* to quantify the overhead of executing a transformation in a *DEVS* environment.

11.1 Introduction

Graph transformation is an attractive approach to perform model transformation thanks to the general and easy-to-grasp nature of the underlying graph formalism, as well as the intuitive way in which graph changes can be encoded using transformation rules. Moreover, it allows one to easily capture concepts from many different domains and rapidly develop a given system as a prototype model. The price to pay for the generality that makes graph transformation widely applicable is the cost of manipulating graphs, including storage, editing, and rule application. In particular, the latter involves graph matching, which is an NP-hard problem. In applications where graphs are large or the number of transformation steps is large, it is therefore crucial to optimise tool performance.

Throughout this thesis, we have developed a framework for modelling and executing model transformation languages. This work focused on the design and the expressiveness of the languages. Although some performance analysis was done in Chapter 4, it evaluated the performance of the underlying data structure, namely Himesis. This chapter presents some results regarding the performance of *MoTif* based on a standard case study described in the next section. We compare the solution using *MoTif* with the same solution implemented in *Py-T-Core* to evaluate the overhead of executing a transformation in a *DEVS* environment.

The following section outlines the specifications of the AntWorld case study. In Section 11.3, we illustrate two solutions to the case study problem: one using *MoTif* and one using *Py-T-Core*. Then, Section 11.4 analyses the performance of each solution and discusses their differences. Finally in Section 11.5, we compare our results to other solutions to the case study.

11.2 The AntWorld Case Study

The case study used in this chapter is based on case no. 2 (AntWorld Simulation case study) of the GraBaTs 2008 tool contest [VGR08]. This is a benchmark for the comparison of graph transformation tools that stresses local rule application. The complete description of the behaviour can be found in [Zün08] and is as follows.

The AntWorld simulation map is discretized into concentric circles of nodes (representing a large area) centered at a hill (the ant home). Ants are moving around searching for food. When an ant finds food, it brings it back to the ant hill in order to grow new ants. On its way home, the ant drops pheromones marking the path to the food reservoir. If an ant without food leaves the hill or if a searching ant hits a pheromone mark, it follows the pheromone path leading to the food. This behaviour already results in the well known ant trails.

The AntWorld simulation works in rounds (similar to time-slices). Within each round, each ant makes one move. If an ant is not in carry mode and is on a node with food parts, it takes one piece of food and enters carrying mode. Note that it may still move within the current round. On the other hand, if an ant carries some food, it follows the links towards the inner circle one node per round. During its way home (towards the unique hill at the centre of all node), on each visited node (including the node that it picked food from) the ant drops 1024 parts of pheromones in order to guide other ants to the food place. However, if a carrying ant is on the hill, it drops the food and enters the search mode. It may leave the hill within the same round. Any ant without food is in search mode. In this mode, the ant checks the neighbouring node(s) of the next outer circle for pheromones. If some hold more than 9 parts of pheromones, the ant chooses one of these nodes randomly. Otherwise, the ant moves to any of its neighbour nodes based on a fair random choice (but never enters the hill).

Whenever during one round an ant is on a node on the outmost circle, a new circle of nodes shall be created. For each outmost grid node, a new grid node is created; but three nodes are created in the case of a main axis node. During the creation of this next circle, every 10^{th} node shall carry 100 food parts. If a circle has for example 28 nodes, node 10 and node 20 of that circle shall have food. Thus, this circle would need just two more nodes to create a third food place. Therefore, these 8 nodes are kept in mind and during the creation of the next circle (in our example with 36 nodes) we add another food place when two more nodes have been added. Thus, across circles, every 10^{th} node becomes a food place. After each round, all pheromones shall evaporate: reducing by 5%. Also, the hill shall consume the food brought to it by creating one new ant per delivered food part.

11.3 The Solution

In our approach we define the syntax and semantics of the AntWorld formalism in the tool *AToM*^{3}. On the one hand, we synthesize a domain-specific modelling environment to design the initial model of the transformation as illustrated in Figure 11.1. The operational semantics of the formalism is implemented in *AToM*^{3} using the *MoTif* language.

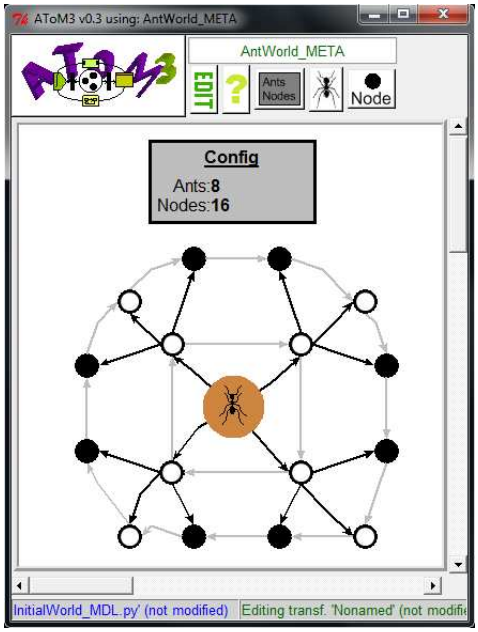


Figure 11.1: The input model.

11.3.1 The AntWorld Language

As shown in Figure 11.2, the AntWorld formalism consists of ants and grid nodes. An Ant element can be standing on one *GridNode* at a time. The "carry mode" of the ant is modelled by its `hasFood` attribute. A grid node can either be on the main axis or be a hill or neither of them. Grid nodes can hold pheromones and food parts. The grid nodes are connected in circles following the Right association and are centred at the hill following the Forward association. The former connects grid nodes on the same concentric circle and the latter connects grid nodes on one circle to neighbouring grid nodes on the next outer circle. A Configuration element is added to keep track of the number of ants and the number grid nodes, which summarizes a snapshot of the model while simulating. Using *AToM*^{3} as a

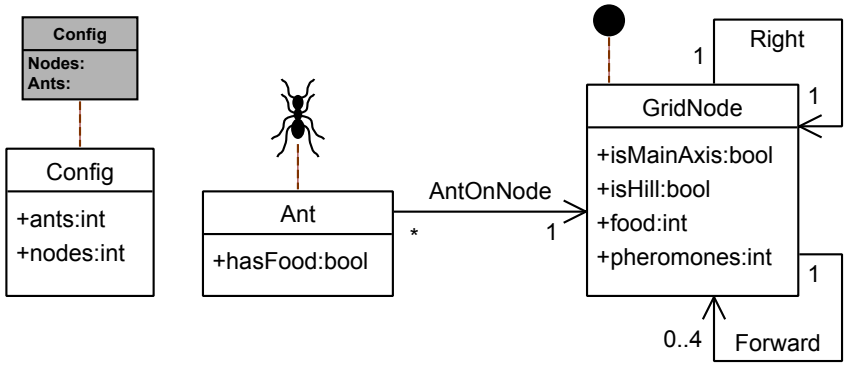


Figure 11.2: The AntWorld meta-model.

modelling environment enables one to associate a concrete syntax for each meta-model element. The meta-model in Figure 11.2 is annotated with the icon representing visually each meta-model element. Note that for grid nodes, it is possible to specify a different icon depending whether it is a hill or on a main axis. Also, the Forward associations are represented by a full line arrow and the Right associations are represented by a dashed line arrow.

11.3.2 AntWorld Simulation

We define the operational semantics of the AntWorld language as a graph transformation in lieu of model transformation in *MoTif*. After creating a meta-model of the transformation following the RAM process of the AntWorld meta-model for both pre- and post-condition patterns (c.f. Chapter 5), we automatically synthesize an environment for designing a *MoTif* transformation, specific to AntWorld in *AToM*^{3}. Figure 11.3 depicts all the graph transformation rules necessary for the simulation. In complement, Figure 11.4 depicts the *MoTif* workflow controlling the application of these rules.

11.3.3 MoTif Solution

The `Round` CRule encapsulates the actions within a single round of the simulation. First it lets all ants move once, giving priority first to ants carrying food and then to those in searching mode. After that and if necessary, a new circle of grid nodes is created. Then, if the hill contains food parts, the SRule `AntBirth` creates as many ants as there are food parts. Finally, the FRule `Evaporate` reduces the amount of pheromones present on grid nodes by 5%.

The `MoveCarryingAnts` block is an LRule iterating over each ant in carry mode. If it is on the hill then, with the ARule `DropFood`, the ant deposits the food part and is back in search mode. Otherwise, it must move on the next inner circle. If the ARule `MoveToHill` fails, then there must be design fault in the transformation (according to the requirements). A transformation-specific exception is then raised and propagated up to the `Round` block.

The `MoveSearchingAnts` block is an LRule iterating over each ant in search mode. First, the rule `MoveToPhero` gives an ant the chance to move to a grid node on the next outer circle containing pheromone drops. Recall that the requirements specify that if there are multiple grid nodes with pheromones adjacent to the current one, one of them must be selected randomly. In order to not rely on the order in which the matching algorithm chooses the nodes, `MoveToPhero` is an FRule with a maximum of 1 iteration. Thus, in its corresponding *MoTif-Core* model, the Matcher will find all neighbouring grid nodes that satisfy the condition and the Iterator will choose one of them randomly. If no such grid node is found, the BRule `Move` lets the ant move in any direction non-deterministically. Again here, the requirements specify that the direction must be chosen randomly. Recall that the select function of the *MoTif-Core* Composer corresponding to this block is implemented in a Monte-Carlo sense like the selection process of the Iterator: randomly but repeatable using sampling from a uniform distribution to provide a reproducible, fair sampling. Since all ants must move within a round, the failure of the BRule generates an exception as before. After a searching ant has moved, the ARule `GrabFood` allows it to get a hold a food part on the grid node it is currently on, if any. Note that if an ant in carry mode had dropped a food part on the hill in the previous LRule, it will move out of the

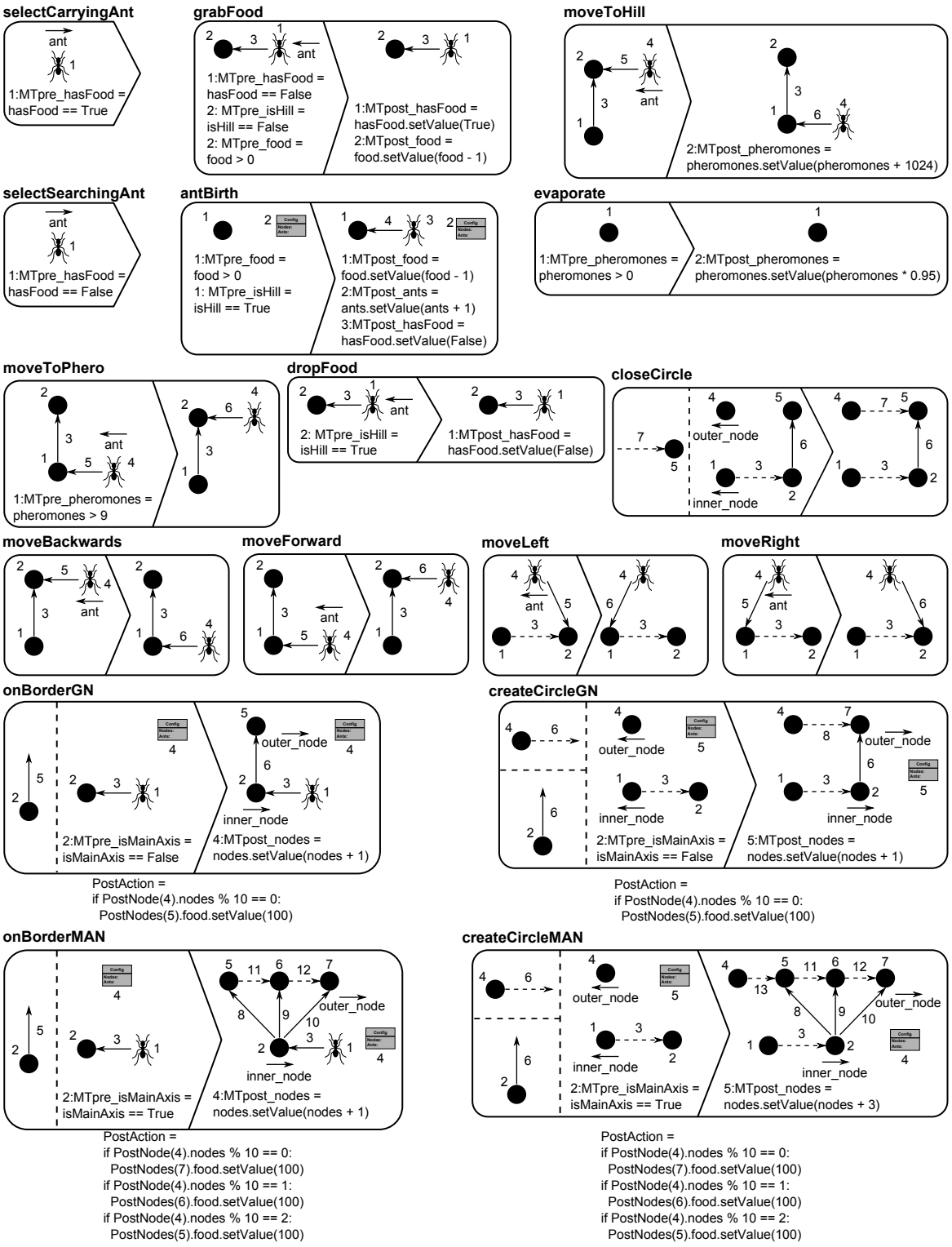


Figure 11.3: The AntWorld transformation rules.

hill after this LRule is applied.

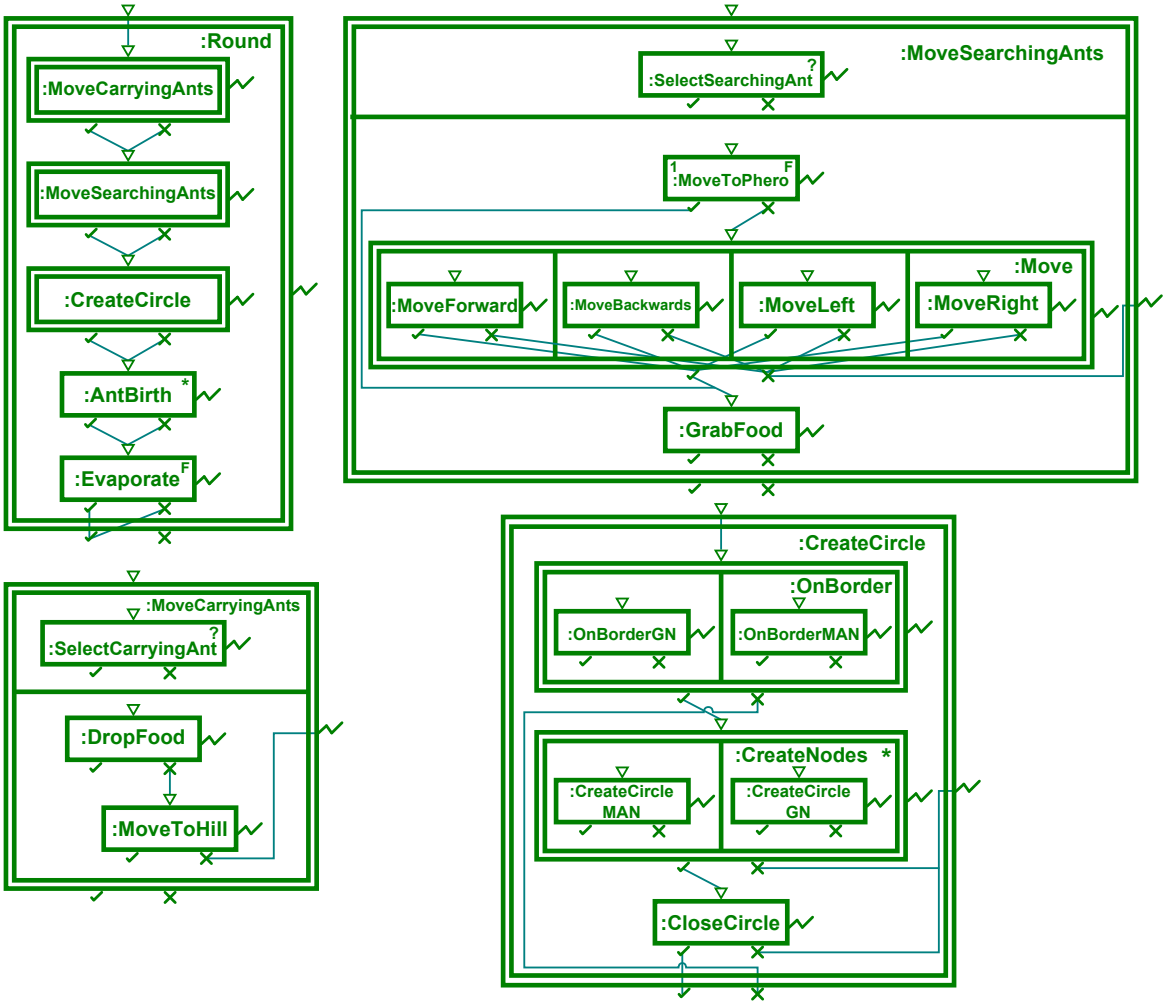


Figure 11.4: The Round transformation block.

After all ants have moved, the CRule `CreateCircle` encapsulates the creation of a new outmost circle. To test whether it necessary to create one, the BRule `OnBorder` looks for an ant at the border of the current map. In the successful case, a series of new grid nodes (and main axis nodes) are created starting from the grid node on which the ant was found. The rules `CreateCircleGN` and `CreateCircleMAN` make sure that every tenth node created holds the required number of food parts. These two ARules are enclosed in a variant of the BRule: the BSRule. It applies every branch recursively, giving to each the chance of re-applying at each iteration. Note that if `OnBorder` was successful, then `CreateNodes` must be applied. If not, an exception is raised as before. Finally, the ARule `CloseCircle` makes sure that the new circle is closed by connecting the last newly created grid node to the first one. The transformation rules of the simulation make use of two pivots to optimize the execution. The *`ant`* pivot points to the ant to move in each LRule. Alternatively, one could have altered the RAMified meta-model by augmenting the Ant class with a *`isProcessed`* boolean attribute, to

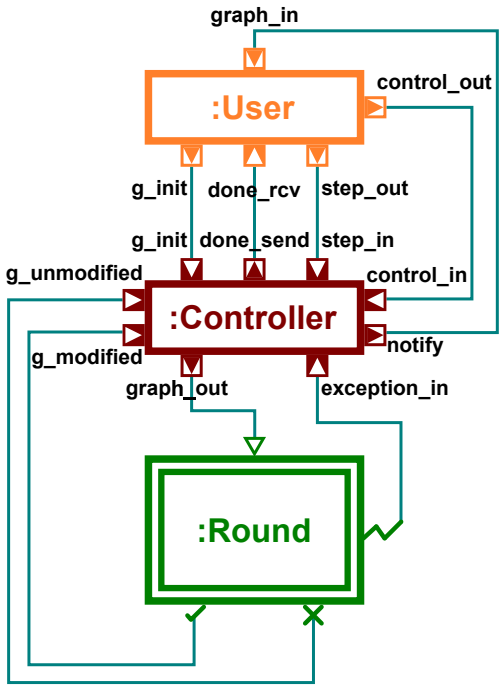


Figure 11.5: The overall transformation model.

move each ant exactly once per round. However, this would require an additional rule to reset the flag on all ants. The use of pivot reduces the search space in the matching phase of the rules by pre-binding the selected ant to move. Similarly, the *node* pivot is used during the creation of a new outer circle. All the rules in the CRule `CreateCircle` can be specified without this pivot and still produce a correct result. However, every time a rule is matched, it will have to search through all grid nodes of the map instead of focusing on the local region of the node bound to the pivot. Another optimization in this model is that no Resolver is needed in the *MoTif-Core* model generated from the presented *MoTif* model. The only FRule of the transformation loops over all grid nodes of the map and therefore no matching conflicts with another. As for the LRules, no rule nesting is present and thus no conflict is possible.

Similar to the previous case studies, the transformation environment is entirely modelled in *DEVS*. As depicted in Figure 11.5, the Controller receives the initial model from the User and sends it to the `Round` transformation unit. It also receives from the User the number of rounds the simulation shall make (set to infinity by default). After each round, the Controller receives the graph from the `Round` and notifies the User about its current state. If an exception occurred during the round, the simulation is stopped and the transformation exception is propagated up to the user.

When debugging, we have implemented a graphical visualisation of the graph received in the external transition function of the User. Because the received model is implemented as a Himesis graph which relies on the IGraph Python module (c.f. Chapter 4), the API allows one to graphically render the graph using Cairo. A snapshot taken at round 35 shows a model with 33 ants and 257

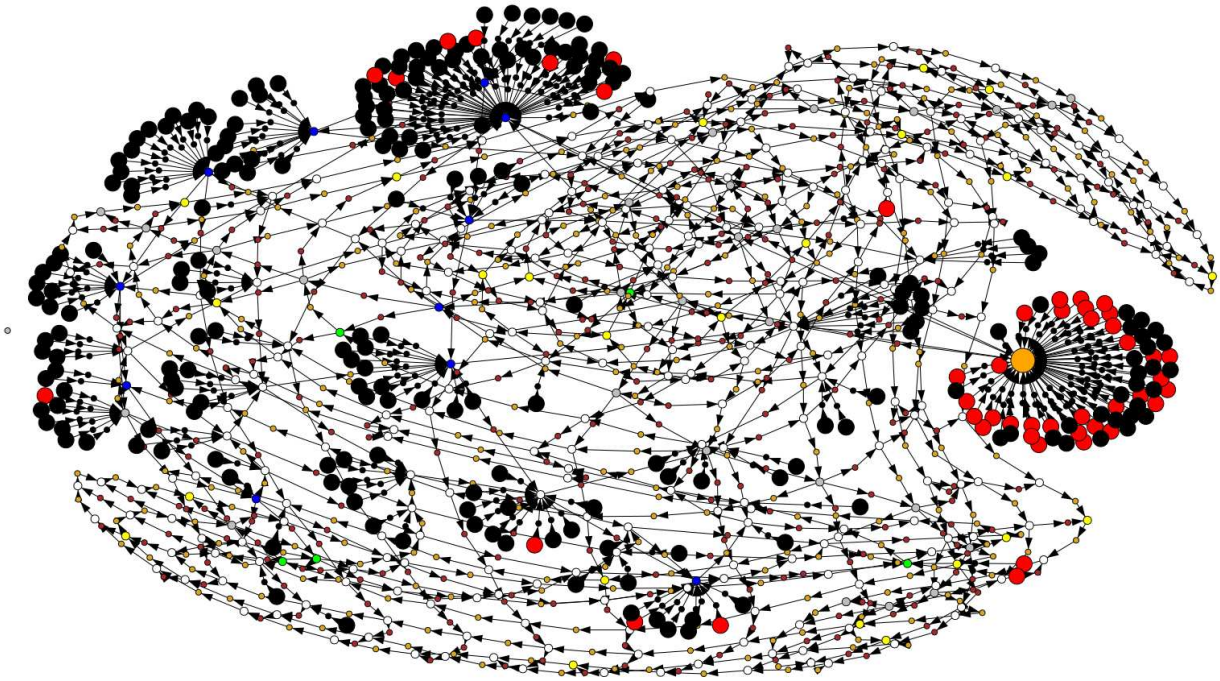


Figure 11.6: A snapshot of the simulated model at round 55.

grid nodes in Figure 11.6. Ants in search mode are represented as large black rounds and ants in carry mode are in red. Grid nodes are rendered as small black rounds, main axis nodes as small white circles, and the hill as a large orange round. Grid nodes with at least 9 pheromones are coloured in blue, those holding food are in yellow, and grid nodes with both food are in green.

11.3.4 Py-T-Core Solution

The *Py-T-Core* solution is implemented exactly like the *MoTif* solution (but without time advance). The following listings show snippets of the implementation in Python.

Listing 11.1: *Py-T-Core* implementation of the AntWorld simulation.

```
class MoveCarryingAntsLoop(Composer):
    def __init__(self):
        super(MoveCarryingAntsLoop, self).__init__()
        self.DropFood = ARule(HDropFoodLHS(), HDropFoodRHS(),
                              ignore_resolver=True)
        self.MoveToHill = ARule(HMoveToHillLHS(), HMoveToHillRHS(),
                                ignore_resolver=True)

    def packet_in(self, packet):
        self.exception = None
        self.is_success = False
```

```
        # Drop Food
        packet = self.DropFood.packet_in(packet)
        if not self.DropFood.is_success:
            if self.DropFood.exception is not None:
                self.exception = self.DropFood.exception
                return packet
            # Move To Hill
            packet = self.MoveToHill.packet_in(packet)
            if not self.MoveToHill.is_success:
                if self.MoveToHill.exception is not None:
                    self.exception = self.MoveToHill.exception
                    return packet
                else:
                    self.exception = TransformationException(
                              None, "Carrying Ant can't move to hill!")
                    self.exception.packet = packet
                    self.exception.transformation_unit = self
                return packet
        # Output
        self.is_success = True
        return packet

class MoveSearchingAntsLoop(Composer):
    def __init__(self):
        super(MoveSearchingAntsLoop, self).__init__()
        self.GrabFood = ARule(HGrabFoodLHS(), HGrabFoodRHS(),
                              ignore_resolver=True)
        self.MoveForwardToPheromones = FRule(HMoveForwardToPheromonesLHS(),
                                             HMoveForwardToPheromonesRHS(),
                                             ignore_resolver=True)
        self.MoveForwardToPheromones.I.max_iterations = 1
        self.Move = BRule(branches=
           [ARule(HMoveForwardLHS(), HMoveForwardRHS(), ignore_resolver=True),
           ARule(HMoveBackwardsLHS(), HMoveBackwardsRHS(), ignore_resolver=True),
           ARule(HMoveLeftLHS(), HMoveLeftRHS(), ignore_resolver=True),
           ARule(HMoveRightLHS(), HMoveRightRHS(), ignore_resolver=True)])

    def packet_in(self, packet):
        self.exception = None
        self.is_success = False
        # Move Forward To Pheromones
        packet = self.MoveForwardToPheromones.packet_in(packet)
        if not self.MoveForwardToPheromones.is_success:
            if self.MoveForwardToPheromones.exception is not None:
                self.exception = self.MoveForwardToPheromones.exception
                return packet
            # Move
            packet = self.Move.packet_in(packet)
            if not self.Move.is_success:
                if self.Move.exception is not None:
                    self.exception = self.Move.exception
```

```
                else:
                    self.exception = TransformationException(
                                None, 'Ant is stuck!')
                    self.exception.packet = packet
                    self.exception.transformation_unit = self
                return packet
        guid = self.MoveForwardToPheromones.M.condition['GUID__']
        if guid in packet.match_sets:
            del packet.match_sets[guid]
        # Grab Food
        packet = self.GrabFood.packet_in(packet)
        if not self.GrabFood.is_success:
            if self.GrabFood.exception is not None:
                self.exception = self.GrabFood.exception
            return packet
        # Output
        self.is_success = True
        return packet

class GenerateCircle(Composer):
    def __init__(self):
        super(GenerateCircle, self).__init__()
        self.CloseCircle = ARule(HCloseCircleLHS(), HCloseCircleRHS(),
                                 ignore_resolver=True)
        self.CreateCircle = BSRule(branches=
             [ARule(HCreateCircleGNLHS(), HCreateCircleGNRHS(),
                    ignore_resolver=True),
             ARule(HCreateCircleMANLHS(), HCreateCircleMANRHS(),
                   ignore_resolver=True)])
        self.OnBorder = BRule(branches=
             [ARule(HOnBorderGNLHS(), HOnBorderGNRHS(), ignore_resolver=True),
             ARule(HOnBorderMANLHS(), HOnBorderMANRHS(), ignore_resolver=True)])

    def packet_in(self, packet):
        self.exception = None
        self.is_success = False
        # On Border
        packet = self.OnBorder.packet_in(packet)
        if not self.OnBorder.is_success:
            if self.OnBorder.exception is not None:
                self.exception = self.OnBorder.exception
            return packet
        # Create Circle
        packet = self.CreateCircle.packet_in(packet)
        if not self.CreateCircle.is_success:
            if self.CreateCircle.exception is not None:
                self.exception = self.CreateCircle.exception
            else:
                self.exception = TransformationException(
                                None, 'Impossible to create new circle!')
                self.exception.packet = packet
```

```
                self.exception.transformation_unit = self
            return packet
        # Close Circle
        packet = self.CloseCircle.packet_in(packet)
        if not self.CloseCircle.is_success:
            if self.CloseCircle.exception is not None:
                self.exception = self.CloseCircle.exception
            else:
                self.exception = TransformationException(
                            None, 'Impossible to close new circle!')
                self.exception.packet = packet
                self.exception.transformation_unit = self
            return packet
        # Output
        self.is_success = True
        return packet

class Round(Composer):
    def __init__(self):
        super(Simulation_Fast_Map, self).__init__()

        self.AntBirth = SRule(HAntBirthLHS(), HAntBirthRHS(),
                              ignore_resolver=True)
        self.Evaporate = FRule(HEvaporateLHS(), HEvaporateRHS(),
                              ignore_resolver=True)
        self.MoveCarryingAnts = LRule(HSelectCarryingAntsLHS(),
                              MoveCarryingAntsLoop())
        self.MoveSearchingAnts = LRule(HSelectSearchingAntsLHS(),
                              MoveSearchingAntsLoop())
        self.GenerateCircle = GenerateCircle()

    def packet_in(self, packet, round):
        self.exception = None
        self.is_success = False
        # Move Carrying Ants
        packet = self.MoveCarryingAnts.packet_in(packet)
        if not self.MoveCarryingAnts.is_success:
            if self.MoveCarryingAnts.exception is not None:
                self.exception = self.MoveCarryingAnts.exception
                return packet
        # Move Searching Ants
        packet = self.MoveSearchingAnts.packet_in(packet)
        if not self.MoveSearchingAnts.is_success:
            if self.MoveSearchingAnts.exception is not None:
                self.exception = self.MoveSearchingAnts.exception
                return packet
        # Generate Circle
        packet = self.GenerateCircle.packet_in(packet)
        if not self.GenerateCircle.is_success:
            if self.GenerateCircle.exception is not None:
                self.exception = self.GenerateCircle.exception
```

```
            return packet
        # Ant Birth
        packet = self.AntBirth.packet_in(packet)
        if not self.AntBirth.is_success:
            if self.AntBirth.exception is not None:
                self.exception = self.AntBirth.exception
                return packet
        # Evaporate
        packet = self.Evaporate.packet_in(packet)
        if not self.Evaporate.is_success:
            if self.Evaporate.exception is not None:
                self.exception = self.Evaporate.exception
                return packet
        # Output
        self.is_success = True
        return packet
```

The following illustrates the implementation of the BRule, BSRule, and LRule in *Py-T-Core*.

Listing 11.2: The BRule implemented in *Py-T-Core*.

```
from t_core.composer import Composer
from util.seeded_random import Random

class BRule(Composer):
    def __init__(self, branches):
        super(BRule, self).__init__()
        self.branches = branches

    def packet_in(self, packet):
        self.exception = None
        self.is_success = False
        remaining_branches = range(len(self.branches))
        # Success on the first branch that is in success
        while True:
            if len(remaining_branches) == 0:
                # They all failed
                return packet
            branch_no = Random.choice(remaining_branches)
            branch = self.branches[branch_no]
            packet = branch.packet_in(packet)
            if not branch.is_success:
                if branch.exception is not None:
                    self.exception = branch.exception
                    return packet
                else:
                    # Ignore this branch for next try
                    remaining_branches.remove(branch_no)
            else:
                self.is_success = True
                return packet
```

Listing 11.3: The BSRuleimplemented in *Py-T-Core*

```
from util.infinity import INFINITY
from t_core.composer import Composer
from tc_python.brule import BRule

class BSRule(Composer):
    def __init__(self, branches, max_iterations=INFINITY):
        super(BSRule, self).__init__()
        self.brule = BRule(branches)
        self.max_iterations = max_iterations
        self.iterations = 1

    def packet_in(self, packet):
        self.exception = None
        self.is_success = False
        # Apply the BRule
        packet = self.brule.packet_in(packet)
        if not self.brule.is_success:
            self.exception = self.brule.exception
            return packet
        else:
            # Rule has been applied once, so it's a success anyway
            self.is_success = True
            while self.iterations < self.max_iterations:
                # Re-apply the BRule
                packet = self.brule.packet_in(packet)
                if not self.brule.is_success:
                    self.exception = self.brule.exception
                    return packet
                self.iterations += 1
            return packet
```

Listing 11.4: The LRule implemented in *Py-T-Core*

```
from util.infinity import INFINITY
from t_core.composer import Composer
from t_core.matcher import Matcher
from t_core.iterator import Iterator

class LRule(Composer):
    def __init__(self, LHS, inner_rule, max_iterations=INFINITY):
        super(LRule, self).__init__()
        self.M = Matcher(condition=LHS, max=max_iterations)
        self.I = Iterator(max_iterations=max_iterations)
        self.inner_rule = inner_rule

    def packet_in(self, packet):
```

```
        self.exception = None
        self.is_success = False
        # Match
        packet = self.M.packet_in(packet)
        if not self.M.is_success:
            self.exception = self.M.exception
            return packet
        # Choose the first match
        packet = self.I.packet_in(packet)
        if not self.I.is_success:
            self.exception = self.I.exception
            return packet
        while True:
            # Apply the inner rule
            packet = self.inner_rule.packet_in(packet)
            if not self.inner_rule.is_success:
                if self.inner_rule.exception:
                    self.exception = self.inner_rule.exception
                    return packet
            # Clean the packet: required since there is no Rewriter in a Query
            if  len(packet.match_sets[self.I.condition].matches) == 0:
                del packet.match_sets[self.I.condition]
            # Choose another match
            packet = self.I.next_in(packet)
            # No more iterations are left
            if not self.I.is_success:
                if self.I.exception:
                    self.exception = self.I.exception
                else:
                    # Output success packet
                    self.is_success = True
                return packet
```

11.4 Performance Analysis

After running the simulation for hundreds of runs, we analyse both solutions.

11.4.1 Properties of the case study

The goal of this case study is to analyse the performance of graph transformation tools with respect to local search. That is in every round, the transformation focuses on the behaviour of each ant individually. In the meantime, the number of grid nodes grows while the ants are discovering the map. Because of pheromones, the trajectories of the ants will form trails to grid nodes with food. It is thus expected that the size of the map grows faster in the beginnings of the simulation while there are not any pheromone trails then when at least a trail is present. Since food is limited on each grid node, the map will grow at a faster rate when no more food is found on a grid node and the ants have to look for other sources of food. Therefore the performance of the simulation relies on two aspects:

the movement of the ants and the evolution of the map. Since pheromones and food parts are implemented as integer attributes, the time the rules spend managing them is negligible compared to ants and grid nodes management. On the one hand, all ants move exactly once per round. Thus the three rule blocks `MoveToHill`, `MoveForwardToPheromones`, and `Move` are applied exactly a times per round, where a is the number of ants in a round. On the other hand, if an ant is on the border of the map, `CreateCircleMAN` and `OnBorderMAN` are applied together 4 times, creating 12 new grid nodes. Also, to have c concentric circles, `CreateCircleGN` is applied $8c-16$ times for $c \geq 3$. From the above observations, the number of grid nodes n and associations e in the map G is given by the following equations:

$$n_c = 4c^2 + 1, c \geq 0 \tag{11.1}$$

$$e_c = 8c^2, c \geq 0 \tag{11.2}$$

$$G_c = 12c^2 + 1, c \geq 0 \tag{11.3}$$

In this case study, there is a significant amount of variability: the number of ants with respect to the number of grid nodes qualifies the transformation setup since it highly depends on the non-deterministic choices of directions ants take. We define the level of the simulation as the number of elements present in the model after a round. Equation 11.4 describes the level of the simulation with the number of Himesis vertices H.

$$H_c = 12c^2 + 2a + 1, c \geq 0 \tag{11.4}$$

11.4.2 Simulation

Table 11.1 shows the performance measurements of the execution of the *Py-T-Core* transformation for up to 1000 rounds. The simulations were performed in the same environment as in Chapter 4. Table 11.2 shows the performance measurements of the same experiment implemented in *MoTif*. A regression analysis reveals a quadratic complexity for both, which corresponds to the other solutions that implemented this benchmark (see Section 11.5).

Round	Level	Ants	Food	Nodes with Phero	Himesis Vertices	Round Time	Total Time
25	6	27	1,274	8	487	0.819	18
50	8	119	2,278	14	1,007	1.843	58
100	18	392	12,268	80	4,673	12	456
200	37	1,177	52,935	314	18,783	112	7,580
300	61	2,393	145,580	657	49,439	664	51,556
400	82	3,989	263,901	1,060	88,667	2,304	204,381
500	102	5,833	408,484	1,537	136,515	5,885	616,019

Table 11.1: Performance measurements of the AntWorld simulation using *Py-T-Core*. Time measurements are in seconds.

Round	Level	Ants	Food	Nodes with Phero	Himesis Vertices	Round Time	Total Time
25	8	19	2,375	16	807	1.494	37
50	8	67	2,283	20	903	1.701	79
100	19	283	13,950	68	4,899	49	1,195
200	31	958	37,098	174	13,449	216	16,197
300	50	2,093	97,193	498	34,187	1,287	92,420
400	73	3,637	208,522	871	71,223	3,736	293,002
500	96	5,177	361,830	1,352	120,947	8,425	874,339

Table 11.2: Performance measurements of the AntWorld simulation using *MoTif*. Time measurements are in seconds.

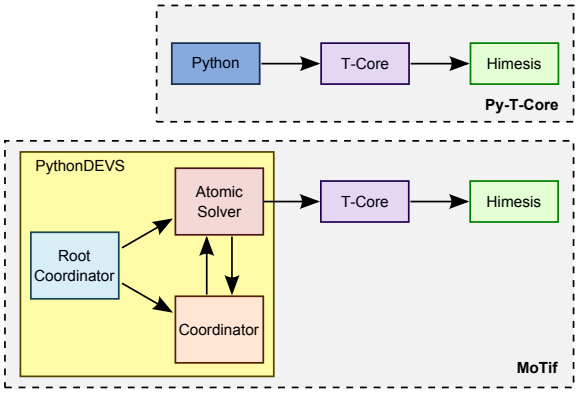


Figure 11.7: The component interaction in *MoTif* and in *Py-T-Core*.

Given these two measurements, we notice that the *MoTif* solution is slower than the *Py-T-Core* solution by a factor of 2.65. This number represents the average of all ratios for the same number of vertices, both for the round time and for the total time. This is to be expected since, at run-time, *MoTif* involves a larger number of interactions with more components than in *Py-T-Core*. As Figure 11.7 illustrates, in *Py-T-Core*, the transformation is specified directly in Python, interfacing with the API of *T-Core*. Furthermore, *T-Core* manipulates Himesis data structures. However in *MoTif*, the transformation is specified as a *DEVS* model—implicitly. At run-time, the `PythonDEVS` simulator involves atomic solvers (for atomic DEVS models), coordinators (for coupled DEVS models), and a root coordinator to execute the whole model. The interaction between these components was described in detail in Chapter 6.

As far as memory usage is concerned, the *Py-T-Core* solution uses up to 319 megabytes for the 500*th* round, while the *MoTif* solution uses up to 432 megabytes of memory. This difference is to be expected since the whole *DEVS* structure must be loaded in memory for a transformation implemented in *MoTif*.

11.4.3 Optimizations

We have tried to reduce the execution time by optimizing the transformation model only. That is, the optimizations performed do not rely on the code generator nor on the pattern matching algorithm, unlike other solutions [GZ10, MMLA10]. This allows a non-*MoTif* developer to carry out similar optimizations.

1. We define a pivot for every pattern element that is used in more than one rule. For example, declare an assigning pivot to the Ant[1], AntOnNode, and GridNode objects of every rule that makes a searching ant move. Then declare a binding pivot to these object in the rule `GrabFood`, so that its search space is reduced. Also, the rules involved in the CRule `CreateCircle` only bind the inner and outer nodes as pivots. Instead we should also bind the Forward and Right connections that are associated with them. The goal of this optimization is to reduce the search space of each rule applied in sequence by passing the output of a rule as input to the next rule in the sequence.
2. In Figure 11.4, the ARules `OnBorderGN` and `OnBorderMAN` are evaluated after all ants have moved. Instead, we could verify if an ant is on the border of the map as soon as an ant moves forward. In the first design, the rules must evaluate every grid node holding an ant. In the worst case, the rule will have to process all the nodes of the map. In the second design, all the elements involved in the pre-condition pattern are already bound by a previous rule that made the ant move. Therefore the search space of `OnBorderGN` and `OnBorderMAN` is reduced significantly, but the drawback is that it has to be evaluated as many times as there are ants in the model.
3. One can refactor the rules involved in a BRule to avoid matching the overlapping pattern more than once. The idea is to precede every BRule with an extra query corresponding to the overlap of the pre-condition pattern of each branch. In a BRule with *b* branches, the Matcher of each branch is applied as soon as a packet is received. In the current implementation, the pre-condition pattern is matched by all *b* matchers. However if the patterns overlap, an optimization could first pre-match the overlapping part. This is done by augmenting the *MoTif-Core* model corresponding to the BRule with an extra Matcher whose pre-condition pattern is the pattern overlapping with all branches. For example, the four rules involved in the `Move` BRule all have the ant, the grid node it is on, and the association between them in common. This can be computed in a similar way as the bridge for the overlap between the LHS and the NACs of a single rule. If this Matcher fails, then no Matcher from any branch can succeed and hence the BRule fails. Otherwise, the nodes of the current match is used as pivots for the subsequent Matchers of each branch. That way, the overlapped pattern is matched only once instead of *b* times.

We have incorporated all seven combinations of these optimizations, but no significant speed up was noticed: each “optimized” solution was less than 1% faster than the solution presented in Section 11.3, which is negligible given the error margin. This is mainly due to the overhead of adding more pivots for such small pre-condition patterns. Recall from Chapter 4 that in the implementation

[1]The Ant object is already assigned a pivot to correctly move it inside the loop of the `MoveSearchingAnts` block.

of the pattern matching in Himesis needed to convert graph nodes identified by a universally unique identifier (UUID) into the corresponding IGraph vertex index before matching and back afterwards. The reason why the implementation can not directly rely on these indices is because (1) when deleting a node all indices are affected and (2) the IGraph implementation may change in future releases.

11.5 Comparison With Other Solutions

The tool contest motivated many graph transformation tool builders to submit their solutions to the AntWorld case study. The task of addressing a "fair" tool-to-tool comparison for this benchmark is a hard task, since we are not experts of the transformation language of other tools and, most of all, they have radically different technological approaches on how the problem is meta-modelled and the rules are designed (*e.g.*, "compiled" transformation engines are typically directed towards different use cases than "interpreted" tools). Furthermore, the lack of standard and complete measurements of the different solutions prevents us to precisely align their performance. We will nevertheless briefly outline the solutions proposed by other tools.

FUJABA provided the fastest solution [GZ10]. The simulation reached 66.5×10^6 vertices[2] in a total of about 5.1×10^3 seconds after 1000 rounds. Recall that *FUJABA* is not a graph transformation tool but a CASE tool that allows for forward and reverse engineering, based on UML class diagrams. Nevertheless, the specification of the body of the methods inside each class is expressed in Story Charts, implemented as a graph transformation engine. By design, pre-condition pattern elements are always bound to either the formal parameters of the method, an object created by a previous rule inside the scope of the method, an attribute defined in the class encapsulating the method, or a reference to the current object (using the Java keyword `this`). This reduces the search space of each rule significantly. The meta-model of AntWorld proposed by the *FUJABA* solution is not as straightforward as the one proposed in this chapter. It incorporates a lot of information that is not mandatory to model the language. For example, neighbouring nodes are stored as an array of to-one associations instead of having a single to-many association. As mentioned in [GZ10], the authors have also adapted the code generator to optimize the execution time. All these optimizations require deep knowledge of the implementation of the tool. Our solution did not incorporate such optimizations since, in our opinion, this defeats the philosophy of MPM where the goal is to reduce accidental modelling complexity. Designing transformation models that rely on the implementation detail of the transformation language is not to be expected from the domain engineer, but the transformation language engineer.

The *VMTS* solution [MMLA10] also relies on implementation-aware optimizations. For example, the number of pattern elements in the rules is reduced by accessing the underlying data structure directly in the constraint and action code of the rules. In contrast, the constraint and action code in our solution is quite succinct and is therefore more intuitive for the modeller. Nevertheless, unlike *Py-T-Core*, *VMTS* implements pattern matching using sophisticated search plans [BKG08]. It introduces a cost-model for primitive matching operations and tries to minimize the overall cost of the execution of a rule by minimizing the possible backtracks when matching an element. The idea of the approach

[2]This corresponds to the number of Himesis vertices that would be required. It is computed using equation (11.4).

is to build a special *plan* graph for the pattern to be matched. The plan graph is a directed graph, whose nodes correspond to the elements (both nodes and edges) of the pattern graph and edges reflect the possible matching orders in the pattern graph. The 1000^{th} round had 1.25×10^6 vertices and was computed in 54 seconds.

The tool *GrGen.NET* provided a fast solution [JBK10], computing 1.1×10^6 vertices in 232 seconds for the 1000^{th} round. It implements similar search plans as *VMTS*. *GrGen.NET* is a graph transformation tool where the specification of the rules is in textual syntax. The power of its efficiency is attributed to the generative approach of adapting the search plan to the input graph. The rules are then compiled into a *concatenated* sequence of rules as described in [MG07].

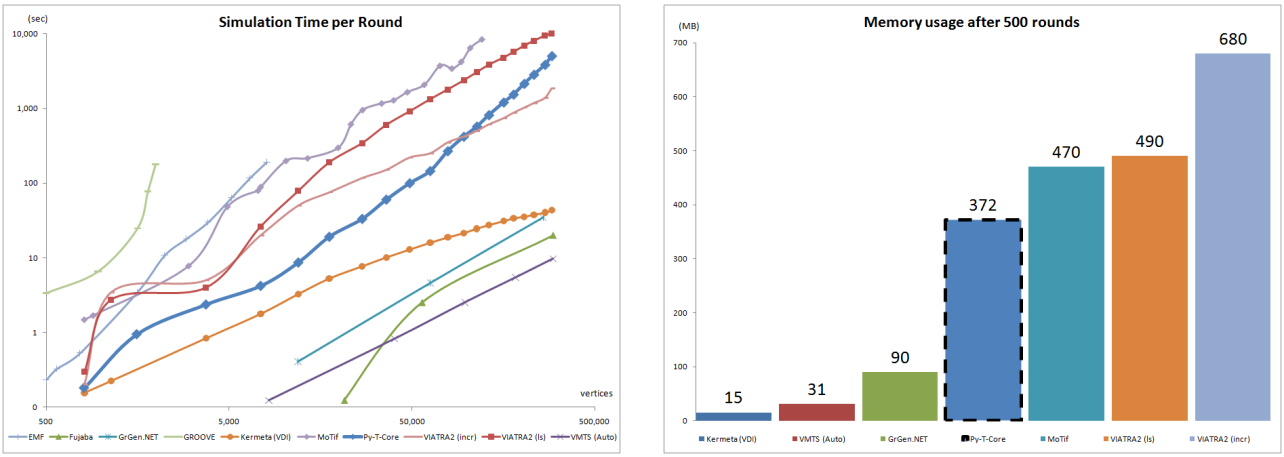
The previous tools proposed solutions using the graph transformation technology in a compiled approach. The following tools solve the benchmark in an interpreted approach[3]. The first one was implemented as an EMF transformation. However, the simulation was only measured up to 110 rounds with about 8×10^3 vertices taking about 192 seconds for this round. Our *Py-T-Core* solution is faster by 33 seconds for a similar graph.

Another solution was implemented in *VIATRA2* [HBRV10]. The authors solved the case study using the incremental transformation technique [BÖR+08]. This technique consists of caching the matches of a pattern for future rule applications. The match set is thus available from the cache at any time without having to perform further pattern matching. The cache is incrementally updated whenever changes are made to the model. This solution consumes about 20 seconds on the hundredth round for an input graph with 8×10^3 vertices. For this case study the incremental approach of *VIATRA2* is faster than its non-incremental approach with a polynomial order difference, but at the price of utilizing significantly more memory [HBRV10].

A solution using the tool *GROOVE* was also submitted [SR08]. The simulation results provided only showed measurements up to 100 rounds because the machine started swapping given the limited hardware. The hundredth round was computed in 178 seconds.

Figure 11.8(a) compares the execution time (per round) of the abovementioned solutions. The measurements were scaled appropriately to match the processor's speed as in Chapter 4. VIATRA2 (incr) refers to the incremental solution of *VIATRA2* and VIATRA (ls) refers to the non-incremental solution (local search). Additionally, we incorporated the performance results of a solution using *Kermeta* [MSF+10]. *Kermeta* is an object-oriented language that allows one to manipulate models directly through imperative code. For this case study, the authors have designed the meta-model in ECore and the simulation is encoded as operations. The meta-model is then augmented with these operations through aspect-oriented modelling techniques [KAAK09]. Since its execution is compiled, *Kermeta* outperforms the interpreted graph transformation approaches, although the compiled graph transformation approaches are still more efficient. Given the log-log scale of the graph in Figure 11.8(a), all curves depict a polynomial behaviour: *VMTS* seems to be the fastest while *GROOVE* the slowest. *Py-T-Core* seems to be the fastest interpreted solution until the 300^{th} round where the incremental solution of *VIATRA2* becomes faster.

[3] A solution implemented in an old version of *MoTif* was also submitted, but was erroneous.



(a) (b)

Figure 11.8: Time (a) and memory (b) performance measurements of all the solutions to the AntWorld benchmark.

Figure 11.8(b) shows the amount of memory required to run some of the solutions. The compiled solutions are those that consume the least memory. The *Py-T-Core* is situated in the middle with 372 megabytes of memory, like in the time performance case. Note that the *MoTif* solution requires 100 megabytes more because of the overhead of the *DEVS* structure and simulator. As expected, the incremental solution of *VIATRA2* requires the most amount of memory since it caches all the matches in memory.

11.6 Conclusion

In this chapter, we have implemented the AntWorld benchmark in both *Py-T-Core* and *MoTif*. On the one hand, this allowed us to quantify the overhead imposed by executing a transformation model in the *MoTif* framework over directly executing the transformation in Python. We discovered that the overhead of using a *DEVS* structure on top of Python induces a slow down by a factor 3. On the other hand, it allowed us to compare their performance with other model transformation approaches. This ensured us that the *MoTif* framework presented in Chapter 7 together with all the meta-layers involved (c.f. Chapter 5) does not affect the complexity class of the speed of execution of a completely modelled transformation language.

This thesis focuses on the expressiveness of model transformation rather than on performance. The performance results of *Py-T-Core* appearing in this chapter depict that, although it is not as efficient as existing tools, the performance of *Py-T-Core* is still in the range of performance of similar approaches (*e.g.*, interpreted). One possibility to improve the performance is to move to a compiled approach. This would however require a re-design of the run-time architecture of the framework. Another possibility is to implement the pattern matching using search plans rather than considering it as a constraint satisfaction problem.

When solving this case study, we have discovered several potential optimization points in the implementation of *MoTif*. For this case study, we have incorporated this optimization manually. In the future, we would like to automatically enhance the transformation model at compile-time or propose refactorings to the modeller at design-time. The *VMTS* solution [MMLA10] proposed an automatic way of refactoring the rules involved in a BRule to avoid matching the overlapping pattern more than once. Model transformation refactoring [MTR05, ZLG05] and design patterns [MLM10] has become an emerging research topic in the field.

Conclusions

Summary

Model transformation is at the centre of model-driven development efforts. It is therefore crucial that developers are equipped with well-founded model transformation languages. According to MPM principles, a model transformation must be modelled at the right level of abstraction, using the most appropriate formalism. Treating a model transformation language as a domain-specific language satisfies this rule. This thesis presented a framework to engineer model transformation languages following the MPM principles. This enables one to re-engineer existing transformation languages as well as to develop novel ones tailored to the specific domains and problems to be solved. In the following, the contributions of each chapter of the thesis are summarized.

Survey of Model Transformation

Establishing a framework for the design of model transformation languages requires a thorough analysis of existing languages, approaches, and paradigms. The review first focuses on the theory which has its roots in algebraic graph transformations. Then, a survey is presented of existing model transformation languages implemented as controlled graph transformation languages (such as *GReAT*, *ProGReS*, and *VIATRA2*), as model-to-model relations (such as *TGG* or *QVT*), and hybrid model transformations (such as *ATL*).

T-Core

From this survey, what is common in all model transformation approaches is extracted. The approach proposed in this thesis is to express model transformation at the level of their primitive building blocks. De-constructing and then re-constructing model transformation languages by means of a small set of most primitive constructs offers a common basis to compare the expressiveness of transformation languages. It may also help in the discovery of novel (possibly domain-specific) model transformation languages by combining the building blocks in new ways. Furthermore, it allows transformation language engineers to focus on maximizing the efficiency of the primitives in isolation, leading to more efficient transformations overall. *T-Core* is introduced as a collection of transformation language primitives for model transformation. It comprises primitive rule operations (such as *matching* and *rewriting*) and control-flow primitives (such as rule *selection* and *synchronization*). Inter- and intra-rule conflict detection and resolution are also available at the primitive level. This allows us to ensure a consistent application of rules executed in iteration or concurrently.

Systematic Development of Transformations

Despite the pivotal significance of transformations for model-driven approaches, there have not been any attempts to explicitly model transformation languages yet. In this thesis, a novel approach for the specification of transformations is presented, by treating model transformation languages as domain-specific languages. That is, for each pair of domains (the meta-models involved in the transformation), the meta-models of the rules are (semi-)automatically generated to create a language tailored to the transformation. This allows transformation developers to change the design of their transformation languages by modelling, rather than programming. Also, they may use environments to create transformations that are customized with respect to the input and output languages involved. The goal is to systematically support developers in creating transformation languages through the relaxation, augmentation, modification process.

Py-T-Core

T-Core is implemented as a module based on a model-centric virtual machine. The API is usable with a modelling language or a programming language. This "glue language" provides the scheduling of transformation units encapsulated in *T-Core*. *Py-T-Core* is the result of implementing *T-Core* in Python, thus making model transformation technologies available to programmers. From there, a transformation language engineer only needs to choose which *T-Core* primitives are necessary to define his custom transformation language.

MoTif

Py-T-Core is an example of how to use the model transformation language engineering framework when transformations are defined in programming languages. *MoTif* is a new model transformation language built using this framework. Here, transformations are defined in a modelling language. *MoTif* is the result of combining *T-Core* primitives with the discrete-event formalism *DEVS*. First, it is shown how the execution engine of a DEVS model can itself be modelled as a DEVS model. Then, execution of graph transformation control structures in *DEVS* is introduced. Thus graph transformation control structures are formalized by expressing them in terms of DEVS models. This is done by embedding the structure and the behaviour of *MoTif* constructs in terms of atomic and coupled DEVS models, and embed graphs in the events they exchange. This is therefore a contribution in both the MDE community and the discrete-event simulation community. Since all the components of the model transformation language *MoTif* are modelled explicitly, a transformation defined in this language is therefore a model conforming to the meta-model of *MoTif*. Higher-order transformation can then be applied on such transformation models.

Timed Model Transformations

Since *DEVS* is inherently a timed formalism, the notion of time in model transformation is explored. One may now model a time-advance for every rule as well as interrupt (pre-empt) rule execution. It is demonstrated how the explicit notion of time allows for the simulation-based design of the well-known Pacman game. Its dynamics is modelled with programmed graph transformation based on

DEVS. This also allows the modelling of player behaviour, incorporating data about human players' behaviour and reaction times. Thus, a model of both player and game is obtained which can be used to evaluate, through simulation, the playability of a game design. The case study proposes a playability performance measure and varies parameters of the Pacman game. For each variant of the game thus obtained, simulation yields a value for the quality of the game. This allows us to choose an "optimal" (from a playability point of view) game configuration.

Exception Handling in Model Transformation

An important aspect of model transformation which had not been investigated in the past is the notion of exception handling. This allows one to increase the dependability of transformation models. The different kinds of exceptions that can occur in model transformations were first analysed and classified. Some are more closely related to the execution environment. Some exceptions cannot be generalized to all transformation paradigms and thus are more transformation-language specific. A more subtle class of errors includes exceptions resulting from an inconsistent specification of transformation rules. The category of transformation-specific exceptions covers domain-specific, application-specific, and user-defined exceptions. The novelty of this work lies in the explicit *modelling* of exceptions and hence introducing *model* exception handling in the transformation language. For that, transformation rules are made exception-aware. The outcome of such rules is either a successfully transformed model (in case of a successful match and execution of the transformation), or an unmodified model (in case the rule is inapplicable on the model), or an exception (in case an exceptional situation occurred). Also, with appropriate control-flow support, the transformation modeller can directly specify how to handle all possible exceptions that can occur. Furthermore, the modeller can specify if the transformation should resume, restart, or terminate after an exception is handled.

Expressiveness and Performance Analysis

The CD2RDBMS case study was implemented to evaluate and compare the expressiveness of *MoTif* with existing model transformation languages. Additionally, the AntWorld case study provides a time and size performance comparison of *Py-T-Core* with *MoTif*, as well as with other model transformation tools. The results from these two benchmarks can be generalized: transformation languages produced by the framework presented in this thesis have a higher level of expressiveness and can perform better than some of the existing languages and tools.

Outlook

To conclude this thesis, follows an enumeration of what I think are the grand challenges that remain to be solved in model transformation to enable industrial adoption. The following are possible extensions to the work presented in this thesis.

Scalability. The first aspect of scalability is in terms of the **size of models** a transformation should be able to handle: models with 10^5 elements or 10^6 elements or more. A second aspect is scalability with respect to the **size of the transformation**. This can be measured in terms of the number of

rules applied on a given model or of the number of rules attempted at a given point during the execution of the transformation.

Interoperability. As in programming, one should be able to treat transformations as black-boxes while ensuring a meaningful and efficient communication between them. The work presented in [HKA10] is a step in this direction. Alternatively, we should investigate a common formalism that can serve as "bus" that serves as a communicating channel between transformation units, with plug-and-play behaviour. This requires a standardization of model/graph representation (such as GXL [Lam05]). We believe the modularity of *DEVS* may enable the development of such a bus.

Expressiveness. It is crucial to investigate what are the necessary and sufficient features a model transformation must be equipped with. *T-Core* can be used as a basis for the transformation units. The expressiveness of the pattern language and the scheduling language should also be considered.

Reversibility/Bidirectionality. With the advances of declarative transformation languages such as *QVT-R* and *TGG*, a single specification of the transformation can be executed from a source to a target model and vice versa. Handling arbitrarily complex attribute constraints is still under investigation, since the constraints may not be invertible. A link with non-causal languages, such as Modelica [Fri04], seems promising. Generalizing transformations to relate three or more modes is also an interesting challenge, with **multi-directional** transformations [KS06a]. Modular composition of relational models is another promising direction of research.

Analysis. The designer of a model transformation must be able to validate and verify the correctness of his transformation; property preservation of the input and output models must be ensured. **Formal verification** techniques [Flo67] allow verifying all behaviours of the transformation for any input model. **Model checking** techniques [CES82] restrict the verification to a fixed input model. When neither of these techniques is possible, model testing of the transformation can be used to test the transformation on a large set of generated input models. Alternatively, simulation of the transformation can be performed to calibrate, optimize, and validate a single behaviour of the transformation on a fixed input model (c.f. Chapter 9). These techniques should also validate the correct application of a transformation (at each step) and ensure that its properties are satisfied [BLA+10]. For example, critical pair analysis [MTR05] gives feedback to the modeller about possible conflicts between rules.

Testing. Properly testing model transformation is not a trivial task. On the one hand, one should identify adequate test inputs and how to assess the quality of models [MBLT06, SBM09]. On the other hand, one should investigate an adequate design of an oracle, be it generic, domain-specific, or transformation-specific.

Profiling. Profiling is an important activity in the development of software. The profiling results of a transformation should be provided in terms of, for example the *T-Core* primitives of a transformation or to its domain of application, instead of the generated code.

Debugging. Debugging the transformation should not rely on the code generated by the transformation. Instead, it should provide the appropriate support to the transformation designer to determine the origin of failure. The work on exception handling in this thesis is a first step in this direction. Nevertheless, debugging should also be provided in terms of the artefacts generated from the the transformation (*i.e.*, output or input model) as proposed in [MV10a].

Traceability. One should be able to trace back to the origin of an error. Properly tracing back to which element the created/modified element originated from is also desired. Tracing can be performed at different levels of a model transformation language. Tracing at the rule-level acts like a log of the rules that have been applied or not. Tracing at the level of primitive *T-Core* operation or at the level of matches is another possibility. One use of such traces is the inference of statistical measures on the execution of the transformation.

Evolution. Currently there is very little support to, in an evolving modelling language, automatically migrated the models to conform to the new version of the language [CDREP08, MV11]. For transformations, a first attempt at automatically migrating simple graph transformation rules was proposed in [ASWK11].

Although the abovementioned topics are important for industry, this thesis has already advanced the foundations of model transformation language engineering. I sincerely hope that further research will favour a close collaboration between research and industry in the area of model transformation and more generally in MDE.

List of Publications

[1] Eugene Syriani and Hans Vangheluwe. A Modular Timed Model Transformation Language. *Journal on Software and Systems Modeling*, (to appear), 2011.

[2] Márk Asztalos, Eugene Syriani, Manuel Wimmer, and Marouane Kessentini. Towards Rule Composition. *Journal of the Electronic Communications of the European Association of Software Science and Technology*, Oslo (Norway), October 2010.

[3] Eugene Syriani and Hans Vangheluwe. De-/Re-constructing Model Transformation Languages. *Journal of the Electronic Communications of the European Association of Software Science and Technology*, 29, March 2010.

[4] Eugene Syriani and Hans Vangheluwe. *DEVS as a Semantic Domain for Programmed Graph Transformation*. In Discrete-Event Modeling and Simulation: Theory and Applications, pages 3–28. CRC Press, December 2010.

[5] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Systematic Transformation Development. *Journal of the Electronic Communications of the European Association of Software Science and Technology*, 21, October 2009.

[6] Márk Asztalos, Eugene Syriani, Manuel Wimmer, and Marouane Kessentini. Simplifying Model Transformation Chains By Rule Composition. In *MODELS 2010 Workshops*, volume (to appear) of *LNCS*, Oslo (Norway), October 2011.

[7] Eugene Syriani, Jörg Kienzle, and Hans Vangheluwe. Exceptional Transformations. In Laurence Tratt and Martin Gogolla, editors, *ICMT'10*, volume 6142 of *LNCS*, pages 199–214, Màlaga (Spain), July 2010. Springer-Verlag.

[8] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Explicit Transformation Modeling. In Sudipto Ghosh, editor, *MODELS 2009 Workshops*, volume 6002 of *LNCS*, pages 240–255, Denver (USA), 2010. Springer.

[9] Eugene Syriani and Hans Vangheluwe. Using MoTif for the AntWorld Simulator Case Study. In Peiter Van Gorp and Arend Rensink, editors, *GraBaTs'08*, 2008.

[10] Eugene Syriani and Hans Vangheluwe. Programmed Graph Rewriting with Time for Simulation-Based Design. In Alfonso Pierantonio, Antonio Vallecillo, Jean Bézivin, and Jeff Gray, editors, *ICMT'08*, volume 5063 of *LNCS*, pages 91–106, Zürich (Switzerland), July 2008. Springer-Verlag.

[11] Eugene Syriani and Hans Vangheluwe. Programmed Graph Rewriting with DEVS. In Manfred Nagl and Andy Schürr, editors, *AGTIVE'07*, volume 5088 of *LNCS*, pages 136–152, Kassel (Germany), 2007. Springer-Verlag.

[12] Eugene Syriani, Jörg Kienzle, and Hans Vangheluwe. Performance Analysis of Himesis. Technical Report SOCS-TR-2010.8, McGill University, School of Computer Science, August 2010.

[13] Eugene Syriani and Hans Vangheluwe. A Modular Timed Model Transformation Language. Technical Report SOCS-TR-2010.4, McGill University, School of Computer Science, March 2010.

[14] Eugene Syriani, Hans Vangheluwe, and Amr Al-Mallah. Modelling and Simulation-based Design of a Distributed DEVS Simulator. Technical Report SOCS-TR-2010.3, McGill University, School of Computer Science, March 2010.

[15] Eugene Syriani, and Hans Vangheluwe. Exceptional Transformations. Technical Report SOCS-TR-2010.2, McGill University, School of Computer Science, January 2010.

[16] Eugene Syriani and Hans Vangheluwe. De-/Re-constructing Model Transformation Languages. Technical Report SOCS-TR-2009.8, McGill University, School of Computer Science, August 2009.

[17] Eugene Syriani and Hans Vangheluwe. Matters of model transformation. Technical Report SOCS-TR-2009.2, McGill University, School of Computer Science, March 2009.

Bibliography

[ABFL+09] Mathieu Anquetin, Pierre Baudemont, Thibault Franville-Lafargue, Emmanuel Navarro, and Guillaume Racineux. igraph Database Project. Project report, École nationale supérieure d'électronique, d'électrotechnique, d'informatique, d'hydraulique, et de télécommunications, Toulouse (France), March 2009.

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In Dorina Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *MoDELS'10*, volume 6394 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2010.

[AK07] Colin Atkinson and Thomas Kühne. *A tour of language customization concepts*, volume 70. Academic Press, Elsevier, June 2007.

[AKK+06] Aditya Agrawal, Gabor Karsai, Zsolt Kalmar, Sandeep Neema, Feng Shi, and Attila Vizhanyo. The Design of a Language for Model Transformations. *Journal on Software and Systems Modeling*, 5(3):261–288, September 2006.

[AKRS06] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In Arend Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066 of *LNCS*, pages 361–375. Springer-Verlag, 2006.

[ASWK11] Márk Asztalos, Eugene Syriani, Manuel Wimmer, and Marouane Kessentini. Simplifying Model Transformation Chains By Rule Composition. In *MODELS 2010 Workshops*, volume (to appear) of *LNCS*, Oslo (Norway), October 2011.

[ATL] ATL group. `http://www.eclipse.org/m2m/atl/atlTransformations/\#Class2Relational`.

[Bar95] Fernando J. Barros. Dynamic structure discrete event system specification: a new formalism for dynamic structure modeling and simulation. In Christos Alexopoulos, Keebom Kang, William R. Lilegdon, and David Goldsman, editors, *Winter Simulation (WSC'95)*, pages 781–785, Piscataway (USA), 1995. IEEE Computer Society.

[BBG+06] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. Model transformations? Transformation models! In *MoDELS'06*, volume 4199 of *LNCS*, pages 440–453, Genova (Italy), 2006. Springer Verlag.

[BCCT05] M Brambilla, S Ceri, S Comai, and C Tziviskou. Exception Handling in Workflow-Driven Web Applications. In *14th International World Wide Web Conference*, pages 170–180, Chiba (Japan), May 2005.

[BFG96] Dorothea Blostein, Hoda Fahmy, and Ann Grbavec. Issues in the Practical Use of Graph Rewriting. In Janice E. Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, *Selected papers from the 5th International Workshop on Graph Grammars and Their Application to Computer Science*, volume 1073 of *LNCS*, pages 38–55, Williamsburg (USA), November 1996. Springer-Verlag.

[BFJ+03] Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benoît Langlois, and Damien Pollet. Reflective model driven engineering. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML'03*, volume 2863 of *LNCS*, pages 175–189. Springer, 2003.

[BJ66] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, May 1966.

[BKG08] Gernot Veit Batz, Moritz Kroll, and Rubino Geiß. A First Experimental Evaluation of Search Plan Driven Graph Pattern Matching. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *LNCS*, pages 471–486. Springer, 2008.

[BLA+10] Bruno Barroca, Levi Lúcio, Vasco Amaral, Roberto Felix, and Vasco Sousa. DSLTrans: A Turing Incomplete Transformation Language. In *SLE'10*, LNCS, Eindhoven (The Netherlands), October 2010. Springer.

[BN99] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge Univ Press, 1999.

[BNN+07] Daniel Balasubramanian, Anantha Narayanan, Sandeep Neema, Benjamin Ness, Feng Shi, Ryan Thibodeaux, and Gabor Karsai. Applying a Grouping Operator in Model Transformations. In Manfred Nagl and Andy Schürr, editors, *AGTIVE'07*, volume 5088 of *LNCS*, pages 410–425, Kassel (Germany), 2007. Springer-Verlag.

[BÖ10] Artur Boronat and Peter Ölveczky. Formal Real-Time Model Transformations in MOMENT2. In David Rosenblum and Gabriele Taentzer, editors, *Fundamental Approaches to Software Engineering*, volume 6013 of *LNCS*, pages 29–43. Springer, 2010.

[BÖR+08] Gábor Bergmann, András Ökrös, István Ráth, Dániel Varró, and Gergely Varró. Incremental Pattern Matching in the VIATRA Model Transformation System. In *GRaMot'08*, 2008.

[BRT05] Jean Bézivin, Bernhard Rumpe, and Laurence Tratt. Model Transformation in Practice Workshop Announcement. `http://sosym.dcs.kcl.ac.uk/events/mtip05/long\_cfp.pdf`, 2005.

[BS99] Dorothea Blostein and Andy Schürr. Computing with Graphs and Graph Rewriting. *Software - Practice & Experience*, 9(3):1–21, 1999.

[BV01] Jean-Sébastien Bolduc and Hans Vangheluwe. The Modelling and Simulation Package pythonDEVS for Classical Hierarchical DEVS. MSDL technical report msdl-tr-2001–01, McGill University, June 2001.

[BV06] András Balogh and Dániel Varró. Pattern Composition in Graph Transformation Rules. In *European Workshop on Composition of Model Transformations*, Bilbao (Spain), July 2006.

[CA78] Liming Chen and Algirdas Avizienis. N-Version Programming: A Fault Tolerance Approach to Reliability of Software Operation. In *Fault-Tolerant Computing 1995, Highlights from Twenty-Five Years*, pages 3–9, Toulouse (France), 1978. IEEE Computer Society Press.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.

[CDREP08] Antonio Cicchetti, Davide Di Ruscio, R Eramo, and Alfonso Pierantonio. Automating co-evolution in model-driven engineering. In *EDOC'08*, pages 222–231. IEEE Computer Society, September 2008.

[CES82] Edmund Melson Clarke, Ernest Allen Emerson, and Joseph Sifakis. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs Workshop*, volume 131 of *LNCS*, pages 52–71, London (U.K.), May 1982. Springer-Verlag.

[CFSV04] Luidgi Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions On Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, October 2004.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal, special issue on Model-Driven Software Development*, 45(3):621–645, July 2006.

[Cho51] Noam Chomsky. On the notion rule of grammar. *American Mathematical Society*, 12:6–24, 1951.

[CI84] Robert D. Cameron and M. Robert Ito. Grammar-Based Definition of Metaprogramming Systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(1):20–54, January 1984.

[CLR00] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 2000.

[CN06] Gábor Csárdi and Tamás Nepusz. The igraph software package for complex network research. *InterJournal Complex Systems*, 1695, 2006.

[CSPZ04] Saehoon Cheon, Chungman Seo, Sunwoo Park, and Bernard P. Zeigler. Design and Implementation of Distributed DEVS Simulation in a Peer to Peer Network System. In Herwing Unger, editor, *ASTC'04*, pages 18–22, Arlington (USA), April 2004. Society for Modeling and Simulation International.

[CZ96] Alex Chung-Hen Chow and Bernard P. Zeigler. Parallel DEVS: a parallel, hierarchical, modular modeling formalism. *Transactions of the Society for Computer Simulation International*, 13:55–67, 1996.

[Dah02] Ole-Johan Dahl. The roots of object orientation: the Simula language. *Software pioneers*, pages 78–90, 2002.

[Dev86] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York (USA), 1986.

[DHP02] Frank Drewes, Berthold Hoffmann, and Detelf Plump. Hierarchical Graph Transformation. *Journal of Computer and System Sciences*, 64:249–283, 2002.

[DJK+99] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *International Conference on Software Engineering (ICSE'99)*, pages 285–294, Los Angeles (USA), May 1999. ACM Press.

[dLBE+07] Juan de Lara, Roswitha Bardohlb, Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Attributed graph transformation with node type inheritance. *Theoretical Computer Science Journal*, 376(3):139–163, February 2007.

[dLG08] Juan de Lara and Esther Guerra. Pattern-based Model-to-Model Transformation. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *ICGT'08*, volume 5214 of *LNCS*, pages 427–441, Leicester (UK), September 2008.

[dLV02] Juan de Lara and Hans Vangheluwe. AToM^3: A Tool for Multi-formalism and Meta-Modelling. In Ralf-Detlef Kutsche and Herbert Weber, editors, *FASE'02*, volume 2306 of *LNCS*, pages 174–188, Grenoble (France), April 2002. Springer-Verlag.

[dLV04] Juan de Lara and Hans Vangheluwe. Defining visual notations and their manipulation through meta-modelling and graph transformation. *Journal of Visual Languages and Computing*, 15(3–4):309–330, June 2004.

[dLV10] Juan de Lara and Hans Vangheluwe. Automating the transformation-based analysis of visual languages. *Formal Aspects of Computing*, 22(3-4):297–326, May 2010.

[Don90] Christophe Dony. Exception Handling and Object-Oriented Programming: Towards a Synthesis. In Norman Meyrowitz, editor, *ECOOP'90*, volume 25 of *ACM SIGPLAN Notices*, pages 322–330. ACM Press, 1990.

[Dvo08] Radomil Dvorak. Model transformation with Operational QVT. `http://www.eclipse.org/m2m`, March 2008.

[EEKR97] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg. *Handbook of graph grammars and computing by graph transformation, Volume 1: Foundations*. World Scientific Publishing Co., Inc., 1997.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS. Springer-Verlag, 2006.

[EETW06] Claudia Ermel, Karsten Ehrig, Gabriele Taentzer, and Eduard Weiss. Object Oriented and Rule-based Design of Visual Languages using Tiger. In Albert Zündorf and Dániel Varró, editors, *GraBaTs'06*, volume 1 of *ECEASST*, pages 1–13, Natal (Brazil), September 2006.

[EKT09] Karsten Ehrig, Jochen Küster, and Gabriele Taentzer. Generating instance models from meta models. *Journal on Software and Systems Modeling*, 8:479–500, 2009.

[EPK06] Klaus-D. Engel, Richard Paige, and Dimitrios Kolovos. Using a Model Merging Language for Reconciling Model Versions. In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture-Foundations and Applications*, volume 4066 of *LNCS*, pages 143–157. Springer, 2006.

[EPT04] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental Theory for Typed Attributed Graph Transformation. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *ICGT'04*, volume 3256 of *LNCS*, pages 161–177, Rome (Italy), September 2004. Springer-Verlag.

[FB04] Jean-Baptiste Filippi and Paul Bisgambiglia. JDEVS: an implementation of a DEVS based formal framework for environmental modelling. *Environmental Modelling and Software*, 19(3):261–274, 2004.

[FHN06] Jean-Rémy Falleri, Marianne Huchard, and Clémentine Nebut. Towards a Traceability Framework for Model Transformations in Kermeta. In J Aagedal, T Neple, and J Oldevik, editors, *ECMDA-TW'06*, pages 31–40, Bilbao (Spain), July 2006. HAL-CCSd-CNRS.

[Flo67] Robert W Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19:19–32, 1967.

[FNTZ00] Thorsten Fischer, Jörg Niere, Lars Turunski, and Albert Zündorf. Story diagrams: A new graph rewrite language based on the Unified Modelling Language and Java. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Theory and Application of Graph Transformations*, volume 1764 of *LNCS*, pages 296–309, Paderborn (Germany), November 2000. Springer-Verlag.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Fra08] France Telecom R&D. SmartQVT. `http://smartqvt.elibel.tm.fr`, August 2008.

[Fri04] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, January 2004.

[FSV01] Pasquale Foggia, Carlo Sansone, and Mario Vento. A Database of Graphs for Isomorphism and Sub Graph Isomorphism Benchmarking. In Jean-Michel Jolion, Walter Kropatsch, and Vento Mario, editors, *Workshop on Graph-Based Representation in Pattern Recognition*, IAPR-TC15, pages 176–188, Ischia (Italy), May 2001.

[Ful10] Brent Fulgham. Computer Language Benchmarks Game. `http://shootout.alioth.debian.org/u32q/benchmark.php?test=all\&lang=python\&lang2=gcc`, August 2010.

[GBG+06] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *ICGT'06*, volume 4178 of *LNCS*, pages 383–397, Heidelberg (Germany), September 2006. Springer-Verlag.

[GdL04] Esther Guerra and Juan de Lara. Event-Driven Grammars: Towards the Integration of Meta-modelling and Graph Transformation. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *ICGT'04*, volume 3256 of *LNCS*, pages 54–69. Springer-Verlag, 2004.

[GdL06] Esther Guerra and Juan de Lara. Model View Management with Triple Graph Transformation Systems. In *ICGT'06*, volume 4178 of *LNCS*, pages 351–366. Springer-Verlag, 2006.

[GdL07a] Esther Guerra and Juan de Lara. Adding Recursion to Graph Transformation. In Karsten Ehrig and Holger Giese, editors, *GT-VMT 2007*, volume 6 of *ECEASST*, Braga (Portugal), 2007.

[GdL07b] Esther Guerra and Juan de Lara. Event-Driven Grammars: Relating Abstract and Concrete Levels of Visual Languages. *Journal on Software and Systems Modeling*, 6(6):317–347, 2007.

[GGKH03] Tracy Gardner, Catherine Griffin, Jana Koehler, and Rainer Hauser. A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final standard. In *1st MetaModelling for MDA Workshop*, pages 178–197, York (UK), July 2003.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. AddisonWesley Professional, November 1994.

[GHV02] Szilvia Gyapay, Reiko Heckel, and Dániel Varró. Graph Transformation with Time: Causality and Logical Clocks. In *ICGT'02*, volume 2505 of *LNCS*, pages 120–134, Barcelona (Spain), October 2002. Springer-Verlag.

[Goo75] John B. Goodenough. Exception Handling: Issues and a Proposed Notation. *Communications of the ACM*, 18(12):683–696, December 1975.

[Gri91] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, August 1991.

[GTK+07] Jeff Gray, Juha-Pekka Tolvanen, Aniruddha Kelly, Steven Gokhale, Sandeep Neema, and Jonathan Sprinkle. *Domain-Specific Modeling*, chapter chapter 7, pages 1–20. CRC Press, 2007.

[GvD07] Bas Graaf and Arie van Deursen. Using MDE for generic comparison of views. In *Models in Software Engineering*, Nashville (USA), October 2007. INRIA.

[GZ10] Leif Geiger and Albert Zündorf. Fujaba case studies for GraBaTs 2008: lessons learned. *International Journal on Software Tools for Technology Transfer*, 12:287–304, 2010.

[HBJ09] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. Cope - Automating Coupled Evolution of Metamodels and Models. In Sophia Drossopoulou, editor, *23rd European Conference on Object-Oriented Programming (ECOOP)*, volume 5653 of *LNCS*, pages 52–76, Genova (Italy), July 2009. Springer-Verlag.

[HBRV10] 'Akos Horváth, Gábor Bergmann, István Ráth, and Dániel Varró. Experimental assessment of combining pattern matching strategies with VIATRA2. *International Journal on Software Tools for Technology Transfer*, 12:211–230, 2010.

[Hec06] Reiko Heckel. Graph Transformation in a Nutshell. In *Proceedings of the School on Foundations of Visual Modelling Techniques (FoVMT 2004) of the SegraVis Research Training Network*, volume 148 of *ENTCS*, pages 187–198. Elsevier, 2006.

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3–4):287 – 313, June 1996.

[HHT02] Jan Hendrik Hausmann, Reiko Heckel, and Gabriele Taentzer. Detection of Conflicting Functional Requirements in a Use Case-Driven Approach. In *ICSE'02*, pages 105–115, Orlando (USA), May 2002. ACM.

[HKA10] Florian Heidenreich, Jan Kopcsek, and Uwe Assmann. Safe Composition of Transformation. In Laurence Tratt and Martin Gogolla, editors, *ICMT'10*, volume 6142 of *LNCS*, pages 108–122, Màlaga (Spain), July 2010. Springer-Verlag.

[HKT02] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of Typed Attributed Graph Transformation Systems. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *ICGT 2002*, volume 2505 of *LNCS*, pages 161–176, Barcelona (Spain), October 2002. Springer-Verlag.

[HLM04] Reiko Heckel, Georgios Lajios, and Sebastian Menge. Stochastic Graph Transformation Systems. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *ICGT'04*, volume 3256 of *LNCS*, pages 243–246. Springer, 2004.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(3):100–107, July 1968.

[HR00] David Harel and Bernhard Rumpe. Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff. Technical report, Weizmann Institute Of Sience, 2000.

[HSKP97] Joon Sung Hong, Hae-Sang Song, Tag Gon Kim, and Kyu Ho Park. A Real-Time Discrete Event System Specification Formalism for Seamless Real-Time Software Development. *Discrete Event Dynamic Systems*, 7:355–375, 1997.

[HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In Gäel Varoquaux, Travis Vaught, and Jarrod Millman, editors, *SciPy'08*, pages 11–15, Pasadena (USA), August 2008.

[Hun05] Robin Hunicke. The Case for Dynamic Difficulty Adjustment in Games. In *Proccedings of the ACM SIGCHI International Conference on Advances in Computer Entertainment technology*, pages 429–433, Valencia,Spain, 2005. ACM.

[igr09] igraph.sourceforge.net. Igraph Library v0.5.3, November 2009.

[JABK08] Frédérick Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming, Special Issue on Second issue of experimental software and toolkits (EST)*, 72(1-2):31–39, June 2008.

[JBK10] Edgar Jakumeit, Sebastian Buchwald, and Moritz Kroll. GrGen.NET: The expressive, convenient and fast graph rewrite system. *International Journal on Software Tools for Technology Transfer*, 12:263–271, 2010.

[JK06] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *MTiP'05*, volume 3844 of *LNCS*, pages 128–138. Springer-Verlag, January 2006.

[Jou05] Frédéric Jouault. Loosely coupled traceability for ATL. In *ECMDA Workshop on Traceability*, 2005.

[Jou06] Frédéric Jouault. *Contribution à l'étude des langages de transformation de modèles*. Ph.D. thesis, Université de Nantes, September 2006.

[KAAK09] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-Oriented Multi-View Modeling. In *AOSD'09*, pages 87–98, Charlottesville (USA), March 2009. ACM Press.

[KBC05] Audris Kalnins, Janis Barzdins, and Edgars Celms. Model Transformation Language MOLA. In Uwe Assmann, Mehmet Aksit, and Arend Rensink, editors, *ECMDA-FA'05*, volume 3599 of *LNCS*, pages 62–76, Nuremberg (Germany), November 2005. Springer.

[KH04] Evgeny B. Krissinel and Kim Henrick. Common subgraph isomorphism detection by backtracking search. *Software - Practice & Experience*, 34(6):591–607, 2004.

[Kil90] Haim Kilov. From semantic to object-oriented data modeling. In *First International Conference on System Integration*, pages 385–393, 1990.

[KKS07] Felix Klar, Alexander Königs, and Andy Schürr. Model transformation in the large. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 285–294, Dubrovnik (Croatia), 2007. ACM.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira-Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Askit and Satoshi Matsuoka, editors, *ECOOP*, volume 1241 of *LNCS*, pages 220–242, Jyväskylä (Finland), June 1997. Springer-Verlag.

[KMS+09] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Systematic Transformation Development. *Electronic Communications of the European Association of Software Science and Technology*, 21, October 2009.

[KMS+10] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Explicit Transformation Modeling. In Sudipto Ghosh, editor, *MODELS 2009 Workshops*, volume 6002 of *LNCS*, pages 240–255, Denver (USA), 2010. Springer.

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August 1988.

[KS06a] Alexander Königs and Andy Schürr. MDI: A Rule-based Multi-document and Tool Integration Approach. *Journal on Software and Systems Modeling*, 5(20):349–368, December 2006.

[KS06b] Alexander Königs and Andy Schürr. Tool Integration with Triple Graph Grammars - A Survey. In Reiko Heckel, editor, *Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques*, volume 148 of *ENTCS*, pages 113–150, Amsterdam (Netherlands), 2006. Elsevier Science Publishers.

[Küh06a] Thomas Kühne. Clarifying Matters of (Meta-)Modeling. *Journal on Software and Systems Modeling*, 5(4):395–401, December 2006.

[Küh06b] Thomas Kühne. Matters of (Meta-)Modeling. *Journal on Software and Systems Modeling*, 5(4):369–385, December 2006.

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained. The Model Driven Architecture: Practice And Promise*. Addison-Wesley, 2003.

[KWSB10] Marouane Kessentini, Manuel Wimmer, Houari Sahraoui, and Mounir Boukadoum. Generating transformation rules from examples for behavioral models. In *Workshop on Behaviour Modelling: Foundation and Applications (BM-FA '10)*, pages 21–27, New York, NY, USA, 2010. ACM.

[Lam05] Leen Lambers. A New Version of GTXL : An Exchange Format for Graph Transformation Systems. In *GraBaTs'04*, volume 127 of *ENTCS*, pages 51–63, March 2005.

[Lan66] Peter J. Landin. The next 700 programming languages. *Communications of ACM*, 9(3):157–166, March 1966.

[LEO08] Leen Lambers, Hartmut Ehrig, and Fernando Orejas. Efficient Conflict Detection in Graph Transformation Systems by Essential Critical Pairs. In Roberto Bruni and Dániel Varró, editors, *GT-VMT'06*, volume 211 of *ENTCS*, pages 17–26, Vienna (Austria), April 2008.

[LLC04] Jong-Keun Lee, Ye-Hwan Lim, and Sung-Do Chi. Hierarchical Modeling and Simulation Environment for Intelligent Transportation Systems. *Simulation*, 80(2):61–76, February 2004.

[LLC05] Tihamér Levendovszky, Lázló Lengyel, and Hassan Charaf. A UML class diagram-based pattern language for model transformation systems. In Walter Dosch, Rudolf Freund, and Nikos Mastorakis, editors, *WSEAS'05*, volume 4 of *WSEAS Transactions on Computers*, pages 190–195, Salzburg (Austria), February 2005. World Scientific and Engineering Academy and Society.

[LLM09] Tihamér Levendovszky, Lázló Lengyel, and Tamás Mészáros. Supporting domain-specific model patterns with metamodeling. *Journal on Software and Systems Modeling*, 8(4):501–520, September 2009.

[LLMC05] László Lengyel, Tihamér Levendovszky, Gergely Mezei, and Hassan Charaf. Control Flow Support in Metamodel-Based Model Transformation Frameworks. In *EUROCON'05*, pages 595–598, Belgrade (Serbia), November 2005. IEEE.

[LLMC06] László Lengyel, Tihamér Levendovszky, Gergely Mezei, and Hassan Charaf. Model Transformation with a Visual Control Flow Language. *International Journal of Computer Science*, 1(1):45–53, 2006.

[LS06] Michael Lawley and Jim Steel. Practical Declarative Model Transformation with Tefkat. In Jean-Marc Bruel, editor, *Satellite Events at the MoDELS'05 Conference*, volume 3844 of *LNCS*, pages 139–150, Montego Bay (Jamaica), October 2006. Springer-Verlag.

[MBJR07] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, and Rodrigo Ramos. Towards a Generic Aspect-Oriented Modeling Framework. In *Models and Aspects workshop, at ECOOP'07*, Berlin (Germany), July 2007.

[MBLT06] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Mutation Analysis Testing for Model Transformations. In *ECMDA-FA*, pages 376–390, 2006.

[MBP99] Peter Mc Brien and Alexandra Poulovassi. Automatic Migration and Wrapping of Database Applications - A Schema Transformation Approach. In Jacky Akoka, Mokrane Bouzeghoub, Isabelle Comyn-Wattiau, and Elisabeth M'etais, editors, *Conceptual Modeling ER'99*, volume 1782 of *LNCS*, pages 99–114, London (UK), 1999. Springer-Verlag.

[Meh84] Kurt Mehlhorn. *Graph Algorithms and NP-Completeness*, volume 2 of *Monographs in Theoretical Computer Science. An EATCS Series*. Springer, 1984.

[MG07] Jens Müller and Rubino Geiß. Speeding up Graph Transformation through Automatic Concatenation of Rewrite Rules. Technical Report 76131, Universität Karlsruhe, Karlsruhe (Germany), 2007.

[Min06] Mark Minas. Generating meta-model-based freehand editors. *Electronic Communications of the European Association of Software Science and Technology*, 1:1–13, September 2006.

[MLM10] Tamás Mészáros, Tihamér Levendovszky, and Gergely Mezei. Active Model Patterns with Interactive Model Transformation. In *Proceeding of the Multi-Paradigm Modelling Workshop (MPM'10)*, ECEASST, 2010.

[MMLA10] Tamás Mészáros, Gergely Mezei, Tihamér Levendovszky, and Márk Asztalos. Manual and automated performance optimization of model transformation systems. *International Journal on Software Tools for Technology Transfer*, 12:231–243, 2010.

[MOL] MOLA. `http://mola.mii.lu.lv/mola_examples_CDtoRDBrec.html`.

[MP08] Greg Manning and Detelf Plump. The GP Programming System. In Claudia Ermel, Reiko Heckel, and Juan de Lara, editors, *GT-VMT'08*, volume 10 of *ECEASST*, pages 235–247, Budapest (Hungary), March 2008.

[MSF+10] Naouel Moha, Sagar Sen, Cyril Faucher, Olivier Barais, and Jean-Marc Jézéquel. Evaluation of Kermeta for solving graph-based problems. *International Journal on Software Tools for Technology Transfer*, 12:273–285, 2010.

[MSVG05] Olaf Muliawan, Hans Schippers, and Pieter Van Gorp. Model driven, Template based, Model Transformer (MoTMoT). `http://motmot.sourceforge.net`, 2005.

[MTR05] Tom Mens, Gabriele Taentzer, and Olga Runge. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. In Reiko Heckel and Tom Mens, editors, *SETra'04)*, volume 127 of *ENTCS*, pages 113–128, Rome (Italy), April 2005. Elsevier.

[MV04] Pieter J. Mosterman and Hans Vangheluwe. Computer Automated Multi-Paradigm Modeling: An Introduction. *Simulation: Transactions of The Society for Modeling and Simulation International*, 80(9):433–450, September 2004.

[MV10a] Raphael Mannadiar and Hans Vangheluwe. Debugging in Domain-Specific Modelling. In *SLE'10*, 2010.

[MV10b] Raphael Mannadiar and Hans Vangheluwe. Modular Synthesis of Mobile Device Applications from Domain-Specific Models. In *7th Model-based Methodologies for Pervasive and Embedded Software workshop*, 2010.

[MV11] Bart Meyers and Hans Vangheluwe. A Framework for Evolution of Modelling Languages. *Science of Computer Programming*, (in press), 2011.

[MVG06] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. In *GraMoT'05*, volume 152 of *ENTCS*, pages 125–142, Tallinn (Estonia), March 2006.

[net10] networkx.lanl.gov. NetworkX v1.1. `http://networkx.lanl.gov`, April 2010.

[NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *ICSE'00*, pages 742–745, Limerick (Ireland), June 2000. ACM Press.

[NPC01] Raquel Navarro-Prieto and Jose J. Cañas. Are visual programming languages better? The role of imagery in program comprehension. *International Journal of Human-Computer Studies*, 54(6):799–829, June 2001.

[Obj03] Object Management Group. *MDA Guide Version 1.0.1*, June 2003.

[Obj04] Object Management Group. Joint revised submission for the QVT RFP. `http://www.omg.org/cgi-bin/doc?ad/2004-04-01`, April 2004.

[Obj06a] Object Management Group. *Meta Object Facility 2.0 Core Specification*, January 2006.

[Obj06b] Object Management Group. *Object Constraint Language*, May 2006.

[Obj08] Object Management Group. *Meta Object Facility 2.0 Query/View/Transformation Specification*, April 2008.

[Obj09] Object Management Group. *Unified Modeling Language Superstructure*, 2.2 edition, February 2009.

[PM05] Gergely Pintér and István Majzik. Modeling and Analysis of Exception Handling by Using UML Statecharts. In Gianna Guelfi, Nicolas Reggio and Alexander Romanovsky, editors, *FIDJI'04)*, volume 3409 of *LNCS*, pages 58–67, Luxembourg-Kirchberg (Luxembourg), November 2005. Springer.

[Pra71] Terrence W. Pratt. Pair Grammars, Graph Languages and String-to-Graph Translations. *Journal of Computer and System Sciences*, 5(6):560–595, 1971.

[Pro05] Marc Provost. Himesis: A Hierarchical Subgraph Matching Kernel for Model Driven Development. Master's thesis, McGill University, Montréal (Canada), 2005.

[Pro10a] Eclipse Modeling Project. ATL Transformations. `http://www.eclipse.org/m2m/atl/atlTransformations`, 2010.

[Pro10b] Eclipse Modeling Project. Xpand. `http://wiki.eclipse.org/Xpand`, August 2010.

[RBÖV08] István Ráth, Gábor Bergmann, András Ökrös, and Dániel Varró. Live Model Transformations Driven by Incremental Pattern Matching. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *ICMT'08*, volume 5063 of *LNCS*, pages 107–121. Springer, 2008.

[Ren04] Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In J. Pfalz, Manfred Nagl, and B. Böhlen, editors, *AGTIVE'03)*, volume 3062 of *LNCS*, pages 479–485. Springer-Verlag, 2004.

[RK09] Arend Rensink and Jan-Hendrik Kuperus. Repotting the Geraniums: On Nested Graph Transformation Rules. In Tiziana Margaria, Julia Padberg, and Gabriele Taentzer, editors, *GT-VMT'09*, volume 18 of *ECEASST*, York (UK), March 2009.

[RKR+06] Thomas Reiter, Elizabeth Kapsammer, Werner Retschitzegger, Wieland Schwinger, and Markus Stumptner. A Generator Framework for Domain-Specific Model Transformation Languages. In *International Conference on Enterprise Information Systems (ICEIS'06)*, volume 3, pages 27–35, Paphos (Cyprus), May 2006. INSTICC.

[Rud98] Michael Rudolf. Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *TAGT'98, Selected Papers*, volume 1764 of *LNCS*, pages 381–394, Paderborn (Germany), November 1998. Springer.

[RX95] Brian Randell and Jie Xu. *Software Fault Tolerance*, chapter The Evolution of the Recovery Block Concept, pages 1–25. John Wiley & Sons Ltd, 1995.

[SBM09] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. Automatic Model Generation Strategies for Model Transformation Testing. In Richard Paige, editor, *ICMT'09*, volume 5563 of *LNCS*, pages 148–164. Springer, 2009.

[Sch94] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In Gottfried Tinhofer, editor, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *LNCS*, pages 151–163, Heidelberg (Germany), June 1994. Springer-Verlag.

[SK04] Jonathan Sprinkle and Gabor Karsai. A Domain-Specific Visual Language for Domain Model Evolution. *Journal of Visual Languages and Computing*, 15(3-4):291–307, April 2004.

[SK08] Andy Schürr and Felix Klar. 15 Years of Triple Graph Grammars. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *ICGT'08*, volume 5214 of *LNCS*, pages 411–425, Leicester (UK), September 2008.

[SKHP07] Park Sunwoo, Sean H. J. Kim, C. Anthony Hunt, and Dongsun Park. DEVS Peer-to-Peer Protocol for Distributed and Parallel Simulation of Hierarchical and Decomposable DEVS Models. In Hyongsuk Kim and Benjamin Wah, editors, *ISITC'07*, pages 91–95, Jeonju (Korea), November 2007. IEEE Computer Society.

[SKV10] Eugene Syriani, Jörg Kienzle, and Hans Vangheluwe. Exceptional Transformations. In Laurence Tratt and Martin Gogolla, editors, *ICMT'10*, volume 6142 of *LNCS*, pages 199–214, Màlaga (Spain), July 2010. Springer-Verlag.

[SPB+04] Chungman Seo, Sunwoo Park, Kim Byounguk, Saehoon Cheon, and Bernard P. Zeigler. Implementation of Distributed high-performance DEVS Simulation Framework in the Grid Computing Environment. In Herwing Unger, editor, *ASTC'04*, Arlington (USA), April 2004. Society for Modeling and Simulation International.

[SR08] Tom Staijen and Arend Rensink. A GROOVE Solution for the GraBaTs'08 AntWorld Case. `http://www.fots.ua.ac.be/events/grabats2008/submissions/grabats2008_submission_20.pdf`, September 2008.

[SV07] Eugene Syriani and Hans Vangheluwe. Programmed Graph Rewriting with DEVS. In Manfred Nagl and Andy Schürr, editors, *AGTIVE'07*, volume 5088 of *LNCS*, pages 136–152, Kassel (Germany), 2007. Springer-Verlag.

[SV08a] Eugene Syriani and Hans Vangheluwe. Programmed Graph Rewriting with Time for Simulation-Based Design. In Alfonso Pierantonio, Antonio Vallecillo, Jean Bézivin, and Jeff Gray, editors, *ICMT'08*, volume 5063 of *LNCS*, pages 91–106, Zürich (Switzerland), July 2008. Springer-Verlag.

[SV08b] Eugene Syriani and Hans Vangheluwe. Using MoTif for the AntWorld Simulator Case Study. In Peiter Van Gorp and Arend Rensink, editors, *GraBaTs'08*, 2008.

[SV09] Eugene Syriani and Hans Vangheluwe. Matters of model transformation. Technical Report SOCS-TR-2009.2, McGill University, School of Computer Science, March 2009.

[SV10] Eugene Syriani and Hans Vangheluwe. *DEVS as a Semantic Domain for Programmed Graph Transformation*, chapter 1, pages 3–28. CRC Press, Boca Raton (USA), December 2010.

[SV11] Eugene Syriani and Hans Vangheluwe. A Modular Timed Model Transformation Language. *Journal on Software and Systems Modeling*, (to appear), 2011.

[SVC06] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development – Technology, Engineering, Management*. John Wiley & Sons, 2006.

[SWZ95] Andy Schürr, Andreas J. Winter, and Albert Zündorf. Graph Grammar Engineering with PROGRES. In Wilhelm Schäfer and Pere Botella, editors, *5th European Software Engineering Conference*, volume 989 of *LNCS*, pages 219–234, Sitges,Spain, September 1995. Springer-Verlag.

[Syr10] Eugene Syriani. T-Core. `http://msdl.cs.mcgill.ca/people/eugene/motif/tcore.zip`, October 2010.

[Sys00] Sparx Systems. Enterprise Architect. `http://www.sparxsystems.com.au/`, 2000.

[Tae04] Gabriele Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *AGTIVE'03*, volume 3062 of *LNCS*, pages 446–453. Springer-Verlag, 2004.

[TEG+05] Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan de Lara, Laszló Lengyel, Tihamér Levendovszky, Ulrike Prange, Dániel Varró, and Szilvia Varró-Gyapay. Model Transformation by Graph Transformation: A Comparative Study. In *MTiP'05*, Montego Bay (Jamaica), October 2005.

[TJF+09] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-Order Model Transformations. In Richard Paige, Alan Hartman, and Arend Rensink, editors, *ECMDA-FA*, volume 5562 of *LNCS*, pages 18–33, Enschede (The Netherlands), June 2009. Springer-Verlag.

[Ull76] Julian R. Ullmann. An Algorithm for Subgraph Isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[URH03] Adelinde M. Uhrmacher, Mathias Röhl, and Jan Himmelspach. Unpaced and Paced Simulation for Testing Agents. In Alexander Verbraeck and Vlatka Hlupic, editors, *15th European Simulation Symposium*, pages 71–80, Delft (The Netherlands), October 2003. SCS.

[VB07] Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):214–234, 2007.

[VFV06] Gergely Varró, Katalin Friedl, and Dániel Varró. Implementing a Graph Transformation Engine in Relational Databases. *Journal on Software and Systems Modeling*, 5(3):313–341, September 2006.

[VG08] Pieter Van Gorp. *Model-Dirven Development of Model Transformations*. PhD thesis, University of Antwerp, 2008.

[VGR08] Pieter Van Gorp and Arend Rensink. 4th International Workshop on Graph-Based Tools: The Contest. `http://fots.ua.ac.be/events/grabats2008/`, September 2008.

[Vis01] Eelco Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In Aart Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *LNCS*, pages 357–362, Utrecht (The Netherlands), May 2001. Springer.

[VJ04] Nico Verlinden and Dirk Janssens. A Framework for NLC and ESM: Local Action Systems. In *6th International Workshop on Theory and Application of Graph Transformations*, volume 1764 of *LNCS*, pages 194–215. Springer-Verlag, February 2004.

[VJBB09] Andrés Vignaga, Frédéric Jouault, María Cecilia Bastarrica, and Hugo Brunelière. Typing in Model Management. In Richard Paige, editor, *ICMT'09*, volume 5563 of *LNCS*, pages 197–212, Zürich (Switzerland), June 2009. Springer-Verlag.

[VM95] Gerhard Viehstaedt and Mark Minas. DiaGen: A Generator for Diagram Editors Based on a Hypergraph Model. In Amihai Motro and Moshe Tennenholtz, editors, *International Workshop on Next Generation Information Technologies and Systems (NGITS'95)*, pages 155–162, Naharia (Israel), June 1995.

[VP03] Dániel Varró and András Pataricza. The Mathematics of Metamodeling is Metamodeling Mathematics. *Journal on Software and Systems Modeling*, 2:187–210, August 2003.

[VSV05] Gergely Varró, Andy Schürr, and Dániel Varró. Benchmarking for Graph Transformation. In *VL/HCC'05*, pages 79–88, Dallas (USA), September 2005. IEEE Press.

[VVF05] Gergely Varró, Dániel Varró, and Katalin Friedl. Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans. In Gabor Karsai and Gabriele Taentzer, editors, *GraMoT'05*, volume 152 of *ENTCS*, pages 191–205, Tallinn (Estonia), September 2005. Elsevier.

[Whi97] Kirsten N. Whitley. Visual Programming Languages and the Empirical Evidence For and Against. *Journal of Visual Languages and Computing*, 8(1):109–142, February 1997.

[WKS+09] Manuel Wimmer, Angelika Kusel, Johannes Schönböck, Thomas Reiter, Werner Retschitzegger, and Wieland Schwinger. Lets's Play the Token Game – Model Transformations Powered By Transformation Nets. In *Workshop on Petri Nets and Software Engineering (PNSE'09)*, pages 35–50, Paris (France), June 2009. Université Paris 13.

[XBZPZ08] Hengheng Xie, Azzedine Boukerche, Ming Zhang, and Bernard P. Zeigler. Design of A QoS-Aware Service Composition and Management System in Peer-to-Peer Network Aided by DEVS. In *DS-RT*, pages 285–291, 2008.

[Zei84] Bernard P. Zeigler. *Multifacetted Modelling and Discrete Event Simulation*. Academic Press, 1984.

[ZLG05] Jing Zhang, Yuehua Lin, and Jeff Gray. Generic and domain-specific model refactoring using a model transformation engine. In *Volume II of Research and Practice in Software Engineering*, pages 199–218. Springer, 2005.

[ZS92] Albert Zündorf and Andy Schürr. Nondeterministic control structures for graph rewriting systems. In Ernst W. Mayr, editor, *Graph-Theoretic Concepts in Computer Science*, volume 570 of *LNCS*, pages 48–62, Fischbachau,Germany, May 1992. Springer-Verlag.

[ZS02] Aleksey V. Zaitsev and Yu A. Skorik. Mathematical Description of Sensorimotor Reaction Time Distribution. *Human Physiology*, 28:494–497 (4), 2002.

[Zün92] Albert Zündorf. Implementation of the imperative / rule based language PROGRES. Aachener Informatik -Berichte 92–38, Department of computer science III, Aachen University of Technology, Germany, 1992.

[Zün94] Albert Zündorf. Graph Pattern Matching in PROGRES. In Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 1073 of *LNCS*, pages 454–468, Williamsburg (USA), November 1994. Springer-Verlag.

[Zün08] Albert Zündorf. The AntWorld Simulation Tool Case. `www.se.eecs.uni-kassel.de/~fujabawiki/index.php/AntWorld`, May 2008.

[ZZH06] Ming Zhang, Bernard P. Zeigler, and Phillip Hammonds. DEVS/RMI-An Auto-Adaptive and Reconfigurable Distributed Simulation Environment for Engineering Studies. *Journal of Test and Evaluation*, 27(1):49–60, April 2006.