A Multi-Paradigm Foundation for Model Transformation Language Engineering[*]

Proposal for Thesis Research in Partial Fulfilment
of the Requirements for the Degree of Doctor of Philosophy

Eugene Syriani
School of Computer Science
McGill University

March 24th, 2010

Abstract

Model transformation is at the heart of current model-driven engineering research. Today's research in model transformation focuses on the scalability of the approaches to solve industrial problems. The current state-of-the-art in model transformation includes a plethora of techniques and tools. Nevertheless, industrial adoption of model transformation requires a diversity of model transformation languages optimally suited for particular transformation tasks. This is why my dissertation contributes to the engineering of model transformation languages, providing a framework to build such languages. The development of this framework is driven by multi-paradigm modelling principles. I illustrate the use of the framework by designing and implementing a novel model transformation language.

[*]Progress Committee: Hans Vangheluwe (supervisor), Jörg Kienzle, and Clark Verbrugge

1 Introduction

Model-Driven Engineering (MDE) [1] is now considered a well-established development methodology. MDE, and in particluar, domain-specific modelling, 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, focusing on some of its properties. Models are used to specify, simulate, test, verify, and generate code for applications. In software language engineering terms, a model conforms to a *meta-model*. 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[1]. MDE allows 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-model. Although a model transformation is defined at the meta-model level, it is nevertheless applied on models. The Object Management Group (OMG) has proposed the the Model-Driven Architecture (MDA), which promotes model transformation at the heart of MDE. The Query, Views, and Transformations (QVT) language [4] 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) [5] addresses these issues and formulates a domain-independent framework. One key aspect of MPM is multi-abstraction. A model abstraction is a view on 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.

1.1 Model Transformation

Given a formalism, the meta-model allows to precisely specify the abstract syntax of the formalism and thus modelling environments can be generated from it. Model transformation can provide *semantics* to models in 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, models in a domain-specific modelling formalism may be transformed into Petri-Nets or Statecharts models. 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 may not hold anymore. Model transformation can be used to *synchronize* models, specifying a bidirectional transformation or relation between different 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.

[1]A common representation of meta-models uses the Unified Modelling Language (UML) Class Diagram notation [2] with Object Constraint Language (OCL) constraints [3].

1.2 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 [6]. They are supported by various implementations such as **AGG** [7], **ATL** [8], **AToM**[3] [9], **GReAT** [10], **MOFLON** [11], **QVT** [4], **VMTS** [12], 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 [13].

In my research, I propose to contribute to the engineering of model transformation languages at the foundation level, following MPM principles. This, 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. I therefore provide a framework for building such model transformation languages and illustrate its applicability by designing and implementing a new model transformation language[2] following the MPM principles for the core algorithms, the transformation language building blocks, and the transformation formalism. The presented approach focuses on the expressiveness of model transformation.

The remainder of this proposal is organized as follows. Section 2 presents an overview of my dissertation. Then, in Section 3, I outline the contributions I have already completed, whereas Section 4 describes the remaining work to be done. In Section 5, I discuss the most relevant related work and finally conclude in Section 6.

2 Overview of the Proposed Solution

In this section, I present an overview of the development of the proposed thesis. The proposed solution first focuses on the foundation of model transformation and then on the elaboration of a novel model transformation language.

2.1 A Basis for Model Transformation

A model transformation language consists of two components: *patterns* and *scheduling*. A transformation is specified in the form of patterns, the fundamental unit of a transformation. For example, in rule-based model transformation (*e.g.*, graph transformation), the transformation unit is a rule. A transformation rule uses patterns as *pre-conditions* and *post-conditions*. The pre-condition pattern determines the applicability of a rule: it is usually described with a left-hand side (LHS) and optional negative application conditions (NACs). The LHS defines the pattern that must be found in the input model to apply the rule. The NAC defines a pattern that shall not be present, inhibiting the application of the rule. The right-hand side (RHS) imposes the post-condition pattern to be found after the rule was applied. An advantage of using rule-based transformation paradigms is that it allows to specify the transformation as a set of operational rewriting rules instead of using imperative programming languages. Model transformation can

[2]More precisely, it is a graph transformation language.

thus be specified at a higher level of abstraction (hiding the implementation of the matching algorithms), closer to the domain of the models it is applied on.

When a model transformation is executed, the rule scheduling describes in what order the rules will be applied (inter-rule management). In declarative transformation approaches, the rule scheduling is implicit and transparent to the modeller who defines a relation between meta-models. On the other hand, operational transformations require an explicit scheduling of the transformation rules. According to MPM principles, both patterns and rule scheduling must be explicitly modelled. Once a meta-model for the pattern language is defined, an environment for specifying individual transformation rules can be generated. The methodology I propose allows to automatically build customized pattern specification languages based on the domain of application (input and output meta-models) of the transformation [14]. It is defined as a systematic *procedure* for explicitly modelling transformation languages. Domain-specific modelling combined with multi-formalism modelling makes us realize that certain formalisms are more appropriate for use in particular applications than others. Applying this philosophy to transformations by systematically and explicitly modelling them rigorously allows to automatically use the same generators for visual modelling environments to generate model transformation environments. This empowers a domain expert to specify the transformation himself, *e.g.*, enhancing the business models with a specific behaviour.

In controlled graph transformation, rule scheduling is often described in a control flow language. Once more, if the scheduling language is meta-modelled, the modeller has control over the order in which the rules can be applied, which increases the expressiveness of the transformation language enormously. Having completely meta-modelled the transformation language, a model transformation can itself be considered as a model, which can in turn be transformed. As a side-effect, this enables to provide a “clean” environment for designing *higher-order transformations*[3] (HOT) [15]. The performance of the automatic synthesis of model transformation environments can be analysed with respect to two criteria: the time and space complexity of the procedure, as well as the usability of a completely modelled environment.

We can examine model transformation languages at a lower level of abstraction: the primitive building blocks that compose these languages. The pre- and post-condition patterns are examples of transformation rule primitives. From an operational point of view, we must also consider the execution of a rule. 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 elements of the input model that satisfy the pre-conditions and of a *transformation phase*, *i.e.*, applying the rule such that the resulting model satisfies the post-conditions. Therefore we can explicitly distinguish between these two phases by a Matcher and a Rewriter as primitives. The process of de-constructing model transformation languages into their primitives enables one to identify further primitive rule operators [16]. For example, one that allows iteration over the matches found by the Matcher or one that detects conflicts between different matches—the latter is useful to prevent the concurrent modification of the same model element at the granularity of the matches as opposed to the granularity of the rule [17]. I have designed the **T-Core** module which encapsulates a minimal collection of graph transformation primitives. It also includes control flow primitives and constructs allowing one to compose the primitive building blocks (such as the Composer).

T-Core becomes useful and applicable when combined with a well-formed programming or modelling languages. It then becomes a complete model transformation language. The level of abstraction **T-Core** is defined at allows one to re-construct existing transformation languages and even design new ones [18]. For example, when combined with a minimalistic language only supporting sequencing, branching, and looping we are able to reproduce the expressiveness of common graph transformation

[3]Higher-order transformations have many applications, such as in language evolution: whenever the definition of a language evolves (meta-model evolution), all associated transformations have to be adapted. This can be partially automated by using a higher-order transformation generated from the modifications made to the language.

languages, such as **FUJABA** [19]. More expressive transformation languages can be built, for example supporting amalgamated rules [20]. Integration with common programming languages, such as Python, opens **T-Core** to the programming world as well.

From the implementation point of view, I focus on the efficient design of the Matcher and Rewriter, integrating various optimization techniques from pattern matching and graph transformation theory. Consequently, **T-Core** provides, on the one hand, a common platform to compare different model transformation languages. On the other hand, consistently applying MPM principles and an MDE approach to the foundation of model transformation languages has lead to the description of a transformation core (**T-Core**). **T-Core** allows for the design of families of languages. Hence, the implementation of **T-Core** offers a ready-to-use framework to build transformation languages.

2.2 A Modular Timed Graph Transformation Language

To demonstrate the power of this multi-paradigm framework for engineering model transformation languages, I combine **T-Core** with the *Discrete Event System Specification formalism* (DEVS). This gives birth to a very expressive novel model transformation language, allowing to model asynchronous transformation models, complemented with the dimension of time. **DEVS** is a discrete-event modelling and simulation formalism that allows for highly modular, hierarchical modelling of timed, reactive systems. Therefore, control structures such as sequence, choice, and iteration are easily modelled in DEVS. Non-determinism and parallel composition also follow from DEVS' semantics. Moreover, its modularity and expressiveness are well-suited to encapsulate graph transformation building blocks [21]. DEVS thus allows modelling the control flow of transformations.

The combination of **T-Core** with **DEVS** results in a novel model transformation language **MoTif-Core**. It consists of embedding every model transformation primitive from **T-Core** onto **DEVS**. For example, The Matcher and the Rewriter are each encoded inside an *atomic DEVS* model. The events that DEVS models can send and receive embed the progress of the transformation, called *packet* (the input graph and additional information on the flow of the transformation). The behaviour is as follows. When a **MoTif-Core** element receives a packet, it processes it according to its **T-Core** semantics. Afterwards, it outputs the packet which can then be received by another **MoTif-Core** element. Note that, because of the use of DEVS, the modeller can specify the time to elapse before an atomic DEVS outputs an event. However, another event can be received in the meantime and interrupt the process. As atomic DEVS models behave as independent processes, the packets exchanged can not be shared among them. This is why the formal definition of **MoTif-Core** specifies that copies of packets are stored in the state of individual atomic DEVS models. The actual implementation relaxes this constraint by storing references to packets for space efficiency trade-off. *Coupled DEVS* models allow to specify a parallel composition of sub-models (atomic or coupled). Sub-models are assumed to be independent processes, concurrent with the rest. Therefore, mapping the Composer primitive from **T-Core** to a coupled DEVS model provides encapsulation constructs with parallel capabilities to **MoTif-Core**. The **MoTif-Core** language thus allows to specify modular, timed, and asynchronous graph transformation models.

MoTif-Core transformations are executed by a DEVS simulator. Since the MPM philosophy is to *model* everything, I explicitly model the DEVS simulator as a model conforming to the **DEVS** formalism. The simulation framework associates an *atomic solver* and a *coordinator* with each atomic and coupled DEVS respectively found in the DEVS model to execute. In the DEVS model of the DEVS simulator, solvers and coordinators are modelled as atomic DEVS models, and the simulation protocol is encoded in their behaviour. Since the protocol is identical to simulate a DEVS model in a distributed environment, **MoTif-Core** transformations can therefore be executed in a distributed fashion.

Although **MoTif-Core** is a model transformation language with enriched expressiveness, it is not ideally usable for a non-DEVS expert. This is why I introduce the model transformation language **MoTif**. It allows to design the transformation rules and the control structure at a higher level of abstraction with **MoTif-Core** components. **MoTif** is designed such that every **MoTif** model has an equivalent **MoTif-Core** model. The meta-model of **MoTif** specifies its syntax: how the modeller can specify transformations. The semantics is defined in terms of the semantics of **MoTif-Core**, which is mapped onto DEVS. In order to have a complete and closed system, the translation of a **MoTif** model to its corresponding **MoTif-Core** model is itself specified as a model transformation. Furthermore, this higher-order transformation[4] is entirely specified as a **MoTif** model.

MoTif is a completely modelled transformation formalism (both its syntax and its semantics). Nevertheless, this does not narrow the expressiveness of the language. On the contrary, it is as expressive as common transformation languages. Furthermore, **MoTif** extends the domain of applicability of model transformations, as it allows to express timed and asynchronous transformations. Another aspect of the performance evaluation of **MoTif** is that it can deal with large host graphs, given that it can be run distributed, without the modeller having to modify the transformation model. Although this dissertation focuses on expressiveness, it is still necessary to analyse the time performance of **MoTif** transformation compared to other very efficient languages (such as **FUJABA** or **GrGen.NET** [22]).

The ultimate goal of my dissertation is to gear model transformation towards industrial-strength usability. The dependability of model transformation systems then 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 from the transformation execution environment at runtime. I therefore introduce exception handling into model transformation languages to increase the dependability of model transformations. Since **MoTif-Core** is an event-driven language, interruption can easily be modelled, making it a good candidate to incorporate *exceptions* [23]. Following MPM principles, the solution treats exceptions as models too. The **MoTif-Core** (and thus **MoTif**) language can be extended to support explicit exception handlers for transformation exception. This improves, on the one hand, the development process of a transformation model by providing debugging abilities to the tools implementing model transformation languages. On the other hand, it also improves the quality of the model transformation software delivered in the context of industrial applications.

Since the **MoTif** framework is entirely modelled following the MPM principles, model checking, model verification, and model-based testing techniques can be applied to analyse the behaviour of transformation models. Safety, liveness, and invariant properties can thus be proven. Note that my research enables analysis of transformations. The actual development of such analyses is however outside the scope of this thesis.

2.3 Applications

A common benchmark comparing the expressiveness of model transformation languages is the *Class Diagram to Relational Database Model System* (CD2RDBMS) [24]. This is a non-trivial transformation converting UML class diagram models into relational database models. One of the challenges of the CD2RDBMS case-study is to compute the transitive closure with respect to inheritance hierarchies in UML and meaningfully represent its semantics in relational databases.

The *AntWorld Simulation* [25] is a case-study considered as a performance benchmark. It consists of letting ants wander around in an area looking for food to bring back to their hill. While ants are in exploration mode, the area expands gradually. Furthermore new ants are generated as food is brought back to the hill. On the one hand, it focuses on the use of locality, *i.e.*, the ability of applying different

[4]This is a HOT as it relates between transformation models. It is specified using the meta-models of **MoTif** and **MoTif-Core**.

rules on the same location of the input model. On the other hand, it focuses on the performance related to the size of the models transformation languages can handle, as many elements are created from the transformation.

Additionally, I propose a case-study that illustrates an application of using DEVS as a semantic domain for timed model transformation. It shows how the explicit notion of time allows for the simulation-based design of reactive systems and, in particular, computer games such as the *Pacman Game* [26]. Modelling the dynamic behaviour of this game in **MoTif** 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 the playability of a game design through simulation.

Finally, I propose a case-study that shows the application of model transformation for aspect-oriented modelling technology. I show how *Aspect Weaving* can be accomplished when using **MoTif** to model that operation. This case-study focuses on the expressiveness of the transformation language as well as on its usability in a *new* domain of application. Since a solution already exists with **Kermeta** [27], I compare the two approaches both from expressiveness and efficiency points of view.

3 Past Contributions

In this section, I summarize the various original contributions I have worked on up to this date. This describes in more detail the work outlined in Section 2.

3.1 Systematic Modelling of Transformations

In collaboration with several participants of the CAMPaM workshop in 2009, I have realized that despite the pivotal significance of transformations for model-driven approaches, there have not been any attempts to explicitly model transformation languages yet. The paper [14], published at the MoDELS MPM workshop in the same year, presents a novel approach for the specification of transformations, 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-model of the rules are (quasi-)automatically generated to create a language tailored to the transformation. This allows, on the one hand, 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 of this paper is to systematically support developers in creating transformation languages by means of semi-automated meta-modelling.

The solution proposes to adapt transformation languages to their specific input and output languages. This is done by explicitly specifying a meta-model for the pre-condition and post-condition patterns of the transformation language based on the input and output meta-models. The required meta-model metamorphosis is obtained by the *RAM process*: Relaxation, Augmentation, and Modification of the involved meta-models. The relaxation step weakens meta-model constraints in the UML class diagram and in the constraint language (*e.g.,* expressed in OCL). this allows partial models to represent patterns. The augmentation step adds transformation-specific features to the meta-model of the patterns. For example, its allows to connect different model elements from different meta-models, *i.e.,* multi-formalism patterns. The modification step increases the expressiveness of the patterns: arbitrary constraints can be specified on pre-condition attributes and arbitrarily complex computations can be used to set values of post-condition attributes.

Figure 1 depicts how a transformation from model M_1 to model M_2 (from formalisms F_1 and F_2 respectively) is defined with this approach. We call T_{1-2} the (transformation) model that defines this

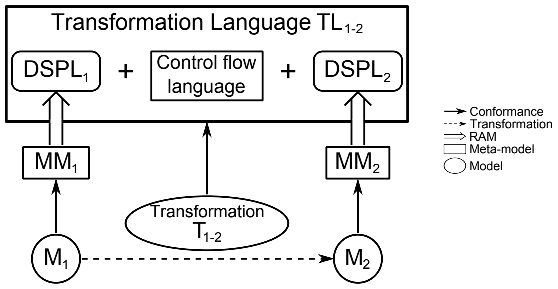

Figure 1: Schema of domain-specific transformation languages

mapping. The two models conform to their respective meta-models MM_1 and MM_2. Applying the RAM technique, *domain-specific pattern languages* are generated from these meta-models, namely $DSPL_1$ and $DSPL_2$ respectively. The meta-models of the patterns (specific to this transformation), combined with the meta-model of the transformation control logic language (*e.g.,* **MoTif**), form the *transformation language* TL_{1-2}. The transformation T_{1-2} is thus a model conforming to its meta-model TL_{1-2}.

One of the main advantages of explicitly modelling the transformation language is to easily define higher-order transformations. A subsequent extension of this work, published in the workshop proceedings of MoDELS 2009 [15], made use of the explicit modelling of transformation languages (both its pattern language and its control-flow language) to easily specify higher-order transformations in a reusable way.

3.2 De-/Re-Constructing Model Transformation Languages

The approach I propose in [28], in collaboration with Hans Vangheluwe, 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 constructs 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. Lastly, once re-constructed, different transformation languages can seamlessly interoperate as they are built on the same primitives.

This is why **T-Core** is introduced as a collection of transformation language primitives for model transformation[5]. This 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, to ensure consistent application of rules executed in iteration or concurrently.

Being sufficiently minimal, **T-Core** primitives are meant to be composed by means of well-formed programming or modelling languages. Figure 2 illustrates examples of such combinations. In [28], **T-Core** was combined with a very simple language that allowed to re-construct full-fleshed existing graph transformation languages as well as more complex ones as mentioned in Section [21].

[5]The complete definition of **T-Core** can be found in [18].

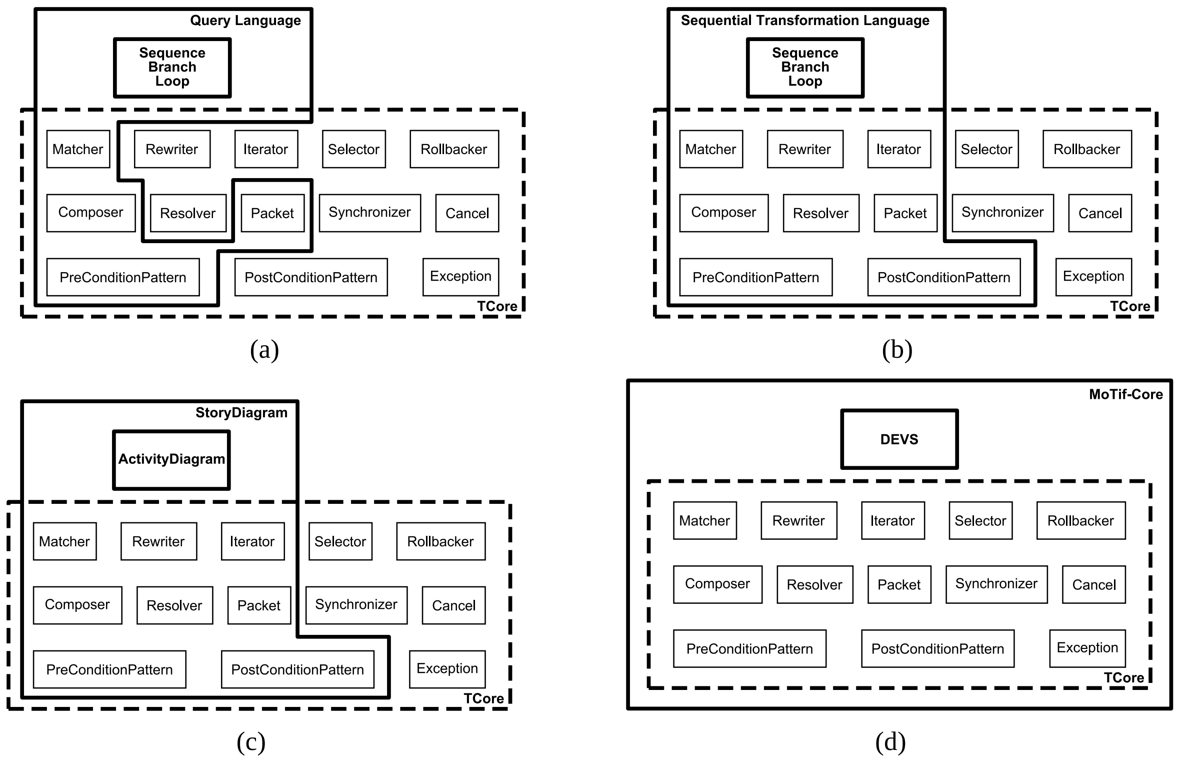

Figure 2: Combining **T-Core** with other languages allows to re-construct existing and new languages

3.3 Programmed Graph Rewriting with DEVS

Jointly with Hans Vangheluwe, I have published a series of articles on how to express controlled graph transformation in terms of the DEVS modelling and simulation formalism. The chronology shows the evolution of the concept preliminary prototypes starting back in 2007 until a formal definition and a complete multi-paradigm integration in the **MoTif** framework.

The first paper, published at AGTIVE 2007 [21], introduces the DEVS formalism to describe and execute graph transformation control structures. In this approach, graphs are embedded in events and individual transformation rules are embedded in atomic DEVS models. Rules can be grouped in encapsulation units (coupled DEVS models). The paper demonstrates that the various graph transformation scheduling paradigms can be modelled in this approach. DEVS can therefore express unordered, layered, priority-based, and programmed graph transformations.

A subsequent publication at ICMT 2007 [26] explores the notion of time in graph transformation, as DEVS is inherently a timed formalism. This allows one to model a time-advance for every rule as well as to interrupt (pre-empt) rule execution. We then demonstrate how the explicit notion of time allows for the simulation-based design of the well-known game of Pacman. 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 paper 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. The user model is subsequently replaced by a visual interface to a real player and the game model is executed using a real-time DEVS simulator. An extended version of this paper has been submitted to the journal of Software and Systems Modelling.

Having emerged interest in the discrete simulation community, Hans Vangheluwe and I published a chapter in a book entitled *Discrete-Event Modelling and Simulation: Theory and Applications* [29]. The chapter formalizes graph transformation control structures by expressing them in terms of DEVS models. This is done by mapping the structure and the behaviour of **MoTif** constructs in terms of atomic and coupled DEVS models. It further elaborates on advantages of using this approach. Moreover, the chapter also shows how this use of DEVS as a semantic domain for controlled rule-based graph transformation allows for simulation and ultimately synthesis of applications. The latter is based on our solution to an extended version of the AntWorld Simulation benchmark found in [25].

3.4 Introduction of Exceptions in Model Transformation

A submitted paper to ICMT 2010 was recently accepted. This joint work with Jörg Kienzle and Hans Vangheluwe introduces the concept of exception handling in the context of model transformation. This allows one to increase the dependability of model transformation languages. This manuscript first analyses and classifies the different kinds of exceptions that can occur in model transformations. Some are more closely related to the execution environment. Those exceptions typically originate from the transformation's virtual machine. For example, such an exception can take the form of a memory overflow, because of infinite recursion or loops in the transformation's control specification, or a null pointer exceptions, because of incorrect expressions specified in the action language of transformation rules. Some exceptions cannot be generalized to all transformation paradigms and thus are more transformation-language specific. A more subtle class of errors are what we have called *rule design exceptions*. These exceptions are due to inconsistent specification of transformation rules. For example, when non-independent rules are applied concurrently or in iteration, the result may lead to non-deterministic, unsafe, or erroneous outputs. The category of transformation-specific exceptions covers domain-specific, application-specific, and user-defined exceptions.

The novelty of this work lies in that we explicitly *model* exceptions and hence *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 the possible exceptions that can occur. We discuss alternative handler designs either in the form of a transformation rule or by an explicit customizable handler. Furthermore, the modeller can also specify if the transformation should resume, restart, or terminate after an exception is handled. In hierarchical transformation languages (such as **MoTif**) where sets of rules can be modularly encapsulated, the propagation of exceptions to a more global context is also modelled.

3.5 Further Work

In [25], I presented my solution to the AntWorld Simulation case-study. The solution used an early prototype of **MoTif** and therefore did not succeed very well in performance efficiency. Nevertheless, because of its expressiveness, **MoTif**/**AToM**^{3} won the price for the best usability solution.

In [28], I propose a review of different model transformation approaches. I first focus on the algebraic foundation of graph transformations and the algorithms used for performing these transformations (more specifically, optimizations on the matching algorithms). Then I present a survey of existing controlled graph transformation languages. They express an operational abstraction for model transformations. Finally, other model-to-model relations (declarative abstraction) are also described.

In [30], I formally define the **MoTif-Core** formalism as a combination of **T-Core** with **DEVS**. Every rule primitive and control flow primitive in **T-Core** is embedded in the an atomic DEVS model. For example, The pre-condition pattern (LHS and NACs) and the matching algorithm of the Matcher are embedded in the external transition of an *atomic DEVS* model. In fact, the pattern is stored in the state of the DEVS model, but evaluated (finding matches) on the packet received when the transition is triggered. The post-condition (RHS) and the rewriting algorithm of the Rewriter are also embedded in an atomic DEVS model in a similar way. Upon reception of a packet, the external transition function triggers the execution of the **T-Core** primitive. The resulting packet is subsequently output by the output function of the DEVS model. Instead of using the **T-Core** Composer, a coupled DEVS model directly encodes the composition of sub-models. The structure and the semantics of **MoTif-Core** are thus defined. Nevertheless, because of the asynchronous nature of this formalism, it is necessary to prove its soundness. That is, whenever a packet is received by a **MoTif-Core** entity, a packet will be output from that entity. This guarantees a proper flow of execution of a **MoTif-Core** transformation and hence ensures its termination from the control structure point of view. This technical report also introduces the meta-model of **MoTif**. Its semantics is defined in terms of **MoTif-Core**, since **MoTif** is a shortcut language. It consists of predefined combinations of **MoTif-Core** entities encapsulated in a Composer.

In [31], we (with the collaboration of Hans Vangheluwe and Amr Al-Mallah) propose to explicitly model the structure and behaviour of a distributed simulator for the DEVS formalism, in terms of a DEVS model. That is because discrete-event simulators, and in particular distributed simulators, 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. The approach first represents a DEVS simulator in a DEVS model. 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. In a modelling and simulation-based approach, we show how the DEVS model of a distributed DEVS simulator is calibrated to behave optimally for the given input model.

4 Anticipated Contributions

What remains to be done is the implementation and the performance analysis of the solution described in Section 2. Although I have already built prototypes for the different publications listed in Section 3, a complete design and efficient implementation of the **MoTif** framework still remains.

1. The RAM process is already implemented. What remains is to evaluate the usability of a completely modelled environment for designing model transformation.
2. I plan to implement **T-Core** as a module based on a model-centric virtual machine. The API must be usable with a modelling language (such as DEVS) or a programming language (such as Python). The advantage of the latter is to make model transformation technologies available to non-model driven paradigms.
3. More precisely, I will elaborate on the implementation of the two crucial operations in graph transformation: *matching* and *rewriting*. As graph pattern matching is known to be NP-complete in the worst case, I will attempt to incorporate existing very efficient techniques in graph pattern matching

that improve the average case performance (*e.g.,* [32, 33]). However, there are also other techniques used by different graph transformation tools. Advanced search plans [34, 22] have been proposed by pruning the search space of the matching with heuristics. Incremental matching [35] sacrifices memory efficiency for time efficiency by loading the whole input graph in memory. Another approach is to map the graph pattern matching problem to a constraint satisfaction problem [36] and then solve the equations/constraints.

4. I will indicate how **QVT Operational** [4] can be implemented using **T-Core**.
5. I will implement the compiler expressing **MoTif-Core** in terms of DEVS. It will be based on the prototype designed from the mapping elaborated in [30].
6. Once **MoTif-Core**'s meta-model and compiler are complete, I will implement the entire **MoTif** framework on top of our model-centric virtual machine[6]. The design of **MoTif** will support handling of transformation exceptions as well as the specification of higher-order transformations.
7. I will then complete the comparison of **MoTif** to other transformation languages.
8. As there are many layers in the **MoTif** framework, I foresee potential bottlenecks in the performance and will address these concerns.
9. I will also assess the feasibility of running **MoTif** transformations in a distributed environment.

Once the framework is designed, implemented, and tested, I will illustrate the application of both a transformation language based on **T-Core** and Python, as well as applications of **MoTif**.

1. I will solve the CD2RDBMS benchmark and compare my results to others. The solution will be designed in **MoTif**.
2. I will refine the solution I proposed for the AntWorld Simulation tool contest. It will however be implemented combining **T-Core** and the object-oriented programming language Python.
3. I will re-implement the Pacman Game case-study using the presented implementation of **MoTif**.
4. Finally, I will provide a solution for the Aspect Weaving case-study to demonstrate the expressiveness of an entirely modelled transformation language **MoTif**.

I plan to complete the first nine points concerning the implementation of the model transformation framework by Fall 2010. I intend to cover the four case-studies by the end of the year 2010. Finally, I expect to write my dissertation during Winter 2011.

5 Related Work

5.1 Existing Graph Transformation Languages

In the sequel, I compare the approaches of some of the relevant scalable graph transformation tools that exist today[7].

5.1.1 ProGReS

The Programmed Graph Rewriting System (**ProGReS**) was the first fully implemented environment to allow programming through graph transformations [37, 38, 39]. The control mechanism is a textual imperative language. A rule in **ProGReS** has a boolean behaviour indicating whether it succeeded or not.

[6]It is a completely redesigned environment for AToM^3 [9] on which I will also work on.

[7]This is by no means an exhaustive list.

Among the imperative control structures it provides, rules can be conjuncted using the `&` operator. This allows 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, isolation, 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 [40]. 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.

5.1.2 AToM^3

AToM^3 is a tool for meta-modelling, multi-formalism modelling, and model transformation [9]. Model transformation can be performed on models conforming to a product of meta-models[8]. 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 overlapping rules conflict detection. It is also possible to divide transformations in layers by sequencing graph grammars – without priorities.

5.1.3 GReAT

GReAT (for Graph Rewriting And Transformation language) is the model transformation language for the domain-specific modelling tool GME [10]. **GReAT**'s control structure language uses a proprietary asynchronous dataflow diagram notation where a production is represented by a "block" (called *Expression* in [10]). 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.

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

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 following expressions 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).

5.1.4 VMTS

The controlled graph rewriting system of **VMTS** is provided by the VMTS Control Flow Language (VCFL) [41], 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 [41].

5.2 Other Model Transformation Approaches

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

5.2.1 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 elements) it is preferable to store them in a database. For that reason, Varró et al. propose in [42] 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 in 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 to be 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 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 indexes on columns where a matching occurs.

5.2.2 ATL

The ATLAS Transformation Language (**ATL**) is a hybrid model transformation language combining declarative and imperative constructs [8, 43]. 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 reuse the target elements they created when applied multiple times. Declarative rules support inheritance as means of reuse 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 of a formal foundation, unlike graph-based transformation.

6 Conclusion

In this proposal, I outlined a novel approach for the engineering of model transformation languages, driven by multi-paradigm modelling principles. The core of this approach consists of three different model transformation formalisms. At the foundation level, the meta-modelled language **T-Core** presents a collection of most primitive model transformation language constructs. This offers a common basis for specifying model transformation formalisms to allow uniform comparison and interoperability possibilities between them. **MoTif-Core** combines **T-Core** with DEVS. This permits to express new dimensions to transformation models with asynchrony and time. Both the language and its execution framework (DEVS simulator) are modelled. Since **MoTif-Core** is not an ideal formalism from a usability point of view, the elaboration of a model transformation language suited for its domains of application is required. I argue that **MoTif** fills this abstraction gap.

Although the focus of my dissertation is on improving the expressiveness of the model transformation paradigm, I will conduct performance analyses to compare the **MoTif** framework to other model transformation engineering approaches.

References

[1] Stahl, T., Voelter, M., and Czarnecki, K. (2006) *Model-Driven Software Development – Technology, Engineering, Management*. John Wiley & Sons.

[2] Object Management Group (2009) *Unified Modeling Language Superstructure*.

[3] Object Management Group (2006) *Object Constraint Language*.

[4] Object Management Group (2008) *Meta Object Facility 2.0 Query/View/Transformation Specification*.

[5] Mosterman, P. J. and Vangheluwe, H. (2004) Computer automated multi-paradigm modeling: An introduction. *Simulation: Transactions of The Society for Modeling and Simulation International*, **80**, 433–450.

[6] Czarnecki, K. and Helsen, S. (2006) Feature-based survey of model transformation approaches. *IBM Systems Journal, special issue on Model-Driven Software Development*, **45**, 621–645.

[7] Taentzer, G. (2004) AGG: A graph transformation environment for modeling and validation of software. *AGTIVE'03)*, LNCS, **3062**, pp. 446–453. Springer-Verlag.

[8] Jouault, F. and Kurtev, I. (2006) Transforming models with ATL. *MTiP'05*, January, LNCS, **3844**, pp. 128–138. Springer-Verlag.

[9] de Lara, J. and Vangheluwe, H. (2002) AToM^3: A tool for multi-formalism and meta-modelling. In Kutsche, R.-D. and Weber, H. (eds.), *FASE'02*, Grenoble (France), April, LNCS, **2306**, pp. 174–188. Springer-Verlag.

[10] Agrawal, A., Karsai, G., Kalmar, Z., Neema, S., Shi, F., and Vizhanyo, A. (2006) The design of a language for model transformations. *SoSym*, **5**, 261–288.

[11] Amelunxen, C., Königs, A., Rötschke, T., and Schürr, A. (2006) MOFLON: A standard-compliant metamodeling framework with graph transformations. In Rensink, A. and Warmer, J. (eds.), *Model Driven Architecture - Foundations and Applications: Second European Conference*, LNCS, **4066**, pp. 361–375. Springer-Verlag.

[12] Lengyel, L., Levendovszky, T., Mezei, G., and Charaf, H. (2005) Control flow support in metamodel-based model transformation frameworks. *EUROCON'05*, Belgrade (Serbia), November, pp. 595–598. IEEE.

[13] Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., and Lindow, A. (2006) Model transformations? transformation models! *MoDELS'06*, Genova (Italy), LNCS, **4199**, pp. 440–453. Springer Verlag.

[14] Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., and Wimmer, M. (2009) Systematic transformation development. *ECEASST*, **21**.

[15] Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., and Wimmer, M. (2010) Explicit transformation modeling. In Ghosh, S. (ed.), *MODELS 2009 Workshops*, LNCS, **6002**, pp. 240–255. Springer.

[16] Syriani, E. and Vangheluwe, H. (2010) De-/re-constructing model transformation languages. *ECEASST*, **(in press)**.

[17] Heckel, R., Küster, J. M., and Taentzer, G. (2002) Confluence of typed attributed graph transformation systems. In Corradini, A., Ehrig, H., Kreowski, H.-J., and Rozenberg, G. (eds.), *ICGT 2002*, Barcelona (Spain), October, LNCS, **2505**, pp. 161–176. Springer-Verlag.

[18] Syriani, E. and Vangheluwe, H. (2009) De-/re-constructing model transformation languages. Technical Report SOCS-TR-2009.8. McGill University, School of Computer Science.

[19] Fischer, T., Niere, J., Turunski, L., and Zündorf, A. (2000) Story diagrams: A new graph rewrite language based on the Unified Modelling Language and Java. In Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G. (eds.), *Theory and Application of Graph Transformations*, Paderborn (Germany), November, LNCS, **1764**, pp. 296–309. Springer-Verlag.

[20] Rensink, A. and Kuperus, J.-H. (2009) Repotting the geraniums: On nested graph transformation rules. In Margaria, T., Padberg, J., and Taentzer, G. (eds.), *GT-VMT'09*, York (UK), March, ECE-ASST, **18**.

[21] Syriani, E. and Vangheluwe, H. (2007) Programmed graph rewriting with DEVS. In Nagl, M. and Schürr, A. (eds.), *AGTIVE'07*, Kassel (Germany), LNCS, **5088**, pp. 136–152. Springer-Verlag.

[22] Geiß, R., Batz, G. V., Grund, D., Hack, S., and Szalkowski, A. (2006) GrGen: A fast SPO-based graph rewriting tool. In Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., and Rozenberg, G. (eds.), *ICGT'06*, Heidelberg (Germany), September, LNCS, **4178**, pp. 383–397. Springer-Verlag.

[23] Syriani, E., Kienzle, J., and Vangheluwe, H. (2010) Exceptional transformations. Technical Report SOCS-TR-2010.2. McGill University, School of Computer Science.

[24] Taentzer, G., Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Varró, D., and Varró-Gyapay, S. (2005) Model transformation by graph transformation: A comparative study. *MTiP'05*, Montego Bay (Jamaica), October.

[25] Syriani, E. and Vangheluwe, H. (2008) Using MoTif for the AntWorld simulator case study. In Van Gorp, P. and Rensink, A. (eds.), *GraBaTs'08*.

[26] Syriani, E. and Vangheluwe, H. (2008) Programmed graph rewriting with time for simulation-based design. In Pierantonio, A., Vallecillo, A., Bézivin, J., and Gray, J. (eds.), *ICMT'08*, Zürich (Switzerland), July, LNCS, **5063**, pp. 91–106. Springer-Verlag.

[27] Kienzle, J., Al Abed, W., and Klein, J. (2009) Aspect-oriented multi-view modeling. *AOSD'09*, Charlottesville (USA), March, pp. 87–98. ACM Press.

[28] Syriani, E. and Vangheluwe, H. (2009) Matters of model transformation. Technical Report SOCS-TR-2009.2. McGill University, School of Computer Science.

[29] Syriani, E. and Vangheluwe, H. (2009) Discrete-Event Modeling and Simulation: Theory and Applications. CRC Press, Boca Raton (USA).

[30] Syriani, E., Vangheluwe, H., and Al-Mallah, A. (2010) A modular timed model transformation language. Technical Report SOCS-TR-2010.4. McGill University, School of Computer Science.

[31] Syriani, E., Vangheluwe, H., and Al-Mallah, A. (2010) Modelling and simulation-based design of a distributed devs simulator. Technical Report SOCS-TR-2010.3. McGill University, School of Computer Science.

[32] Ullmann, J. R. (1976) An algorithm for subgraph isomorphism. *Journal of the ACM*, **23**, 31–42.

[33] Cordella, L., Foggia, P., Sansone, C., and Vento, M. (2004) A (sub)graph isomorphism algorithm for matching large graphs. *TPAMI*, **26**, 1367–1372.

[34] Valiente, G. and Martínez, C. (1997) An algorithm for graph pattern-matching. In Baeza-Yates, R. (ed.), *4th South American Workshop on String Processing*, Valparaiso (Chile), November, International Informatics Series, **8**, pp. 180–197. Carleton University Press.

[35] Bunke, H., Glauser, T., and Tran, T.-H. (1991) An efficient implementation of graph grammars based on the RETE matching algorithm. *Graph Grammars and Their Application to Computer Science*, Bremen (Germany), March, LNCS, **532**. Springer.

[36] Larrosa, J. and Valiente, G. (2002) Constraint satisfaction algorithms for graph pattern matching. *Mathematical Structures in Computer Science*, **12**, 403–422.

[37] Blostein, D. and Schürr, A. (1999) Computing with graphs and graph rewriting. *SPE*, **9**, 1–21.

[38] Zündorf, A. (1994) Graph pattern matching in PROGRES. In Ehrig, H., Engels, G., and Rozenberg, G. (eds.), *Graph Grammars and Their Application to Computer Science*, Williamsburg,USA, November, LNCS, **1073**, pp. 454–468. Springer-Verlag.

[39] Schürr, A., Winter, A. J., and Zündorf, A. (1995) Graph grammar engineering with PROGRES. In Schäfer, W. and Botella, P. (eds.), *5th European Software Engineering Conference*, Sitges,Spain, September, LNCS, **989**, pp. 219–234. Springer-Verlag.

[40] Zündorf, A. (1992) Implementation of the imperative / rule based language PROGRES. Aachener Informatik -Berichte 92–38. Department of computer science III, Aachen University of Technology, Germany.

[41] Lengyel, L., Levendovszky, T., Mezei, G., and Charaf, H. (2006) Model transformation with a visual control flow language. *IJCS*, **1**, 45–53.

[42] Varró, G., Friedl, K., and Varró, D. (2006) Implementing a graph transformation engine in relational databases. *SoSym*, **5**, 313–341.

[43] Jouault, F. (2006) Contribution à l'étude des langages de transformation de modèles. Ph.D. thesis Université de Nantes.