VERIFICATION OF MODEL TRANSFORMATIONS

By

Anantha Narayanan

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

May 2008

Nashville, Tennessee

Approved:

Professor Gabor Karsai

Professor Janos Sztipanovits

Professor Aniruddha Gokhale

Professor Sherif Abdelwahed

Professor Bharat Bhuva

To the memory of my mother,
Shrimati S. Mangalam

ACKNOWLEDGMENTS

This research was supported by a grant from NSF/CSR-EHS, titled Software Composition for Embedded Systems using Graph Transformations, award number CNS-0509098.

First and foremost, I would like to heartily thank my advisor Dr. Gabor Karsai. He has been an inspiration to me, and his motivation and guidance have been invaluable throughout the course of this research. I am also grateful to the other members of my committee, Dr. Janos Sztipanovits, Dr. Aniruddha Gokhale, Dr. Sherif Abdelwahed and Dr. Bharat Bhuva, for their pertinent questions that have directed my towards my goal.

I would also like to acknowledge the support of my fellow students and staff at ISIS. I have I owe special thanks to the GReAT development team, especially Aditya Agrawal, Attila Vizhanyo, Feng Shi and Daniel Balasubramanian, for always helping me out when I had questions. Thanks also to Tivadar Szemethy for the many interesting discussions.

Last but not the least, I would like to thank my father Suryanarayanan and my sister Anuradhai Mukundan for their encouragement and unfaltering belief in my abilities. I would like to thank my wife Suchismita, for always having a word of encouragement when I faced a difficulty. I am indebted to her for her undying affection and sincere support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The increasing complexity of software and hardware systems has driven the need for higher levels of abstraction in their design and development. This has led to the adoption of the traditional engineering practice of *modeling* into software engineering. Models allow designers to view a system's structure or behavior at a higher level of abstraction, thus simplifying the design and analysis of such systems. Model-Based Software Engineering (MBSE) is a software development methodology that places emphasis on the formal understanding of the features and structure of a product family, by creating and using reusable models. This approach has become increasingly popular in recent times, with advances such as OMG's Model Driven Architecture (MDA) initiative [1] and supporting tools.

The MBSE approach is particularly suited to key areas such as embedded systems, where there is a close correspondence between the modeling abstractions and the physical entities of the system. More interestingly, a set of selected modeling abstractions can be used to describe an entire class of systems (called a *domain*). Such a set of abstractions is termed a Domain Specific Modeling Language (DSML), and models of such systems are called Domain Specific Models [2]. Powerful design environments such as GME [3] have helped design DSMLs more effectively.

Domain specific languages are often highly specialized, showing one aspect of a system and hiding several underlying details. While this offers convenient abstractions to visualize and analyze systems, it necessitates the use of several different types of abstractions over the course of a model based development effort. On one hand, the models must be converted to binary code to be implemented on a certain platform.

On the other hand, a system description in one DSML may need to be converted into another DSML for specific activities. For instance, different DSMLs may be suitable for design and for analysis, in which case, the design model must be converted into a model in the analysis DSML. These issues, called *model interpretation* and *model transformation*, form an essential part of model based software engineering.

Current design methods in MBSE focus on syntax and static semantics of DSMLs. However, the interpretation of the model is usually done through hand-written code, which is often cumbersome and error prone. Recent developments in using graph transformations [25] [41] [43] have enabled domain experts to specify model transformations visually, in terms of the domain specific model elements. Transformation of domain specific models from one domain to another, for purposes such as verification and analysis, or porting to a different platform, has become an integral part of the model based development process. Graph transformations are used extensively in these cases. There has also been some work on specifying the dynamic semantics of domain specific languages visually [48]. This would allow domain experts to specify the behavior of the domain specific language in terms of its graphical elements, and not rely on algorithms written in other programming languages.

Though this has simplified the specification process, the transformations are still prone to error. The correctness of these transformations is crucial to the success of the model based software development process. Model based software development projects are typically built over tool-chains that tie modeling, verification and code generation together. Different tools are used to perform the specific tasks of design, analysis and implementation. Models must usually be transformed to the appropriate notation at each step. For instance, a system may be designed using Statecharts, transformed into an FSM representation for verification, and implemented as C code,

with model transformations producing the required models at each stage. The verification may provide valuable results about the properties of the model - but whether these results may be applied to the design model and the generated code hinges on the correctness of the model transformations.

Verification and analysis tools today are extremely powerful, capable of handling very large models and producing very valuable results. However, their results are meaningless if errors in the model transformations produced incorrect representations. Model-based development of high-consequence, critical systems must address the need for verifiable model transformations and code generators that give assurances for the logical consistency of the tool chain.

Traditionally, the correctness of such transformations is provided by reasoning about its design. But this reasoning often does not address all possible practical scenarios. Even if the design were correct, this reasoning will not capture errors in the implementation or the execution of the transformation. There is a need for some guarantee of the correctness of the transformation, or in other words, a *certifiability* of its correctness.

It is clear that the correctness of model transformations is crucial to the success of a model based development approach. This dissertation addresses the problem of verifying model transformations, based on the following hypothesis.

Verifying the correctness of model transformations is, in general, as difficult as verifying compilers for high-level languages. However, the limitations enforced by the syntactic and semantic rules of domain specific languages allow us to reason about the properties of models within a restricted framework. In model transformations between domain specific models, the preservation of certain properties of the models may also be reasoned within the same restricted framework.

For practical purposes, a transformation may be said to have 'executed correctly' if a certain instance of its execution produced a model that preserved a certain property of interest. We call that instance 'certified correct'. If the property can be specified correctly, the model transformation can be extended to verify if it was preserved in the output model. The model transformation process can thus be modified to provide a *certificate of correctness*, along with the output model instance.

This dissertation is organized as follows. Chapter II surveys the state of the art in model based software engineering, from low level models of computation to higher level domain specific languages. The definitions of some formalisms are investigated, which will later be used in the case studies on verification of model transformations.

Chapter III gives an overview of model transformations, beginning with OMG's QVT recommendation for model transformations. The different types of model transformations and the related terminology is discussed next. Special attention is given to graph transformations concepts, with a survey of transformation tools that are based on graph transformations. A detailed overview of the GReAT transformation toolkit is provided. The case studies presented in this dissertation were prepared using GReAT.

Chapter IV provides some background on verification techniques in general. Some compiler verification techniques are discussed, including the 'certification' based verification approach. Comparing program equivalence using bisimulation is introduced. These techniques will be extended and applied to model transformations, to illustrate the verification of behavior preservation across transformations.

Chapters V to VIII are previously published papers that document the progression of this dissertation. Chapter V presents the following paper: Anantha Narayanan and Gabor Karsai, "Towards Verifying Model Transformations", Proceedings of the 5th International Workshop on Graph Transformations and Visual Modeling Techniques,

in Electronic Notes in Theoretical Computer Science (ENTCS), pp. 185-194, 2006. This paper introduces the idea of "instance-based" verification, where we construct a framework around a model transformation such that each execution of the transformation is verified, for certain selected properties of interest. If the verification succeeds for a certain execution, that output instance model is said to be certified correct. Bisimulation is used to verify whether the model transformation preserved reachability related properties over the instance models of two different domains.

Chapter VI presents the following paper: Anantha Narayanan and Gabor Karsai, "Using Semantic Anchoring to Verify Behavior Preservation in Graph Transformations", Electronic Communications of the European Association of Software Science and Technology (ECEASST), Volume 4: Graph and Model Transformation, 2006. This paper extends the idea of instance-based verification to more complex transformation cases. Semantic anchoring is used as the underlying medium for the verification framework. Weak-bisimulation is used to verify the preservation of reachability related properties. It is more lenient than strict bisimulation, allowing for some differences between the source and target domains.

Chapter VII presents the following paper: Anantha Narayanan and Gabor Karsai, "Verifying Model Transformations by Structural Correspondence", Seventh International Workshop on Graph Transformation and Visual Modeling Techniques (accepted for publication, final revised paper is expected to appear in the Electronic Communications of the EASST), Budapest, Hungary, March 2008. This paper introduces the idea of "structural correspondence", which addresses a class of transformations where correctness can be specified by requirements on the structure of the output model (as opposed to a behavior specification). A simple UML Activity Diagrams to a CSP (Communicating Sequential Processes) specification is used as a case study to

explain how correctness can be specified as structural rules that must be satisfied by the output model instance.

Chapter VIII presents the following paper: Anantha Narayanan and Gabor Karsai, "Specifying the Correctness Properties of Model Transformations", 3rd International Workshop on Graph and Model Transformation (GraMoT) (accepted for publication, final revised paper is expected to appear in ACM and IEEE digital libraries), Leipzig, Germany, May 12, 2008. In this paper, the idea of verification by structural correspondence is extended, with special attention to a query language for specifying the structural correspondence. The requirements of a framework to support such a verification is discussed.

Chapter IX covers concluding remarks on the techniques described in the previous chapters, and discusses avenues for future research.

CHAPTER II

BACKGROUND: MODELS IN SOFTWARE AND SYSTEMS ENGINEERING

As computers and the systems they operate in have become more complex, programming them has become increasingly difficult. Abstraction has played an important role in simplifying the task of programming. Assembly language abstracted the actual bits into mnemonics that were easier to remember. Low level languages abstract much of the system hardware details, and higher level languages provided abstractions to represent real-world data.

Models provide a higher level of abstraction, which helps users to reason about systems without getting into the low level system details. Modeling languages can be classified based on the level of detail they capture, into 'low-level' (Finite State Machines, Timed Automata) and 'high-level' (UML) languages. In addition, a modeling language is said to be 'Domain Specific' if it caters to the needs of a specific domain, using modeling primitives to represent computations or system entities prevalent in that domain. Models of computation provide abstractions for reasoning about algorithms and system behavior. Here, the term 'model' refers to a mathematical model that represents the algorithm or system behavior. Higher level modeling languages allow us to design and analyze system architecture at a higher level of abstraction. In this case, 'models' represent physical entities such as hardware components. Some modeling languages allow an integration of the two levels, such as Matlab's Simulink [4]. Simulink models can describe both the physical system and the inherent control algorithms in a single model. In the following sections, we will review some modeling languages at different levels of detail.

We will review some such modeling languages in the following sections, starting with low-level models of computation, and going on to higher level modeling languages. We will also see the significance of domain specific models in Model-Based Software Engineering.

Models of Computation

A model of computation can be described as *a formal, abstract definition of a computer* [5]. Models of computation allow users to reason about algorithms or system behavior by looking at an abstraction of the system, while ignoring the lower level implementation details. A model of computation is suited to model the properties of a certain class of systems, and is usually rooted in formal mathematical foundations that lend themselves to formal analysis.

Finite State Machines

A Finite State Machine (or a Finite State Automaton) [6] is a model of some behavior in terms of a finite number of states, and a finite number of transitions between these states. In addition, a number events can occur, either as input events or as a result of transitioning to a state. A transition function maps the current state to the next state, depending on the input events. An FSM can be formally described as a tuple $\mathbf{(S, s_0, T, I, O)}$, where:

$\mathbf{S}$ is a set of states,

$\mathbf{s_0} \in \mathbf{S}$ is the initial state of the system,

$\mathbf{T}$ is a set of transitions,

$\mathbf{I}$ is a set of input events,

and **O** is a set of output events,

and the transition **T** can be described as a tuple $(\boldsymbol{s_i}, \boldsymbol{s_f}, \boldsymbol{i}, \boldsymbol{o})$, where:

$\boldsymbol{s_i}$ is the starting state of the transition,

$\boldsymbol{s_f}$ is the ending state of the transition,

$\boldsymbol{i}$ is a set of events necessary to activate the transition,

$\boldsymbol{o}$ is a set of events output as a result of taking the transition.

When the output events are associated with transitions, the FSM is classified as a *Mealy Machine*. This means that the output of the FSM depends on both its current state and the input to the FSM. If the output events are associated with states, then the FSM is classified as a *Moore Machine*. In this case, the output of the FSM depends only on its current state. If there exists at most one transition for each combination of input event and initial state, the FSM is said to be *deterministic*. Otherwise, it is said to be *non-deterministic*.

Finite State Machines are very simple to use, and yet powerful enough for a wide range of applications. They are popularly used to model sequential reactive control systems. They are also used in formal analysis tools such as SMV [7] and SPIN [8]. However, the basic FSM model does not allow hierarchy (for simplicity and clarity, by enclosing sub-states in a parent state), concurrency (two or more states being active simultaneously) or communication (to synchronize two operations). This makes them unsuitable for modeling systems with a large number of states, or concurrent processes and broadcast communication. Several variants of the basic FSM have been developed to address specific areas [9]. Some of these variants will be discussed next.

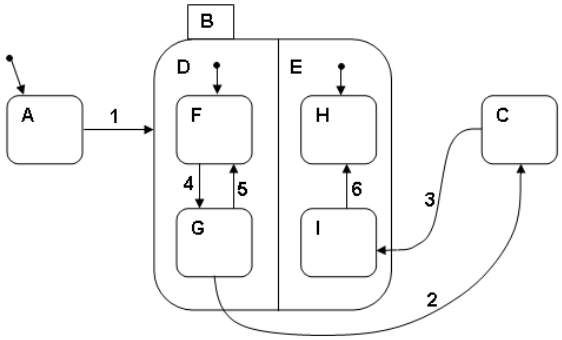
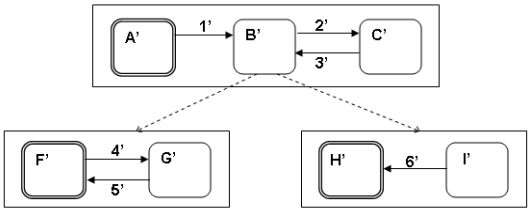


Figure 1: A sample Statechart model

Statecharts

Statecharts were first proposed by Harel [10] as an extension of FSMs to model the reactive behavior of systems. The behavior of such systems is termed *reactive* or *event-driven*, as it is a result of the immediate state of the system and its input events. Statecharts allow hierarchy, concurrency and broadcast communication. States are hierarchical, meaning they can contain other states and transitions. States are classified as *simple* (basic states with no substates), *composite* (also called OR states) or *concurrent* (also called AND states). If a system is in a composite state , it is also in exactly one of its direct sub-states. If a system is in a concurrent state, it is also in all of its direct sub-states. Events are the basic unit of broadcast communication, which allows different orthogonal processes to react to the same event. Figure 1 shows a sample Statechart model.

One of the significant advances to come with Statecharts was the Statemate environment for the design and analysis of complex reactive systems [11]. The Statemate tool offered a graphical environment for specifying Statechart models visually, with tools for simulation and analysis. The combination of a well defined semantics [12] with the availability of supporting tools has made Statecharts a very popular modeling formalism. Yet, concerns with specific syntactical and semantical issues have led



Transition Label	SR	TD
2'	G'	
3'		I', F'

Figure 2: Sample EHA model

to the development of a number of variants of the original Statecharts formalism. A number of these variants are surveyed in [13].

Extended Hierarchical Automata

Extended Hierarchical Automata (EHA) were introduced as an alternate representation to provide formal operational semantics for Statecharts [14]. EHA offers an alternative simplified hierarchical representation for Statecharts. It is a simpler formalism with a restricted syntax. It only allows transitions between a single source and a single target, that cannot cut across levels of hierarchy, as in usual automata. These factors make EHAs useful in correctness proofs [15]. EHA models are composed of one or more *Sequential Automata*, which are non-hierarchical finite automata. The states of a Sequential Automaton (called *Basic States*) may be *refined* into further Sequential Automata, to express hierarchy in a flat notation. Transitions in an EHA model are always contained within one Sequential Automaton, and cannot cut across levels of hierarchy. Transitions in the EHA are also annotated with special attributes called *source restriction* and *target determinator*, which keep track of the actual source and target states, thus allowing the specification of transitions between states at different levels of hierarchy.

By appropriately constructing the EHA and appropriately annotating the transitions, Statechart models can be represented in EHA form. Figure 2 shows a sample

EHA model, which represents the Statechart in Figure 1. The main steps in building an EHA representation of a Statechart model are outlined in [15], where EHA are used as an intermediate format for verifying Statecharts using the SPIN [16] model checker.

Model Based Software Engineering

The models of computation discussed so far can be termed as "low level models". They model the algorithmic details of specific functionalities or behavioral properties of systems. Higher level modeling languages abstract away more of the internal details and functioning of the system. Some high-level modeling languages allow us to view the systems architecture in terms of its components, without going into the details of the functioning of the individual components. In Model Based Software Engineering, high-level models are used to capture the features and structure of a product family. This improves the ability to construct and analyze systems using flexible, reusable components.

Model Driven Architecture

Model Driven Architecture (MDA) [1] is an Object Management Group (OMG) initiative for a vendor-neutral approach to developing applications and writing specifications. It is based on specifying a Platform-Independent Model (PIM) that captures the required functionality or behavior, and one or more Platform-Specific Models (PSM) for each platform that the application must support.

The primary goals of MDA are portability, interoperability and reusability [17]. This is achieved by separating the specification of the system functionality and the specification of its implementation on a certain platform. A Platform Model represents the parts and services provided by a certain platform, along with the restrictions

on the use of those services. A Platform-Specific Model compatible with a certain platform will be able to use only the services provided by its Platform Model. A Platform-Independent Model is usually *transformed* into a Platform-Specific Model to implement the system functionality on the selected platform. The flexibility and automation of this transformation process will be discussed in the next chapter.

The Unified Modeling Language

The Unified Modeling Language (UML) [18] is an OMG standard, for visual modeling of object-oriented system designs. UML is composed of a set of sub-languages, the most popular among which is the language of Class Diagrams. Class diagrams model the hierarchical structure of the systems using classes, attributes, operations and associations. The functional and dynamic aspects of the system can be modeled using the other sub-languages such as use case diagrams, sequence diagrams, activity diagrams and state machines. Together, these allow the user to visually specify and document a software system.

Class diagrams are used to visually model classes and their relationships. Classes are represented by a rectangle with the class name, with its attributes and operations listed below. Relations between classes are indicated by using connector lines between the classes. Classes can have different types of relations, and a specialized type of connector is used for each type of relation. There are three basic kinds of relations between UML classes, as listed below.

Inheritance. A class can *inherit* the methods and attributes of another class, in addition to its own attributes and methods. This is indicated by a connector with an empty triangle. In Figure 3, *ClassB* inherits from *ClassA*.

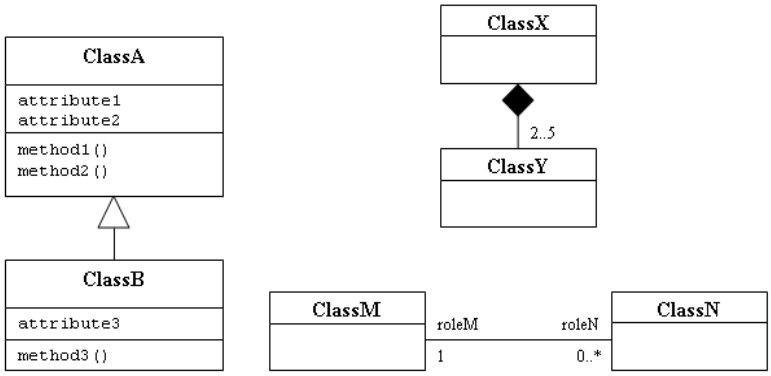


Figure 3: UML Class Diagrams

Containment. Containment indicates a 'whole/part' relationship between instances of the classes. There are two types of containment. *Aggregation* represents a containment association in which the part can exist without the parent. It is represented by a connector with an empty diamond. *Composition* is a stronger form of containment, where the lifetime of the 'part' instance is controlled by the 'whole' instance. This is indicated by a connector with a solid diamond. A multiplicity at the end of the contained class specifies the number of instances of the class that can be contained. In Figure 3, *ClassX* can contain 2 to 5 instances of *ClassY*.

Association. Other relations between classes are called associations, represented by a solid line between the classes. The *role* played by each class is specified on the association, along with the cardinality. In Figure 3, each instance of *ClassM* can be associated with any number of *ClassN* instances.

UML class diagrams are a powerful and intuitive way to visually specify the hierarchical structure of an object-oriented system. UML uses State Machines for modeling behavior, in addition to sequence diagrams and activity diagrams. UML State Machines are a variation of the original Harel Statecharts. UML's ease of use and simplicity have led to its popular use in numerous large software projects. There

are several tools available for designing and viewing UML models and generate code form them, such as Rational Rose [19], Visio [20] etc. However, UML is not completely formal. There have been several attempts to formalize UML, especially the parts involving behavior modeling. In [21], an attempt is made to formalize UML State Machines by converting them into a term rewriting system, and defining an operational semantics., to be verified using a model checker. In [22], a graph transformation based approach is used to provide integrated formal semantics for UML diagrams. Handling the diversity and size of UML models have been the main issues in most of these approaches.

Model Integrated Computing

In applications such as embedded systems, there is a very tight coupling between software and its associated physical system. In such cases, the high level models become specialized to represent a specific domain. Such classes of models are called Domain Specific Modeling Languages. Model Integrated Computing (MIC) is based on the intention to "extend the scope and use of models so that they form the backbone of system development" [23]. Models are used not only for design, but also for analysis, testing and operation, with models capturing information both about the system and its environment [2].

In MIC, domain specific modeling elements are used to represent the real world entities of a system, which include its hardware and software components, and its environment. These elements and their relationships constitute a Domain Specific Modeling Language (DSML). As the system evolves with time, the DSML also evolves, maintaining the coupling between the software and the physical system. The key aspects of MIC are multiple-view models capturing all relevant information about

the system, and tool-specific model interpreters to analyze the models and translate them into executable code [24].

Figure 4 shows the typical MIC development cycle, which begins with the specification of a new application domain. The application domain may itself be specified by modeling, using a specialized modeling language. This is called *Metamodeling*, and the model that specifies the new domain is called a *Meta-model*. A meta-model defines a new Domain Specific Modeling Language (DSML) by specifying the concepts, relationships and integrity constraints in the language, along with the visualization rules which determine how the language will be visualized in a graphical modeling environment. The step called "Meta-level Translation" takes this specification and generates a Domain Specific Design Environment (DSDE), which forms a part of the Model-Integrated Program Synthesis (MIPS) environment. The DSDE is used to create domain-specific models that conform to the rules specified in the metamodel of that domain. "Model Interpretation" is a step where the information in the domain specific model is analyzed by a model interpreter, which may generate executable code or other artifacts relevant to the application domain. Model interpretation may be achieved programmatically (using C++ or Java), or by Graph Transformation. As the application domain evolves, it may demand change in the models, which in turn may demand changes in the meta-model. While it is normal for models to change frequently, the meta-model changes only in rare special cases.

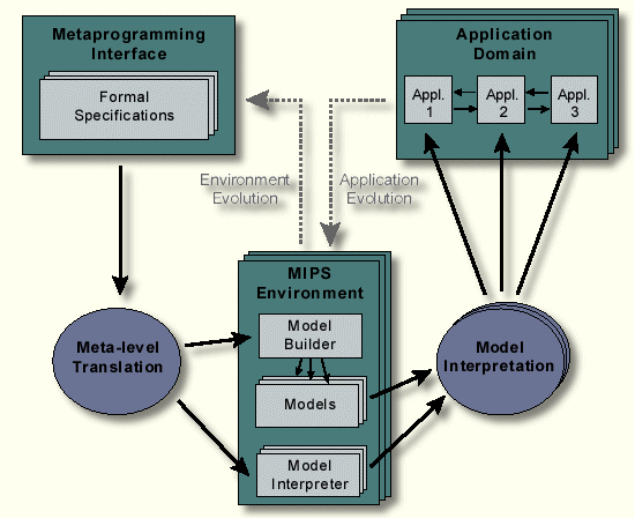


Figure 4: The MIC Development Cycle [24]

GME

The Generic Modeling Environment (GME) [3] is an integrated environment for Model Integrated Computing, developed at the Institute for Software Integrated Services (ISIS), Vanderbilt University. GME includes a powerful set of tools for specifying and visualizing Domain Specific Modeling Languages (DSMLs) using a visual interface, and a powerful API for traversing the domain models.

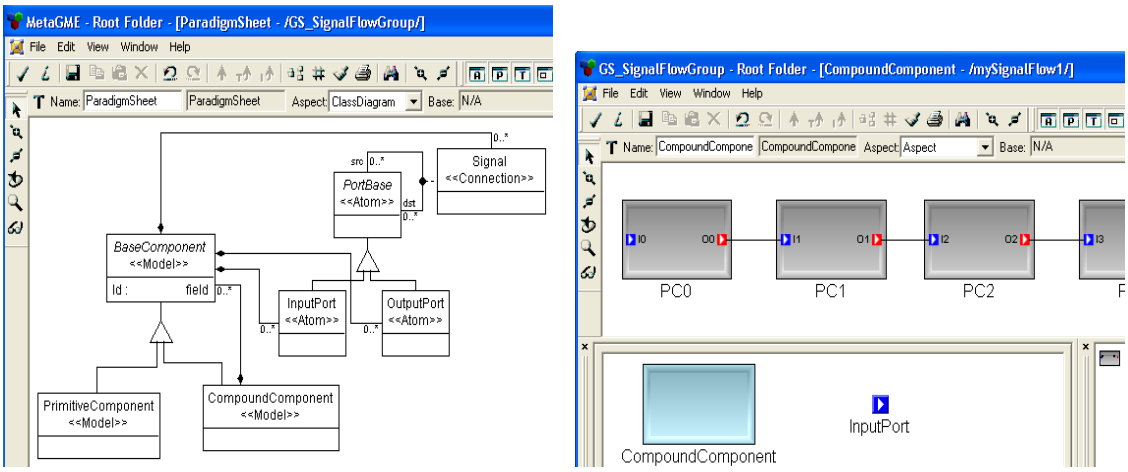
GME is a "meta-programmable" environment, which can be used to design domain specific modeling languages (DSMLs), and be reconfigured as a modeling environment for that specific domain. This reduces the significant cost of developing a special environment for each DSML (The cost of creating such a generic environment is itself high, but is amortized over the large number of domains). In addition to configuring the interface for modeling using the specified language, GME comes with a generic API for traversing and interpreting the domain specific models. For these reasons, GME provides a set of generic concepts that are abstract enough to cover several domains. The main steps in using GME for a model based development effort are

DSML specification, visualization and model interpretation, as described below.

DSML Specification. The "meta-language" of GME is based on UML [18] class diagrams, and allows users to specify the domain visually. Domain elements are specified using stereotyped UML classes, depending on the roles the objects may play in the domain, such as *Atoms*, *Models*, *Attributes* etc. In addition to the entities, their attributes and relations, additional static semantics can be added using OCL constraints. Figure II.5(a) shows a sample meta-model in GME.

Visualization. GME is an integrated environment that can be used both to specify a domain, and visualize it. Once the domain has been specified, a built in meta-interpreter interprets the meta-model, to generate a new modeling environment for GME. GME can be configured to run in the new domain, allowing users to visually specify domain models. Visualization rules in the meta-model specify what entities are visible, and how they appear in the new environment. Figure II.5(b) shows a sample domain model in GME.

Model Interpretation. GME allows the addition of domain specific model-interpreters to interpret the models created using the new domain. Traditionally, these are written in a general purpose language like C++ or Java, using specialized APIs. Recent advances in graph based model transformation have introduced more intuitive options for model transformation. GReAT [25] is a toolkit developed for GME that allows specifying model transformations graphically.



(a) Sample meta-model

(b) Sample domain-model

Figure 5: Modeling using GME

The generic nature of GME allows it to cover a wide range of usage scenarios. A wide variety of supporting tools for code generation, model transformation etc. make GME a powerful option for use in model based software development.

CHAPTER III

BACKGROUND: MODEL TRANSFORMATIONS

The transformation of the Platform-Independent Model into one or more Platform-Specific Model forms the crucial part of an MDA application. The success of the application relies upon the level of automation of the transformation and its correctness. But the transformation from PIM to PSM is not the only situation where model transformations are used. Some other uses of model transformations are:

- Transforming models for tool integration and porting to different platforms
- Transforming models between variants of a formalism, such as from iLogix Statecharts to Matlab Stateflow
- Generating an analysis model from a design model

Model transformations have been implemented using several different approaches, with graph transformations being the recent trend. Each approach must allow users to query an instance model and construct the transformed output model. In an effort to standardize this process, OMG introduced its specification standard for model transformations, called Query / Views / Transformation (QVT). In this chapter, we will review some of these approaches, with emphasis on graph transformations. We will also review a number of graph transformation based model transformation tools currently available.

OMG's MOF and QVT

OMG's Model Driven Architecture approach shifted the focus of models, form just an aid to communication and understanding, to being a key part of the software system. Much of the structure and behavior of the system is captured using models, which also encode platform information, and code is automatically generated from the models (at least in theory). Thus, automated model transformations play a key role in MDA. The Meta-Object Facility (MOF) [26] is OMG's standard environment for exporting models form one application to another, transporting models across a network and storing models in repositories. It provides a framework and set of services for the development and interoperability of model and meta-data driven systems. MOF includes a technology for defining meta-models, which is at the top of a four-layer meta-modeling architecture [17] (for UML modeling: MOF/UML/User Model/User Objects). The models may be stored, retrieved and rendered in different formats, including XMI (XML Meta-data Interchange), which is OMG's XML-based standard format for model transmission and storage.

One of the concerns in this methodology was the specification of model to model transformations. In April 2002, OMG issued a Request For Proposal (RFP) on MOF Query/Views/Transformations [27] to address the main issues related to the manipulation of MOF models. The RFP addresses technology neutral parts of MOF, related to queries on models, views on meta-models and transformations of models. The final adopted specification is available here [29].

Terms and Definitions

Some of the QVT-related terminology (from [28] and [29]) are listed below.

Query. A query is an expression that, when evaluated over a model, results in one or more instances of types defined in the source model or in the query language. A query is said to be *declarative* if it is described as relations between variables or entities, in terms of functions or inference rules, without any explicit implementation details. It is said to be *imperative* if it specifies explicit state manipulation. In addition, a query is called *selective* if it can only return elements from the queried model, and *constructive* if it can return other elements or values. The QVT review committee recommends the use of declarative, constructive queries [28].

View. A view is a model based on a portion of a source model that presents a particular aspect of the model. It may have the same information as the source model, or be reorganized to suit a particular task or user. A view cannot be modified independently of the source model, and is usually not persisted independently. Changes in a view are reflected back to the source model.

Transformation. A transformation generates a target model from a source model. A transformation is *unidirectional* if its application does not change the source model, and *bidirectional* if both the source and target models may be modified. In the latter case, there is a possibility of conflicting changes, which must be detected. Like queries, transformations may be specified declaratively or imperatively, or using a *hybrid* approach that combines the two.

Rule. A rule is a unit of transformation. It specifies of a selection of the source model, and the corresponding transformation to the target model. A set of rules

specify a transformation. A rule may contain a declaration and/or an implementation, depending on the type of the transformation.

Declaration. A declaration is a specification of a relation between the right-hand side and left-hand side models of the transformation. It may describe a transformation from one model to the other, or a bidirectional transformation from either side. A declaration may fully describe a transformation, or be associated with a separate implementation.

Implementation. An implementation is an imperative specification of how to generate the target model elements from the source model elements. It includes the steps required to explicitly construct the target model. Though it is typically unidirectional, bidirectional implementations are allowed.

Match. A match is said to have occurred when rule application identifies elements form the left-had side and/or right-hand side models as satisfying the constraints specified by the rule's declaration. When a match occurs, the target elements are created (or updated), driven by the declarative or implementation parts of the rule.

QVT Framework

QVT specifications are hybrid declarative/imperative in nature. The declarative part has a two-level architecture, which forms the framework for the execution semantics of the imperative part.

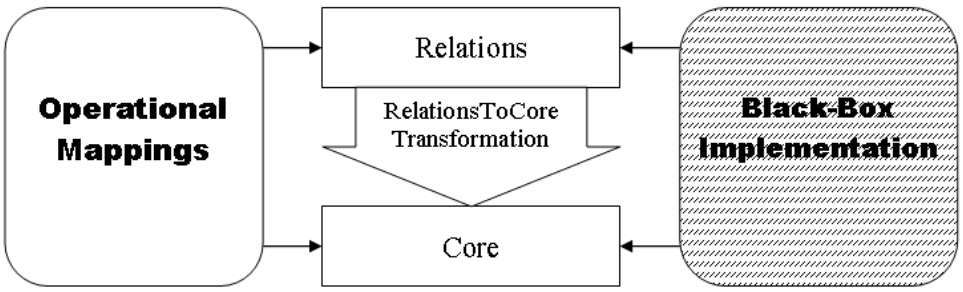


Figure 6: QVT Architecture [29]

Two-Level Declarative Architecture

The declarative part of the specification is structured in two layers, as shown in Figure 6. The first layer is the *Relations* metamodel and language which is used to declaratively specify relationships between MOF models. It supports complex object pattern matching and object template creation, and implicitly creates traces between model elements involved in the transformation. The second layer is the *Core* metamodel and language, which is defined using minimal extensions to EMOF and OCL. It is a small model/language that only supports pattern matching over a flat set of variables by evaluating conditions over those variables against a set of models. The Core language is as powerful as the Relations language, but due to its simplicity, its semantics can be defined more easily. However, transformations described using the Core language are more verbose, and trace models must be explicitly defined.

This architecture is analogous to the Java virtual machine, where the Relations language is like the Java language, and the Core language is like the Java Byte Code.

Imperative Implementation

There are two mechanisms for invoking imperative implementations of the transformations, the standard *Operational Mappings* language and the non-standard *Black-Box* implementations. The Operational Mapping language is specified as a standard way to provide imperative implementations. It provides OCL extensions with side effects that allows a procedural style of programming. When it is difficult to provide a purely declarative specification for populating a Relation, a Mapping Operations can be used to implement it. Transformations can be written entirely using Mapping Operations, and such transformations are called operational transformations.

Black-box implementations are plug-in implementations for MOF operations derived form Relations. They may be coded in any programming language with a MOF binding. They allow the use of libraries to calculate property values using domain specific algorithms, which may be difficult to express using OCL. They are also dangerous, since they have direct access to object references in the model.

Other Model Transformation Approaches

Apart from OMG's attempts to introduce some standardization in this area, several other approaches to model transformation exist today. A survey of popular approaches and their categorization is presented in [30]. Model transformation can be broadly classified into two major categories, *model-to-code* and *model-to-model*. Model-to-code transformations refer to the generation of textual artifacts from models in general. Some of the approaches to achieve model-to-code and model-to-model transformations will be discussed below.

Visitor based

This is a basic approach that uses some visitor mechanism to traverse an internal representation of the model, and write code to a text stream. There is usually some persistence structure for the models, along with an API for manipulating the models, which offers a visitor mechanism. The transformation itself must be hand-coded in some programming language. Some examples of this approach are Jamda [31] and UDM [32].

Though there have been efforts (such as OMG's MOF compliance) to provide standardized APIs, this approach is tedious, error prone and difficult to maintain, due to its direct use of programming language source code. The generated code usually contains static text that is not relevant to the model, which must be generated programmatically. This adds the size and complexity of the transformation code. The querying and transformation mechanism in this approach are imperative.

Template based

Template based methods have been used for text generation in a variety of areas for a very long time, mainly in web based applications where an active server would generate HTML pages from templates upon request. A template usually contains static text interspersed with some form of meta-code, which is interpreted during the generation process. This is usually in a declarative querying language (such as OCL or XPATH), which can be used to access information from a data structure. These portions in the template are replaced by data from the model, and printed out with the static text.

Templates closely resemble the final generated code, and are easier to understand and maintain than visitor based approaches. Since templates are text based and are not dependent on specific APIs, they are easier to port across different versions of

the models. However, they may not offer the advanced capabilities offered by specific APIs.

Direct model manipulation

Direct manipulation of an internal representation of models is a primitive approach, closely related to the visitor based code generation approaches. They are usually implemented as some form of object-oriented framework, with some API to manipulate the internal representation of the models. The transformations themselves are hand-written in some programming language such as Java or C++. Since direct references are made to the internal state of the models, this approach is imperative in nature.

Relational approaches

In this approach, relations between source and target elements are specified using declarative constraints. This form of specification is side-effect free. This satisfies the QVT recommendation that the querying language be purely declarative. They often support backtracking. Relational specifications can be interpreted bi-directionally. The declarative constraints can be given executable semantics, such as in logic programming. In [33], Gerber et. al. explore the application of logic programming to implement the transformation.

Graph Transformation based approaches

Model transformation approaches in this category draw on the theoretical work on graph transformations. These systems operate on typed, attributed, labeled graphs that represent UML-like models. A graph transformation rule or production p is a pair of graphs (L, R), as shown in Figure 7. Application of the rule $p = (L, R)$ means

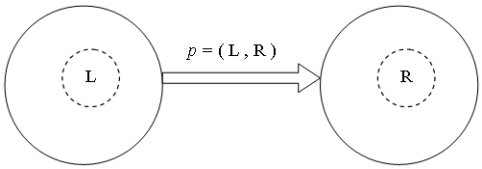


Figure 7: Rule-based modification of graphs [35]

finding a subgraph in the source graph that matches the left-hand side L, and and replacing it with the right-hand side R.

Approaches differ on how to delete L and how to create R. Other variations such as specifying additional conditions with the left-hand side pattern and multiplicities on edges and nodes are offered by different systems. In general, graph transformation rules are non-deterministic. However, most approaches allow a scheduling mechanism to enforce determinism, and offer other capabilities such as iteration. Graph transformation theory and tools will be discussed in greater detail later in this chapter.

Hybrid approaches

Hybrid approaches combine different techniques described above. For instance, a transformation rule may contain both declarative and imperative constructs, or be complemented with an imperative block. Transformation rules in tools such as GReAT [25] allow this usage, where a graph pattern may be supplemented with *Guard* or *AttributeMapping* blocks that contain imperative code that directly affects the state of the models.

Another hybrid approach is the Atlas Transformation Language (ATL) [34], which allows transformation rules to be fully declarative, fully imperative or a hybrid combination.

Taxonomy of Model Transformation

In [36], Mens and Van Gorp have compiled a taxonomy for model transformations, categorizing the main concepts in model transformations into groups that share similar qualities. This is especially useful to software developers for choosing an appropriate model transformation approach or tool for their needs, and to researchers for identifying areas for advancement. Some of the key concepts are discussed below.

Transformations in which the source and target models are in the same language are called *endogenous transformations*. Some examples of such transformations are optimization and refactoring, where the internal structure of a model may be changed to improve certain metrics, while preserving the semantics of the model. If the source and target models belong to different languages, the transformation is called *exogenous*. Code generation and migration of a model to a different platform fall under this category. It must be noted that, in the case of model transformation by graph transformations (as we will see shortly), while the source and target languages may be different, these languages themselves will be defined by a common *meta model*. It would be difficult to place such transformations under either category.

A model transformation is termed *horizontal* if the source and target models are at the same level of abstraction. In a *vertical* transformation, the source and target models reside at different levels. The applicability of this categorization is not always clear, as the distinction in the "abstraction level" of domain specific languages is not well defined. It is obvious for cases such as code generation, and could possibly be based on whether the transformation implicitly adds or abstracts away any specific implementation or algorithmic details in the model components.

A transformation that merely transforms the syntax of a model, for purposes such as importing and exporting to different formats, is called a *syntactic transformation*. A more sophisticated transformation that takes the semantics of the model into account is termed as a *semantic transformation*.

Mens and Van Gorp also discuss some concepts related to the success criteria and correctness of model transformations. The *syntactic correctness* of a model transformation depends on whether a well-formed source model results in a well-formed target model. *Semantic correctness* depends on whether the target model of a transformation has the expected semantic properties. Semantic correctness is crucial in cases where we would like to ensure that the transformation preserves certain behavioral properties of the models. Semantic correctness is usually a much harder question to answer than syntactic correctness. Some other properties related to correctness are *termination* (does the transformation terminate?) and *confluence* (is the result of the transformation the same for a given source model, for different executions of the transformation?).

Graph Transformation Concepts

Graph grammars and transformations have been an area of research in computer science since the 1970s, with origins from Chomsky grammars on strings, term rewriting and textual descriptions of visual modeling. Since graphs are an intuitive way of representing complex situations, graph transformations are applicable in many fields of computer science, such as formal language theory, pattern recognition and modeling of concurrent and distributed systems. Since the underlying structure of visual models is based on an abstract syntax graph, graph transformations can be intuitively applied for designing model transformations.

A graph transformation from that transforms a graph G to a graph H, written as $G \Rightarrow H$, usually involves the following steps [35] (as shown in Figure 8):

1. *Choose* a production p: $L \Rightarrow R$, with a left-hand side L and a right-hand side R, and find the occurrence of L in G.

2. *Check* the application conditions, if any, for the production p.

3. *Remove* the left-hand side L from G. If some parts of L are also present in R, those parts do not have to be removed from G. If there are any dangling edges, they must also removed or the rule is not applied. Let us call resultant graph D.

4. *Glue* the right-hand side R to the graph D. If R has parts of L that were not removed form G, these do not have to be added. Let us call the resulting graph E.

5. If p contains an additional embedding relation, then *embed* R into E according to this relation. This gives the final graph H.

There are two kinds of non-determinism that may be encountered in graph transformation systems:

1. several production rules may be applicable, and one of them must be chosen for application at random,

2. a chosen production rule may produce several matches, and one of them must be chosen at random.

Graph transformation systems may offer some control flow mechanism to resolve such non-determinisms, or may execute the transformation non-deterministically.

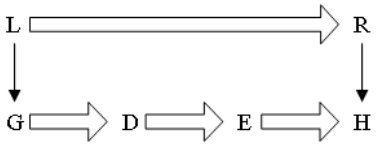


Figure 8: Graph transformation steps [35]

Graph transformation approaches can be distinguished based on key notions such as what is a "graph", how a pattern L is matched in the graph, and how the replacement of L by a graph R is defined.

Graph Transformation based Model Transformation

A domain specific modeling language consists of an abstract syntax that specifies the types of entities in the language and their relationships, usually specified using UML class diagrams. A domain specific model is a network of objects, where each object and its links belong to some type or association in the class diagram, called the meta-model. Thus, domain specific models can be looked at as graphs, or more precisely, typed multi-graphs with labels denoting the entities' types in the meta-model [37]. Thus, a model transformation task can be specified as a graph transformation, taking advantage of the mathematical foundations of graph transformations.

Type and Instance Graphs

A *type graph* [38] is a fixed graph TG of types, such as are labels or colors. A tuple (G, *type*) of a graph G with a graph morphism *type*: $G \rightarrow TG$ is called a *typed graph*. A type graph TG serves as an abstract representation of a class diagram (the meta-model of a domain specific language). The object diagrams of that domain are typed graphs, with a structure-preserving mapping to the type graph.

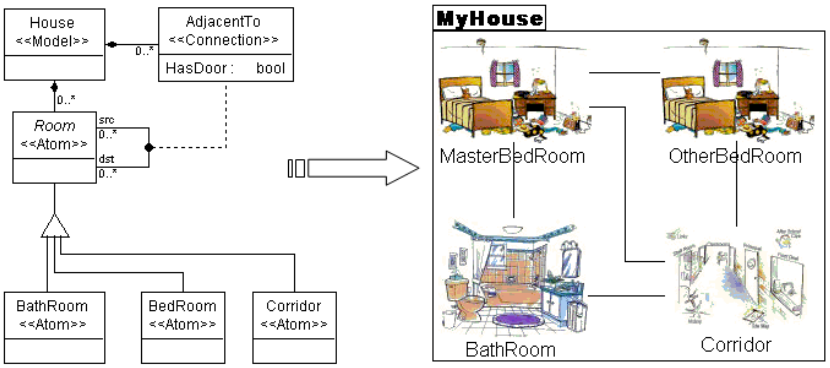


Figure 9: Example of a type graph and instance graph

Figure 9 shows a type graph on the left, in a stereotyped UML notation. The object graph (also called *instance graph*) on the right can be mapped on to the class diagram by defining $type(o) = T$ for each instance $o{:}T$ in the diagram. Similarly, the links between two objects can be mapped to the association class in the class diagram.

Types in Model Transformation

Model transformations can be described using graph transformation over typed graphs, as shown in Figure 10. The source model represented by a graph G_S that conforms to a type graph T_S, and the target model is represented by a graph G_R that conforms to a type graph T_R. Both T_S and T_R must conform to a common type graph T. Thus, the source and target graphs G_S and G_R are also instances of T. The transformation may produce intermediate graphs G_i, which are also instance graphs of T.

In a practical scenario, domains may be specified using UML, and domain models built using the domain specific languages. In a model transformation involving these domains, T would be the type graph for UML, and T_S and T_R the type graphs of the

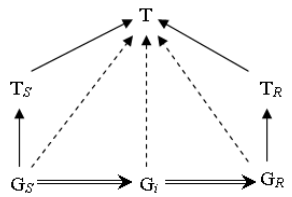


Figure 10: Model transformation using typed graphs

domains. Also, the intermediate graphs in the transformation do not have to conform to either domain (T_S or T_R), but must be instances of the UML type graph T.

Transformation Rules

A *graph transformation rule* p: $L \rightarrow R$ consists of a pair of TG-typed instance graphs L and R such that the $L \cup R$ is defined. This means that nodes or edges that appear both in L and R are bound to the same nodes or edges. A *graph transformation* transforms a graph G into a graph H, where the pre-condition is specified by L, and the post condition is specified by R. The transformation $G \Rightarrow H$ is performed in three steps:

(i) *Find* an occurrence of L in G

(ii) *Delete* from G the part matched by $L \setminus R$

(iii) *Paste* a copy of $R \setminus L$ to the result, giving H

Figure 11 shows an example transformation rule, where the left-hand side L matches two objects of type *BedRoom* that have a connection *AdjacentTo*. The instance graph G contains this subgraph, with a connection between the instances *MasterBedRoom*:*BedRoom* and *OtherBedRoom*:*BedRoom*. $L \setminus R$ is removed from

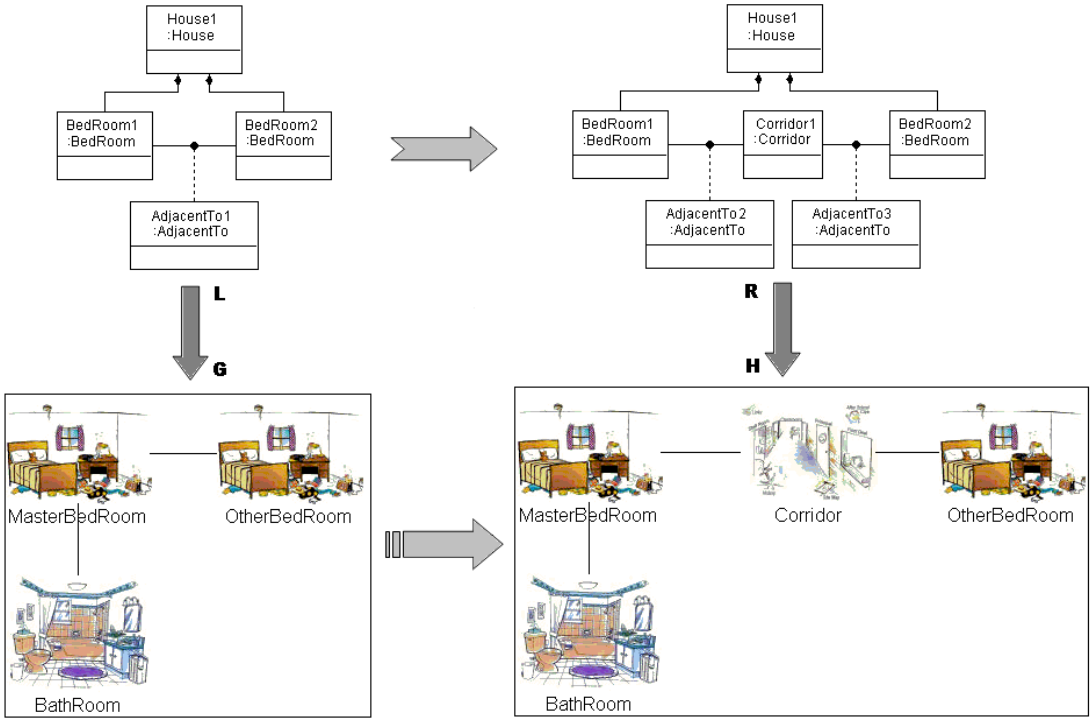


Figure 11: Transformation rule example

G, which removes the connection between *MasterBedRoom* and *OtherBedRoom* (*AdjacentTo1*:*AdjacentTo*). Finally, $R \setminus L$ is pasted on the result, which adds *Corridor1*:*Corridor* and the two associations. This produces the final transformed graph H.

Tools

In this section, we will review of some graph-transformation based model-transformation tools, including GReAT, PROGRES and AGG. We will start with a detailed study of GReAT, and its transformation specification language.

Graph Rewriting and Transformation (GReAT) [25] is a model transformation approach based on UML class diagrams. It is implemented using the GME modeling environment, as a modeling paradigm called *UML Model Transformation* (UMT) for specifying the transformations, and a set of interpreters that execute the transformations and perform other activities.

Cross-domain Transformations

GReAT uses the type graph / instance graph approach discussed in the previous section. The type graphs in this case are the UML class diagrams of the meta-models of the source models being transformed. GReAT allows any number of such type graphs to be associated with a single transformation. The UML class diagrams are typically *imported* from the meta-model definitions in GME, but can also be constructed within the GReAT domain. This allows interesting possibilities such as links between elements belonging to different domains, or temporary elements that exist only during the course of a transformation. Thus, the *type graph* for a GReAT transformation is a composite graph representing a large heterogeneous domain, which is a union of several domains. GReAT allows the specification of *cross-domain* transformations, where models belonging to different domains can be transformed simultaneously. The usual application of this feature is to match a pattern in a source model belonging to one domain, and use the information to construct a target model in another domain.

The GReAT Language

The GReAT language for specifying model transformations consists of three parts, as described below.

Pattern Specification Language. The pattern specification language is used to specify the graph patterns that will form the left-hand side L and the right-hand side R of the transformation rule. L and R are instance graphs whose elements can be mapped to the type graph, which is the heterogeneous graph described above. However, L and R are not specified as two separate graphs in GReAT. Instead, $L \cup R$ is specified as a single pattern. The parts belonging to $L \setminus R$ are marked to be deleted, and the parts belonging to $R \setminus L$ are marked to be created newly, as described using the transformation rule language.

Transformation Rule Language. The transformation rule language is used to completely specify the transformation rule around the graph pattern. Each element (node or edge) in the graph pattern is associated with an *action*. Three types of actions are allowed: *Bind*, *CreateNew* and *Delete*. Elements marked as *Bind* must be matched in the host graph, and can be considered as part of $L \cap R$ in the traditional representation of a rule. Elements marked as *CreateNew* must be newly created, and can be considered as part of $R \setminus L$. Elements marked as *Delete* must be deleted, and can be considered as part of $L \setminus R$.

A transformation rule in GReAT also has an *input interface* and an *output interface*. The input interface is always bound to some element of the host graph, and provides an initial binding for the application of the rule (this strategy, called *pivoted pattern matching* [39], significantly reduces the complexity of the pattern matching algorithm). Elements of the pattern can be connected to the output interface, and

will be passed on to the input interface of the next rule in sequence. The transformation rule may also contain additional pre- and post-conditions on the transformation, encoded in C++. Code placed in a *Guard* restricts the application of the rule only if certain conditions hold, and code placed in an *AttributeMapping* performs further changes on the transformed model.

Figure 12 shows a model transformation rule in GReAT. Note that the type graph is a union of two UML class diagrams, specifying two domains. The graph pattern in the transformation rule contains both the left-hand side L and the right-hand side R of the rule, and contains elements from both domains in the type graph. The element *Item* is marked with the action *CreateNew* (indicated by a tick mark on the bottom right). The rest of the elements are marked with the action *Bind*. The elements attached to the input port are bound to specific nodes in the host graph, that are supplied through the input interface. The rest of the pattern is matched in the host graph with this context. The *Domain1* portion of the pattern does not contain any *Delete* or *CreateNew* elements (i.e. the L and R parts are identical), and is left alone. The *Domain2* portion contains a *CreateNew* element (indicated by the tick mark in *Item*), which is added to the output *Domain2* model. The *Guard* and *AttributeMapping* elements allow exchange of specific data between the *Domain1* and *Domain2* models.

Sequencing and Control Flow Language. The application of a transformation rule requires the elements attached to its input interface to be supplied. This is called the *input packet*. The application of a rule finds elements in the input model corresponding to pattern elements in the transformation rule. These are called *bindings*. After the rule application, the bindings corresponding to pattern elements attached to the rule's output interface (called its *output packet)* are passed on to the next rule.

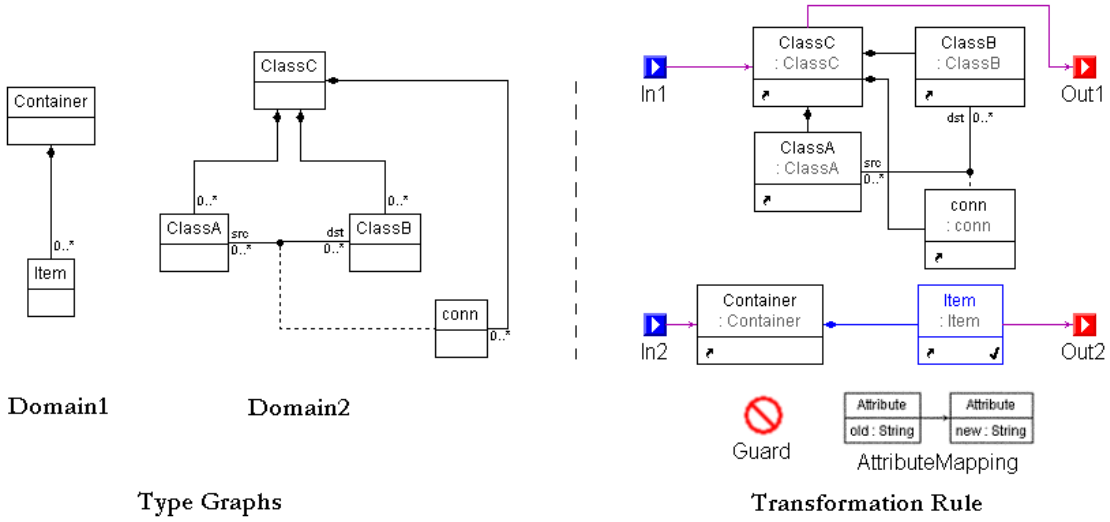


Figure 12: A transformation rule in GReAT

Several transformation rules can be sequenced in this way. The sequencing and control flow language allows users to construct a complete model transformation using a sequence of rules. It incorporates the following basic concepts.

(i) *Sequencing* - Transformation rules can be connected in sequence, by attaching the output interface of one rule to the input interface of the next rule. This results in the rules firing one after the other, with the output packets of the first rule sent to the second rule.

(ii) *Non-determinism* - Rules can also be connected in parallel, in which case the execution order of the rules will be non-deterministic.

(iii) *Hierarchy* - For the sake of simplicity and organization, transformation rules can be composed hierarchically, by placing them in *Blocks*.

(iv) *Recursion* - The output interface of a rule can be connected to the input interface of a rule that appears before it in the execution order. This allows recursive application of rules.

(v) *Conditional Execution* - Rules can be placed in special blocks called *Test* and *Case* blocks, to conditionally branch the execution of a chosen series of rules.

AGG

The Attributed Graph Grammar (AGG) system [40] [41] is a development environment for attributed graph transformation systems based on the algebraic approach to graph transformation. It implements a rule based visual language, and aims at the specification and prototypical implementation of applications with complex graph-structured data. AGG is implemented in Java, and provides a Java API for integration with other Java based tools. The AGG graphical user interface provides a visual layout similar to UML object diagrams.

Transformation Concepts

Attributed graphs are special kinds of graphs, which are extended to carry attributes as a (*name*, *value*) pair. Graphs in AGG can be attributed by Java objects instantiated from Java classes. A graph transformation in AGG consists of a type graph, a start graph that is an instance of the type graph, and a set of transformation rules. The type graph is used to specify the class of graphs that can be used in the transformation, which defines all the allowed node and edge types, their relations and their attribute types. The start graph initializes the system, and the rules describe the actions to be performed. Both the start graph and the transformation rules may be attributed by Java objects (which can be instances of Java classes from JDK libraries or user-defined classes) and expressions. In addition, transformation rules can contain negative application conditions.

AGG employs the algebraic approach [35] to graph transformation, to apply the transformation rules. A graph rule may modify a graph by adding or deleting nodes

and edges, and by performing computations on a nodes attributes. The transformations can be executed in two modes. In the *debug* mode, the user selects a single rule to be applied once to the current graph. In the *interpretation* mode, a sequence of rules are applied non-deterministically, until no more matches can be found for any rule.

Validation

AGG offers some support for validating the models in a transformation. There are three types of techniques . A *graph parser* is provided to check if a given graph conforms to its type graph. *Critical pair analysis* can be used to check if the rules are confluent (the order of execution of the rules does not affect the result), by finding conflicting rules. Rule applications are conflicting under three conditions: when one rule deletes an object that is matched by another rule; when one rule creates an object that is prohibited by a negative application condition in another rule; and when one rule changes the attributes that are matched in another rule. Finally *consistency conditions* can be used to describe some properties that must be invariant in the application of a transformation.

PROGRES

PROGRES [42], short for *PRO*grammed *G*raph *R*ewriting *S*ystems, is an integrated environment for an executable specification language based on graph rewriting systems. PROGRES incorporates a visual programming language for graph transformations that can include textual syntax, and also allows imperative programming of rule application strategies [43].

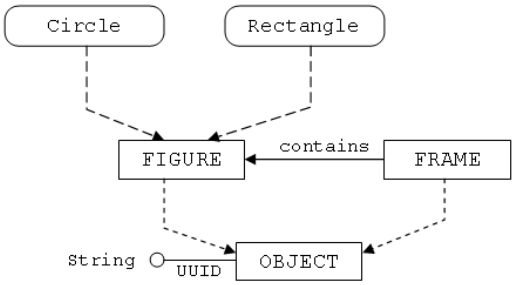


Figure 13: A PROGRES schema example

Graph Schemata

A *graph schema* in PROGRES is a set of syntactic constructs that define a particular class of graphs. A graph schema consists of node types, relationships and attributes. In addition, we can have node classes which define types of node types. Hierarchies of node classes can be built by multiple inheritance. Figure 13 shows a PROGRES schema, where `FIGURE` and `FRAME` are subclasses of `OBJECT`. `Circle` and `Rectangle` are subtypes of `FIGURE`, and `UUID` is an attribute of `OBJECT` that is of type `String`. In addition to the graphical constructs, external types and definitions of functions can be imported from text files.

Graph Queries and Transformations

A graph transformation rule in PROGRES is called a *production*. Productions have a left-hand side and a right-hand side graph pattern. The left-hand side consists of *subgraph tests* and *queries*. A subgraph test searches for the existence (or non-existence) of a subgraph in a host graph, and may contain additional restrictions on the attributes of nodes. Complex *queries* can be constructed by sequencing subgraph tests in a textual specification, optionally including imperative statements to enforce deterministic results.

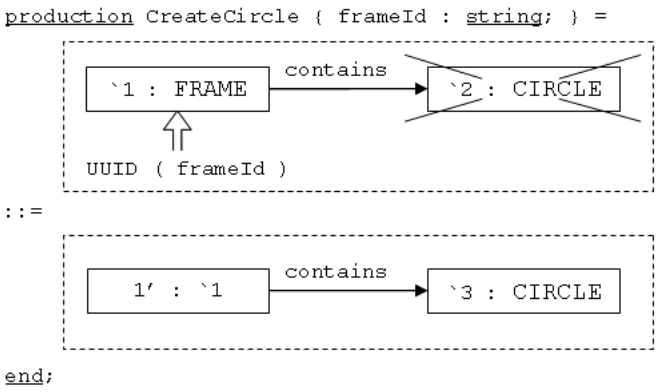


Figure 14: Specification of a graph transformation rule in PROGRES

Figure 14 shows a sample PROGRES production. The left-hand side is a query that contains a `FRAME`, with the arrow indicating a restriction on the `UUID` attribute. It also indicates that the `FRAME` should not contain a `CIRCLE`. The production selects a `FRAME` node that is not adjacent to a `CIRCLE` node (through a `contains` edge), and then replaces it with the image on the right-hand side. The right-hand side nodes that are bound to left-hand side nodes are left as they are. In this case, the `FRAME` node is preserved, and a `contains` edge and a `CIRCLE` edge are added to the graph. The production is parameterized, and must be instantiated an actual value for `UUID`.

Productions can be composed into *transactions*, which can specify complex graph transformations, using a textual specification. Transactions provide control structures for non-deterministic branching, sequential composition and recursive execution.

Graph transformations based behavior specification

Domain specific modeling languages (DSMLs) have found great application in the development of software for embedded systems. But most of the effort in the specification of these languages is on defining the static syntax and integrity constraints of the domain. The behavior of the model, its *operational semantics*, is usually implemented using hand written code. The behavior of a model may be regarded as the change in the state of the model over time or over a sequence of inputs. Since the model can be represented as graphs, it is possible to specify the evolution of a model over time as a transformation on its graph structure. Thus, graph transformations offer a way to specify the operational semantics of DSMLs.

It must be noted here that there is a distinction between using a model to describe the behavior of a system, and specifying the behavior of a modeling language. For instance, the behavior of a reactive system can be modeled using the Statecharts language. However, it is implied that the behavior of Statechart models is already specified and understood. In fact, the behavior of the Statecharts language is rarely specified formally, and different implementations vary greatly in how they realize the behavior of identical Statechart models [13]. This stresses the need for a formal behavior specification.

Transformation into Behaviorally Equivalent Transition System

One approach to provide semantics to a language is to provide a transformation into a canonical form such as finite automata, that already has a precise semantics. This is usually done using graph transformations [44] [15]. The graph transformation then becomes the "behavior specification" for the language.

Flattening of Statecharts

Providing formal semantics for UML diagrams, mainly UML Statecharts, has been the most popular target for this technique. In one of the early works in this area [44], Gogolla and Parisi-Presicce provide graph transformation rules to obtain a normal form representation from nested UML state diagrams, to give it precise semantics in terms of automata. They later extended their approach to other UML diagrams in [22].

In [15], an Statechart models are transformed into an intermediate representation called Extended Hybrid Automata [14], by resolving transitions that cut across hierarchies in the Statechart. The intermediate form can then be converted by a straightforward code generation step into a PROMELA model, which can be verified using the SPIN [8] model checker.

The advantage of these approaches is that the canonical form usually lends itself to verification and analysis in a practical sense, when verifying the original model may be too complicated. However, the correctness of the transformation into the canonical form is crucial. Any error in the specification or application of the transformation can result in a low level model that not truly represent the intended behavior. In [45], an approach is presented to verify whether the intermediate EHA model generated from a Statechart model truly represents the behavior of the original model.

Semantic Anchoring

Semantic anchoring [46] is a technique for specifying the behavior of domain specific modeling languages (DSMLs), being developed at ISIS, at Vanderbilt University. It stems from the observation that a large number of component behaviors may be described using a small set of behavioral abstractions such as Finite State Machines and Timed Automata. These behavioral abstractions are called *semantic units*, and

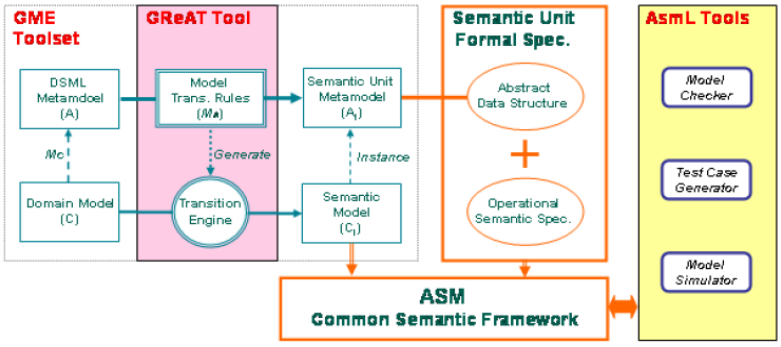


Figure 15: Overview of Semantic Anchoring

we can assume that their behavior is well understood and precisely defined. The behavior of a higher level DSML can be specified as a transformation from the DSML to a chosen semantic unit. This transformation process is called *semantic anchoring*.

Figure 15 shows an overview of the semantic anchoring architecture. It is built within the GME [3] framework, and the transformations are specified using GReAT [25]. The behavior of the semantic unit is described using a formal notation, such as Microsoft's Abstract State Machines Language (AsmL) [47]. The semantic anchoring transformation creates a representation in terms of the semantic unit, from which AsmL code can be generated. This is used with other AsmL tools for verification and analysis.

Meta-level Behavior Specification

We have seen that the abstract syntax of DSMLs can be specified using typed, attributed graphs. The approaches discussed above specify the behavior by transforming the instance models into a behaviorally equivalent transition system. The resultant transition systems are analyzed or verified, to prove properties about the instance model. Another approach is to make the behavior description a part of the meta-model [48]. In this approach, the behavior of a visual language is specified using

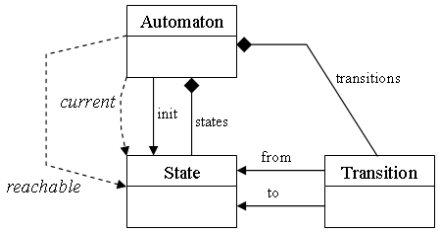


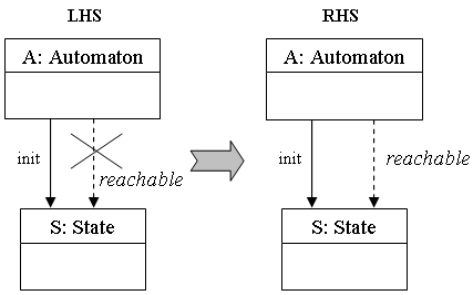
Figure 16: Meta-model of a Finite Automaton [48]

graph-transformation rules on the meta-model of the language, as an evolution of the system. This is presented as a *meta-level analysis technique* in [48].

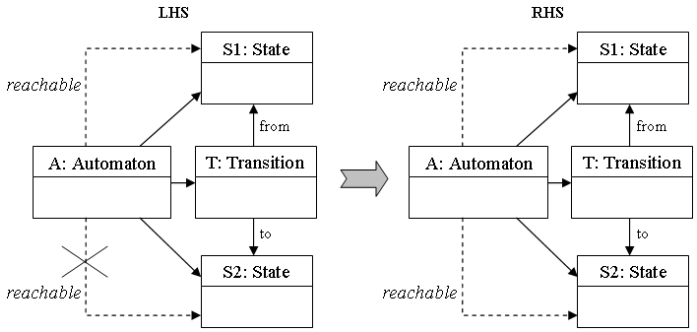
The specification of behavior in the meta-model introduces the notion of *dynamic* and *static* elements in the meta-model. The dynamic elements can be altered during the execution of the model, and are essential for modeling the behavioral aspects of the language. Figure 16 shows a meta-model for an automaton, with the dynamic elements indicated by dashed-lines and italicized text. In this model, *reachable* states and *current* states are behavioral properties.

Figure 17 shows two rules that specify the behavioral property of *reachability*. Figure III.17(a) specifies that the initial state of the system is reachable. The graph transformation rule finds an initial state that does not have a *reachable* edge (indicated by a negative-application condition), and adds the edge. Figure III.17(b) specifies that if a state *S1* is reachable, and has a transition to a state *S2*, then *S2* is also reachable. This is realized by the graph transformation rule that finds such a state with no *reachable* edge, and adds the edge to it. This example shows the function of the dynamic elements of the meta-model in specifying its behavior.

The verification and analysis of behavioral properties is still performed by automatically generating a transition system from the behavior specification. However, since the behavior specification is in the meta-model of the language, it is independent



(a) Initial state



(b) Other states

Figure 17: Specifying reachability using graph transformations [48]

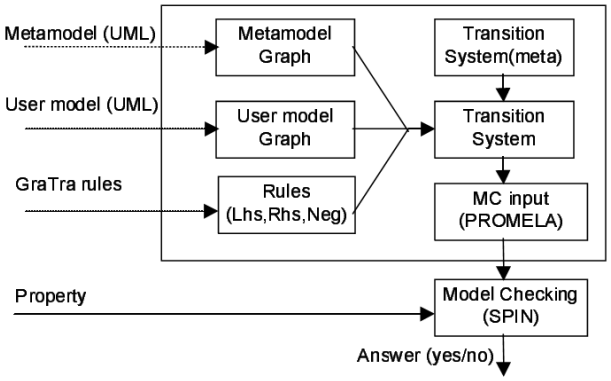


Figure 18: Architecture of CheckVML [49]

of the analysis tool. The CheckVML [49] incorporates this architecture, as shown in Figure 18.

CHAPTER IV

BACKGROUND: VERIFICATION

Verification is the process of determining whether a system satisfies a certain specification. One form of verification is *testing*, which is performed during system execution, dynamically checking its behavior under a specific test condition. But testing alone cannot prove the absence of a defect, or that the system satisfies a certain property under all conditions. *Formal verification* is the process of proving (or disproving) the correctness of a system with respect to a formally specified property, using formal methods of mathematics.

The formal verification of a system consists of three main parts. Firstly, there must be a verifiable *model* of the system. This may be in the form of source code, an algorithm represented in some model of computation, or a high-level model of the system. Secondly, there must be a formal *specification* of the property to be verified. The property may be specified as a desirable value or range of values of the system's variables, or using some form of logic such as first order logic, linear temporal logic (LTL) or computational tree logic (CTL). Finally, there must be a *verification strategy*, or an approach to determine if the system satisfies the specified property. Some popular strategies are model checking [16] (such as state space exploration) and logical inference (such as using a theorem prover [51]).

Formal verification greatly increases the quality of software systems, and the confidence we have in critical software, but it comes at a high cost. It is desirable for all systems, but essential for safety critical systems.

Hoare's Axioms

One of the first attempts at introducing formal verification techniques in programming was by Hoare [50] in 1969. Hoare tries to provide a logical basis for proofs of the properties of a program, using a set of *axioms* and inference rules. He introduced the notation $P\{Q\}R$, which is read as "If the assertion P is true before the initiation of a program Q, then the assertion R will be true on its completion".

Along with a set of axioms for integer arithmetic, Hoare's proof system includes the following axioms and inference rules [50]:

1. *Axiom of assignment.* If *P(f)* is true before the assignment of the value of the expression f to the variable x, then *P(x)* must be true after the assignment. This is expressed more formally as:

$$\vdash P(f)\ \{x{:=}f\}\ P(x)$$

2. *Rules of Consequence.* If the execution of a program Q ensures the truth of an assertion R, then it also ensures the truth of all assertions implied by R. Similarly, if the precondition P for a program Q ensures the truth of R, then any assertion that logically implies P also ensures the truth of R with the program Q. These are formally written as:

$$\text{If } \vdash P\{Q\}R \text{ and } \vdash R \supset S \text{ then } \vdash P\{Q\}S$$
$$\text{If } \vdash P\{Q\}R \text{ and } \vdash S \supset P \text{ then } \vdash S\{Q\}R$$

3. *Rule of Composition.* If the result of a statement Q_1 is identical to the precondition of a statement Q_2, then a program composed of the two statements will produce the same result as the second statement if the precondition to the first statement is satisfied. This is formally written as:

$$\text{If } \vdash P\{Q_1\}R_1 \text{ and } \vdash R_1\{Q_2\}R \text{ then } \vdash R_1\{Q_1;\ Q_2\}R$$

4. *Rule of iteration.* An iteration statement using the notation "**while** B **do** S" requires B to be true at the start of the iteration, and executes the statement S until B becomes false. This is expressed formally as:

$$\text{If } \vdash P \wedge B\{S\}P \text{ then } \vdash P\{\textbf{while } B \textbf{ do } S\} \neg B \wedge P$$

Using these axioms and inference rules, it is possible to construct a correctness proof for programs written in this simple programming language, by making assertions about the values of variables as a result of executing the program. Using machine-independent axioms, programs can also be verified for portability and interoperability. However, there are some difficulties in proving program correctness in this way. In languages permitting side effects, procedures must be proved to be side-effect free. Hoare's axioms and rules assume that the program will terminate, but do not give a proof that it will. They are also limited to bounded integer arithmetic and do not handle reals.

In spite of the limitations, Hoare's work was seminal in the field of program verification. It led to advances such as using verification condition generators to produce a set of subgoals, which can assist in correctness proving using automated provers. Another use of this technique was to provide formal definitions for programming languages, improving their design and making them easier to implement.

Automated Theorem Proving

Automated theorem proving [51] is the process of using a computer program to prove mathematical theorems. More accurately, it is the use of computer programs to show that a *conjecture* is a *logical consequence* of a set of *axioms*. The complexity

of the problem of deciding the validity of a theorem depends on the underlying logic used. For the case of propositional logic, it is decidable but NP-complete [51]. For the case of first order logic, it is decidable given unbounded resources [51]. A first order theorem prover may fail to terminate while searching for a proof for an invalid statement. In spite of these limitations, automated theorem provers have been used to prove some difficult theorems.

Interactive theorem provers are semi-automatic theorem provers, that really on human input to guide the proof of a theorem. The simplest form is a *proof checker* that verifies each individual proof step, to check if a proof for a theorem is valid. Some level of automation with some human interaction avoids the necessity to list all proof steps, and simplifies some search issues, with the theorem prover performing the bulk of the task. Their success largely depends on the nature of the interface between the theorem prover and the human user.

Compiler verification

Formal verification of software is usually performed on the source code or some high level model, and rarely on the machine code that is implemented on the system. A fault in the compiler that generates the machine code would neutralize any useful result proved by the verification of the source code. In model based software engineering, code is often generated from high-level models, which is then compiled and executed. The high level models are usually verified formally, but errors in the code generator could mean that the generated code does not truly represent the verified model. In all these cases, the correctness of the code generator or compiler is crucial. The verification of the correctness of the generator is called *compiler verification*.

The strategies to prove some properties about compilers vary in different aspects. Based on the target of the verification, they can be categorized as:

(i) *Verification of the compiler.* The compiler is itself treated as a program that must meet certain specifications. The requirements of the compiler are formally specified, and a proof is generated, usually using semi-automated theorem provers. The compilers that have been proven correct using this method are usually simple, non-optimizing compilers operating on simplified source and target languages. The compilers commonly used to deploy software on embedded systems are usually much more complex and perform several optimizations.

(ii) *Verification of the generated code.* In this approach, we check whether the code generated by a compiler truly represents the source code. The verification is done by a separate program, which implements some criteria to establish that the generated code is correct. The verifier is usually assisted by some annotations generated by the compiler along with the code. This is usually much simpler than proving the correctness of the compiler itself. However, every instance of the generated code must be verified individually.

The verification strategies may also vary on the degree of correctness we wish to establish, as described below:

(i) *Full correctness.* We may wish to establish the full correctness of the compiler, with a complete specification, and proving that the compiler completely satisfies its specification. In the case of the generated code, this means proving that it is behaviorally identical to the source code. Proving the complete correctness of the compiler is a difficult problem. Strategies such as bisimulation may be used to prove the equivalence of the source and target code instances of a particular run of the compiler, but proving program equivalence is also fairly difficult, and undecidable in many cases.

(ii) *Partial correctness.* Sometimes it may be satisfactory to prove that some safety properties will hold, such as type-safety or memory-safety. We may also be

interested to prove that while the behavior of the generated code may not be identical to the behavior of the source, it behaves identically with respect to a particular property of interest. Proving partial correctness of this nature is usually easier and more practically useful.

Certifiable Program Generation

In [52], Denney and Fischer consider the problem of verifying generated code to ensure that some safety properties are satisfied. Based on the categories discussed above, their approach falls under partial verification of generated code. Firstly, the target of the verification is the generated code, and not the code generator itself. Secondly, the verification is restricted to a set of safety properties, and not the general semantic correctness of the generated code. It is based on the concept of *proof carrying code* [53].

Figure 19 shows an overview of the architecture for certifiable program generation. The objective is to give a *safety certification* to a piece of generated code, based on a set of safety properties. A *safety policy* is a set of proof rules that certify that the program does not violate the safety conditions during execution, such as exceed the bounds of an array or use variables without initializing them first. The code generator is extended such that it produces logical annotations that are required for formal safety proofs in a Hoare-style framework. The annotations serve as lemmas which are used with a verification condition generator to determine the safety obligations. This simplifies the proof construction, which can be done using off-the-shelf theorem provers.

In a framework such as the one shown in Figure 19, there are always components that must be *trusted*. These are components that are themselves not verified during the certification process, and their result must be accepted as correct. In Figure 19,

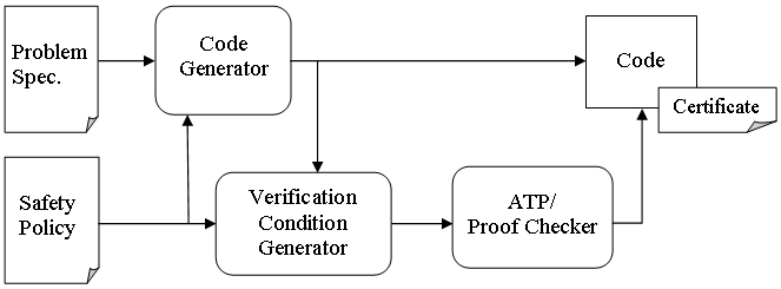


Figure 19: Certifiable Program Generation Architecture [52]

the trusted components are the safety policy, the verification condition generator and the theorem prover. A fault in one of these components will not be captured during the certification process, and the safety certificate will not be valid. These components are usually simple enough to be reasoned correct, and carry a high degree of trust. The theorem prover may be split into a theorem prover and a proof checker, in which case the proof checker, which performs a more straightforward task, must be trusted.

Bisimulation and Operational Equivalence of Programs

One approach to verifying the success of a code generation is to check the equivalence of the generated code to the source specification, based on some criteria of equivalence. In a practical scenario, we are usually interested in the behavioral equivalence. To verify this, we need some way to specify the behavior of a program correctly, and a way to compare two behaviors.

Operational Semantics

There are three main approaches to giving meanings to programming languages, namely, denotational, axiomatic and operational semantics [54]. In *denotational semantics*, the meanings of programs are defined abstractly, using some suitable mathematical structures. In *axiomatic semantics*, program properties are expressed using some form of logic. *Operational semantics* specifies the behavior of programs during execution.

One of the significant advances in behavior specification came with the development of the *structural approach* to operational semantics, by Plotkin [55], which proposed the use of transition systems to define the behavior of programs.

A *Labeled Transition System* (LTS) is a tuple $(S, A, \rightarrow)$, where

S is a finite set of *configurations*,

A is a finite set of *actions* (or *labels* or *operations*),

and $\rightarrow \subseteq S \times A \times S$ is a *transition relation*.

Relations between programs can be studied as relations between labeled transition systems, such as *simulation preorders* and *bisimulation*.

Bisimulation

Bisimulation [56] is an equivalence relation between Labeled Transition Systems (LTS), which can conclude whether the two systems will behave identically. In other words, if two systems have a bisimulation relation, then one system simulates the other and vice versa. Given an LTS (S, Λ, $\rightarrow$), a relation R over S is a *bisimulation* if:

$$(p, q) \in R \text{ and } p \xrightarrow{\alpha} p' \text{ implies that there exists a } q' \in \mathrm{S},$$
$$\text{such that } q \xrightarrow{\alpha} q' \text{ and } (p', q') \in R,$$

and conversely,

$$q \xrightarrow{\alpha} q' \text{ implies that there exists a } p' \in \mathrm{S},$$
$$\text{such that } p \xrightarrow{\alpha} p' \text{ and } (p',\, q') \in R.$$

Bisimilarity is the union of all bisimulations. Bisimilarity is generally accepted as the finest form of behavioral equivalence.

CHAPTER V

TOWARDS VERIFYING MODEL TRANSFORMATIONS

Overview of this Paper

An end-to-end model based software development effort often involves the use of multiple domain specific languages (DSMLs), to perform different activities at different stages of development. For instance, Statecharts may be used as a design language, and Extended Hybrid Automata (EHA) may be used as an analysis language. The automated transformation between such domain specific models plays an important role in such development efforts. In such a scenario, it is crucial that the automatically transformed models preserve certain behavorial properties of the original model. The analysis of the automatically transformed EHA model is useful only if it reproduces the behavior of the Statechart design model.

In this paper, we address the verification of behavior preservation in such transformations. The general problem of verifying behavior equivalence is undecidable. This paper proposes an *instance based* approach, where we verify the equivalence of the specific instances generated by a particular execution of a transformation. Behavior equivalence is determined by establishing a bisimulation between the instance models. The paper describes extensions to the transformation framework to maintain relations between source and target elements, and use an automated bisimulation checker to verify if a bisimulation exists.

The paper begins by introducing the problem being addressed, and covers background information on Statecharts and EHA. The transformation between Statecharts and EHA is described in detail, along with the extensions that facilitate bisimulation checking.

Introduction

Domain specific modeling languages (DSMLs) greatly simplify the task of the system designer, presenting a higher level of abstraction that is easy to work with. DSMLs also facilitate analysis by providing an appropriate abstraction. However, it is not always the case that the same language is suitable for both design and analysis. For instance, Statecharts are very powerful for designing concurrent systems, but their analysis is usually not simple. Extended Hybrid Automata (EHA) were introduced in [14] as an intermediate, simpler language with a more restricted syntax. Subsequent work [15] has shown that this intermediate format can be used to generate verification models that may be verified using model checking tools such as SPIN [16].

Graph transformation has been suggested as a powerful and convenient method for transforming design models into analysis models. The transformation must ensure that the analysis model preserves the semantics of the design model, and truthfully represents the design. As a first step towards this goal, it would be useful to establish that the transformed model is semantically equivalent to the source model, with respect to the property we wish to verify. In this paper, we study this notion of equivalence between the two graphs, and a way to check if there exists a bisimulation relation between the graphs. If it is possible to prove that the analysis model behaves in exactly the same way as the design model with respect to a certain property, then we can conclude that checking for the property in the analysis model is equivalent to checking for the same property in the original design model. In the following sections, we will go through the basics of graph transformation principles and tools, and demonstrate our approach to checking the equivalence using Statechart models and EHA models.

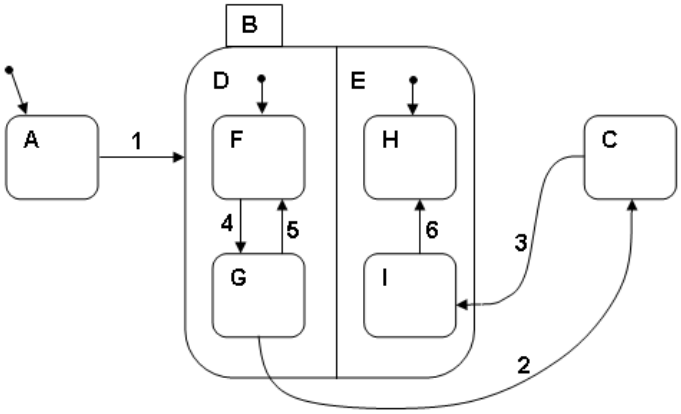


Figure 20: A sample Statechart model

Background

In this section, we review some background information.

Model Integrated Computing

Model Integrated Computing (MIC) [23] is an approach to system development using domain specific models to represent the architecture and behavior of the system and its environment. The development process involves the creation of a meta-model that defines the abstract syntax of the domain, from which a Domain Specific Design Environment (DSDE) is generated. The DSDE can be used to create domain specific models. These models are usually transformed to other formats, such as executable code, or to perform analysis. The MIC tool suite containing GME [57] and GReAT [58] were used in developing the examples for this paper.

GReAT

The transformations in this paper will be written in GReAT [58], a language for specifying graph transformation rules. GReAT belongs to the class of practical graph

transformation systems such as AGG [59], PROGRES [62] and FUJABA [60]. It uses UML and OCL to specify the domains of the transformation.

GReAT allows users to compose source and target meta-models by defining temporary vertex and edge types that can span across multiple domains and will be used temporarily during the transformation. This enables us to tie the different domains together to make a larger, heterogeneous domain that encompasses all the domains and cross-links. This feature plays an important role in our approach to verifying transformations.

Statecharts

State machines, based on Harel's Statecharts [10] are used in UML to represent the reactive behavior of systems. State machines are constructed from *states* and *transitions*. States may be simple, composite or concurrent. States may be connected by directed edges called transitions. Transitions connecting states contained in different levels of hierarchy are called *inter-level* transitions. Figure 20 shows an example of a Statechart. Transitions 2 and 3 in the figure are inter-level. A *state configuration* is a maximal set of states that the system can be active in simultaneously. State configurations are closed upwards, meaning that if a system is in a state A, then it must also be in A's parent state. Some valid state configurations in Figure 20 are {A}, {B, F, H} and {B, G, I}.

EHA

Extended Hierarchical Automata (EHA) were introduced as an alternate representation to provide formal operational semantics for Statechart diagrams. EHA offer an alternative simplified hierarchical representation for Statecharts that helps in correctness proofs [61]. The meta-model for EHA in UML is shown in Figure 21.

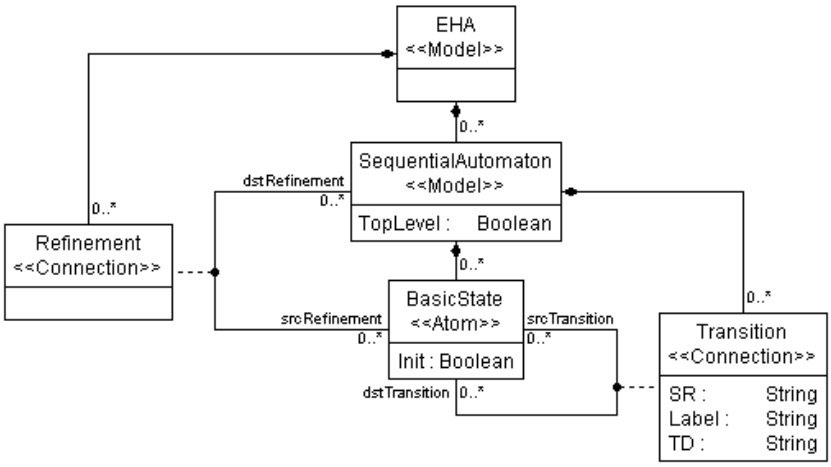


Figure 21: EHA meta-model in UML

Each Statechart model can be represented by one EHA model. Every compound state in the Statechart model is represented by a Sequential Automaton in the EHA. There is one top level Sequential Automaton for the EHA, which represents the initial automaton. Each state in the Statechart has a corresponding Basic State in the EHA. If a state is compound in the Statechart, then it is further "refined" into a Sequential Automaton in the EHA, which will contain Basic States corresponding to all the states within the compound state in the Statechart. Similarly, these states may be refined further.

Transitions in EHA are always within a single Sequential Automaton, i.e. there are no inter-level transitions in an EHA. Inter-level transitions in Statecharts are elevated based on the scope of the transition, to the Sequential Automaton representing the lowest common ancestor of the start and end states of the transition in the Statechart. EHA transitions have special attributes called "source restriction" and "target determinator", which keep track of the actual source and target of the transition in the Statechart. The conversion of Statechart models into EHA models will be discussed in detail in the next section.

Verifying graph transformations

Graph transformation systems such as GReAT allow users to transform models of one meta-model to models of another meta-model using a collection of pattern matching rules. However, it is not certain whether the output of the transformation preserves the semantics of the source model that we intend to analyze. Important semantic information may easily be lost or misinterpreted in a complex transformation, due to errors in the graph rewriting rules or in the processing of the transformation. We need a method to verify that the semantics that we are interested in analyzing are indeed preserved across the transformation.

We propose an approach to check whether the semantics of the input model were preserved in the output model of a transformation. We are *not* trying to prove the correctness of the graph transformation rules in general, but check if a *particular* generated model is a valid representation of a *particular* source model, in order to verify a particular property about the source model. We accomplish this by defining an equivalence relation between objects of the input and the output model, and use this to check if the two models are similar in behavior.

Bisimilarity

Two systems can be said to be *bisimilar* if they behave in the same way, i.e. one system simulates the other and vice-versa. A bisimulation relation can be defined formally as follows. Given a labeled state transition system (S, Λ, $\rightarrow$), a *bisimulation relation* is defined as an equivalence relation R over S, such that for all p, $q \in$ S, if (p, q) is in R, and for all p' $\in$ S and $\alpha \in \Lambda$, $p \rightarrow^{\alpha} p$' implies that there exists a q' $\in$ S such that $q \rightarrow^{\alpha} q$' and (p', q') is in R, and conversely, for all q' $\in$ S, $q \rightarrow^{\alpha} q$' implies $p \rightarrow^{\alpha} p$' and (p', q') is in R.

Though this definition is given in terms of a single set S, we can think of equivalence of two transition systems in terms of a global set containing both the system's states. In our approach to verifying whether the semantics are preserved across a transformation, we will check whether there is a bisimulation relation between the source model and the target model.

Transforming Statecharts into EHA

The EHA notation for Statecharts can be obtained by a graph transformation process [61]. The basic steps of the transformation are listed below:

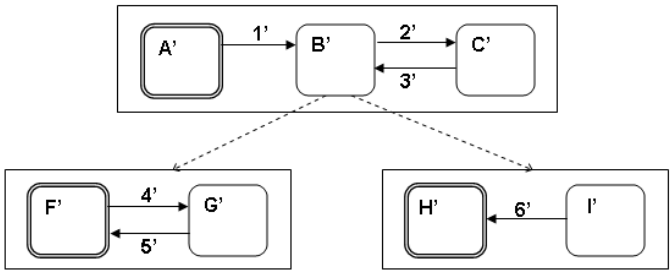
1. Every Statechart model can be transformed into an EHA model, with one top level Sequential Automaton in the EHA model.

2. For every (primitive or compound) state in the Statechart (except for regions of concurrent states), a corresponding basic state is created in the EHA.

3. For every composite state in the Statechart model, a Sequential Automaton is created in the EHA model, and a "refinement" link connects the Basic State in the EHA corresponding to the state in the Statechart, to the Sequential Automaton in the EHA that it is refined to.

4. All the contained states in the composite state are further transformed by repeating steps (ii) and (iii). The top level states in the Statechart will go into the top level Sequential Automaton in the EHA.

5. For every non-interlevel transition in the Statechart model a transition is created in the EHA between the Basic States corresponding to the start and end states of the transition in the Statechart model.



Transition Label	SR	TD
2'	G'	
3'		I', F'

Figure 22: Sample EHA model

6. For every inter-level transition in the Statechart model, we trace the scope of the transition to find the lowest parent state s_P that contains both the source and the target of the transition. A transition is created in the EHA, in the Sequential Automaton corresponding to s_P. The source of the transition in the EHA is the Basic State corresponding to the highest parent of the source in the Statechart that is within s_P, and the target in the EHA is the Basic State corresponding to the highest parent of the target in the Statechart that is within s_P. The transition in the EHA is further annotated, with the "source restriction" attribute set to the basic state corresponding to the actual source in the Statechart, and the "target determinator" set to the basic state corresponding to the actual target in the Statechart.

Figure 22 shows the EHA model obtained by transforming the Statechart model shown in Figure 20. The table on the top right of the figure shows the values for the *source restriction* and *target determinator* annotations for two of the transitions.

Behavioral equivalence of the Statechart model and the EHA model with respect to reachability

A "state configuration" in a Statechart is a valid set of states that the system can be active in. If a state is part of an active configuration, then all its parents are also

part of the active configuration. A transition in the Statechart can take the system from one state configuration to another state configuration, where the source and target states of the transition are subsets of the initial and final state configurations. A state configuration S_f in the Statechart is said to be "reachable" from a state configuration S_i if there exists a series of valid transitions that can take the system from S_i to S_f.

Similarly, a state configuration in an EHA model is a set of Basic States. If a Basic State is part of an active configuration, and is part of a non-toplevel Sequential Automaton, then the Basic State that is refined into this Sequential Automaton is also a part of the active configuration. For instance, B', F', I' is a valid active configuration in Figure 22. A transition in the EHA can take the system from one state configuration to another state configuration, where the union of the source of the transition and its source restriction are a subset of the initial state configuration, and the union of the target of the transition and its target determinator are a subset of the final state. A state configuration S_f in the EHA is said to be "reachable" from a state configuration S_i if there exists a series of valid transitions that can take the system from S_i to S_f.

An EHA model truly represents the reachability behavior of a Statechart model, if every reachable state configuration in the Statechart has an equivalent reachable state configuration in the EHA and vice versa.

For every state s in the Statechart, we have a unique Basic State s' in the EHA. We can specify an equivalence relation R, such that $(s, s') \in R$ and say that s' is equivalent to s. A state configuration S in the Statechart is equivalent to a state configuration S' in the EHA if for all $s \in S$ there is an equivalent $s' \in S'$, and for all $s' \in S'$, there is an equivalent $s \in S$. Furthermore, for every transition t in the

Statechart, we have a unique transition t' in the EHA. We can specify an equivalence relation R_t, such that (t, t') $\in R_t$ and say that t' is equivalent to t.

Given the relations R and R_t, we can check if there is a bisimulation relation between the two models using the following definition. Given a state configuration S_A in the Statechart model, and its equivalent state configuration S_B in the EHA model, the equivalence is a bisimulation if for each transition t from S_A to a state configuration S_A' in the Statechart, there exists an equivalent transition t' in the EHA from S_B to a state configuration S_B', and S_B' is equivalent to S_A' (and vice versa)

If this relation is a bisimulation, then verifying the EHA model for reachability will be equivalent to verifying the Statechart model for reachability. If the check fails, it means that there was an error in the transformation.

Checking for bisimilarity by using cross-links to trace equivalence

GReAT allows us to link input model elements to target model elements using special associations that belong to a composite meta-model, and we call them *cross-links*. These cross-links are maintained throughout the transformation, and used to trace the equivalence relations R and R_t.

When a transformation creates the Basic States and the transitions in the target EHA model, it is known to which states and transitions they correspond to in the Statechart model. What is not certain is whether all states in the Statechart are transformed correctly, all composite states are refined correctly, all transitions are transformed correctly, and all transitions connect the correct sets of states. When a rule matches a state or a transition in the Statechart and creates the equivalent Basic State or transition in the EHA, a cross-link association called "equivalentTo" is created between the Statechart object and its corresponding EHA object. When

the transformation completes, the relations R and R_t can be traced using these associations.

Rather than checking for all possible state configurations in the Statechart, it would be more efficient to consider every transition in the Statechart and its minimal required source configuration. Any superset of this state configuration will be a valid starting configuration, and will not have to be investigated further. For every transition t in the Statechart model, and its equivalent transition t' in the EHA model, if their start state configurations S_A and S_B are equivalent, and also their end state configurations S_A' and S_B' are equivalent, then there exists a bisimulation for this particular instance, according to our definition.

The implementation follows straightforwardly from the discussion. At the end of the transformation, we have access to the source model graph, the output model graph, and also the cross-links between the two. We collect the set of all the transitions from the source graph. For each transition in this set, we find the equivalent transition in the EHA by following the "equivalentTo" cross-link. Now we can compute the minimal source state configuration S_A for the transition in the Statechart model, and the source state configuration S_B for the EHA model. We check the equivalence of S_A and S_B by taking every state s in S_A, finding its equivalent state s' form the EHA, and checking if s' is in S_B, and vice versa. The target states are also checked similarly. If this check succeeds for all transitions in the Statechart, and there are no more transitions in the EHA, then the two systems can be said to be bisimilar with respect to checking reachability. In other words, we can conclude reachability in the Statechart model by verifying it in the EHA model. If this check fails, then there may be errors in the transformation, and the generated EHA model does not truly represent the input Statechart model.

The final step is checking the reachability in the EHA model. [15] provide ways to generate a Promela model from an EHA model, which can be checked using the SPIN model checker. To check the reachability of a certain state configuration, a claim can be attached to the SPIN model that verifies whether that configuration is reachable in the model. Alternately, a claim can be made in SPIN that says that the state is not reachable. If it is indeed reachable, the SPIN verifier refutes this claim and presents a counter-example, as a trace that leads to this state configuration. This represents a valid series of transitions in the EHA that leads to the specified state configuration. As a corollary, we may use the cross-links created during the transformation, to reproduce this trace in the Statechart model. In this way, reachability in the Statechart model can be verified by verifying it in the EHA model.

It should be noted that the technique described above is not an attempt to prove the correctness of the graph transformation rules in general. This is a method to verify if a particular instance of a transformation is valid, and must be executed for each transformation individually. We also do not try to prove the general semantic equivalence of models. We identify the equivalence relations with respect to a specific property and test if there is a bisimulation. The complexity of the transformation is not increased significantly by this method. As the cross-links are created every time the objects of the output model are created, and as we directly trace these cross-links during checking, the complexity of the check is proportional to the size of the model, and not the state space of the model. In other words, we can perform this check without actually having to execute the models.

Related work

We now discuss some related work in the area of automatic verification using model checking, graph transformations and other types of proofs.

Verifying properties by converting models into an intermediate format

[61] [14] convert Statechart models into EHA models. [15] create Promela models from the EHA models, which can be verified using the SPIN model checker. Our approach will be useful in these instances, to provide a certificate that the intermediate formats truly preserve the property we wish to verify using them. An interesting research problem is whether our approach can be used to check whether the generated Promela model (which is code in plain text) truly represents the EHA model it was generated from.

Operational semantics using graph transformations

[65] [49] [48] are some works on using graph transformation rules to specify the dynamic behavior of systems. [48] presents a meta-level analysis technique where the semantics of a modeling language are defined using graph transformation rules. A transition system is generated for each instance model, which can be verified using a model checker. [75] verifies if a transformation preserves certain dynamic consistency properties by model checking the source and target models for properties p and q, where property p in the source language is transformed into property q in the target language. This transformation requires validation by a human expert. Our method does not check whether the models themselves satisfy a property, but automatically does check whether the models are equivalent with respect to that property.

Certifiable program generation

[52] considers the problem of verification of generated code by focusing on each individual generated program, instead of verifying the program generator itself. The generator is extended such that it produces all logical annotations that are required for formal safety proofs in a Hoare-style framework. These proofs certify that the

program does not violate certain conditions during its execution. While the proofs in this case are not related to semantic correctness, the idea of providing an instance level certificate of correctness instead of proving the correctness of the generator has been a great motivation for our ideas.

Summary

We have described a method for checking if a certain execution of a transformation produced an output model that preserved the semantics of the input model. This check is important when the output model is used for verification and analysis, as errors in the transformation may result in an output model that does not truly represent the input model. We are studying how such an equivalence can be established when the target model abstracts away a lot of detail in the source model. Our method does not attempt to prove the correctness of the transformation itself, but checks whether a particular execution produced a correct result. This check does not adversely affect the complexity of the transformation.

CHAPTER VI

USING SEMANTIC ANCHORING TO VERIFY BEHAVIOR PRESERVATION IN GRAPH TRANSFORMATIONS

Overview of this Paper

This paper extends the ideas described in the previous paper, to cover a wider range of transformations. Bisimulation is a strict form of behavioral equivalence that may not be suitable for all transformation cases, especially those involving abstraction or some loss of information. For instance, when transforming models between different variants of a formalism, some information may be lost due to features not supported in one formalism. We would still like to verify if some behavioral properties were preserved, by relaxing some of the restrictions of strict bisimulation.

This paper proposes the use of *weak bisimulation* as an approximation to verify behavior preservation under some relaxed conditions. It also uses the idea of *semantic anchoring* to obtain the behavior models of higher level domain specific languages, and verify weak bisimulation between the behavior models. As a case study, two hypothetical variants of the Statecharts formalism are described, along with the transformation between them, accounting for the differences in the formalism. The extension to the transformation framework for verifying weak bisimulation is described.

The paper describes the hypothetical Statecharts variants in the backgrounds section. The transformation between the languages, and their semantic anchoring is described. The paper describes the conditions for weak bisimulation and how to account for loss of information.

Introduction

The preservation of the behavior of a model is crucial in many kinds of graph transformations. Consider a scenario where different users are exchanging information modeled through Statecharts. Since a formalism like Statecharts has many implementation variants [13], it is often the case that the users use different variants. It is then necessary to transform the models from one variant to the other, such as from iLogix Statecharts to MATLAB Stateflow models. Such transformations could be accomplished through Graph Transformations. One related example is [63], which uses graph transformations to translate Simulink/Stateflow models into Hybrid Automata. In all these cases, it is essential that the transformed model preserves the behavior of the source model.

Defining the behavior formally is a first step to verifying its preservation. Semantic Anchoring [46] is a technique to specify the operational semantics of DSMLs using a semantic framework and anchoring rules. Bisimulation has been suggested as a method to check if a transformation preserves certain behavioral properties [45]. In this paper, we propose a method to verify behavior equivalence by using bisimulation in conjunction with semantic anchoring. As the practical Statecharts implementations vary in very subtle features that are not very suitable for a case study, we have devised two hypothetical variants with certain key differences that can better illustrate the complexities of the transformation. We will first specify their operational semantics by semantic anchoring, using Abstract State Machines as a common semantic framework. This will allow us to generate a behavior model for any instance model. We will then show how we can use bisimulation to check if the behavior models are equivalent. If the behavior models are equivalent, we can conclude that that particular instance of the transformation preserved the behavior correctly.

Though the source and target domains considered here are very similar, this approach can be applied to other types of transformations as long as the semantics of the source and the target languages can be represented using a common semantic framework such as Abstract State Machines.

Background

In this section, we review some background information.

Statecharts

Statecharts [10] were first proposed by Harel to model the reactive behavior of systems. Statecharts were presented as an extension to conventional state machines, allowing hierarchy, concurrency and broadcast communication. Statecharts are constructed from *states* and *transitions*. States may be *simple* (basic states), *composite* (OR states) or *concurrent* (AND states). If a system is in a composite state, it is also in exactly one of its direct sub-states. If a system is in a concurrent state, it is also in all of its direct sub-states. A *state configuration* is a maximal set of states that the system can be active in simultaneously. Transitions take the system from one state configuration to another. Events are the basic units of broadcast communication. Transitions may be annotated with triggers, guards and actions. Triggers are the events required to activate the transition, actions are events that are broadcasted as a result of taking the transition, and guards are boolean conditions that can enable or disable the transition.

Since the introduction of the Statecharts formalism, several variants have been proposed to overcome specific difficulties. A number of them have been surveyed

in [13]. One feature that the variants may differ on is whether inter-level transitions (which are transitions that cut across levels of hierarchy) are allowed. Another difference may be whether instantaneous states are permitted.

Semantic Anchoring

Domain Specific Modeling Languages (DSMLs) capture concepts, relationships and integrity constraints that will allow users to specify their systems declaratively. The meta-modeling step of designing a DSML specifies the syntax and static semantics of the DSML. Semantic anchoring [46] concentrates on the specification of the dynamic semantics of the DSML. Semantic anchoring relies on the observation that a broad category of component behaviors can be represented by a small set of basic behavioral abstractions such as Finite State Machines, Timed Automata etc. We assume that the behavior of these abstractions are well understood and precisely defined. These basic abstractions are called *semantic units*. The behavioral semantics of the DSML is specified as a transformation from the DSML to the selected semantic unit. This last step is called the semantic anchoring. For instance, FSMs can be chosen as a semantic unit to represent the behavior of Statecharts.

Figure 23 [46] shows the tool architecture for specifying operational semantics to DSMLs through semantic anchoring. The GME [57] MIC toolset is used to define the static semantics and integrity constraints of the DSML, and to design the domain models. The semantic anchoring transformation is specified using GReAT [57].

Bisimulation

Bisimulation [56] is defined for Labeled Transitions Systems (LTS). Given an LTS (S, Λ, $\rightarrow$), a relation R over S is a *bisimulation* if:

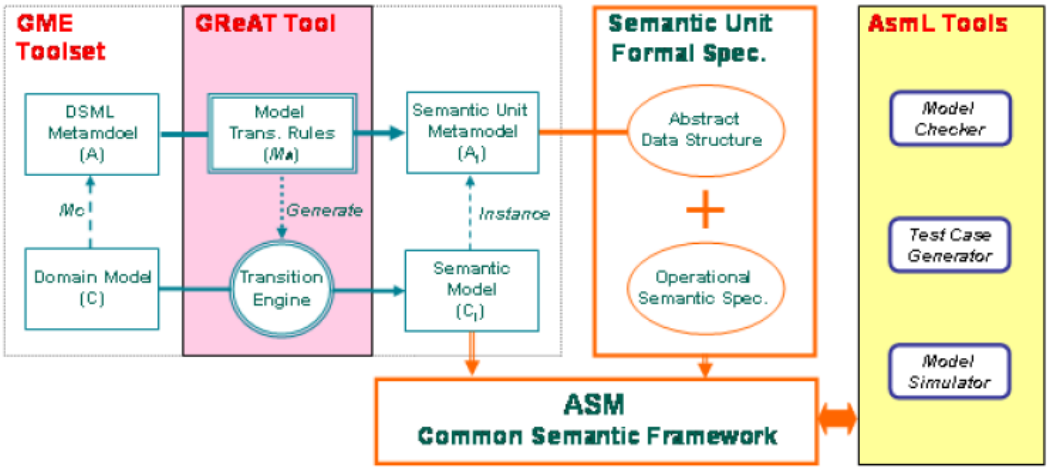


Figure 23: Tool Architecture for Semantic Anchoring

$(p, q) \in R$ and $p \xrightarrow{\alpha} p'$ implies that there exists a $q' \in$ S such that $q \xrightarrow{\alpha} q'$ and $(p', q') \in R$,

and conversely,

$q \xrightarrow{\alpha} q'$ implies that there exists a $p' \in$ S such that $p \xrightarrow{\alpha} p'$ and $(p', q') \in R$.

Bisimilarity is the union of all bisimulations. Bisimilarity is generally accepted as the finest form of behavioral equivalence.

We can also consider different "flavors" of bisimulation by varying the definition of transitions and labels. If we replace the transition $\rightarrow$ by a *weak transition* $\Rightarrow$, we get the definition of a *weak bisimulation* [64]. Suitably defining what constitutes observable states and transitions, we can check if two transition systems have a weak bisimilarity. This will allow us to conclude whether the transition systems are behaviorally equivalent for all practical purposes, though there may not be a fine-grain equivalence. For instance, we may choose to ignore internal representations of data or intermediate states when comparing the transition systems for equivalence.

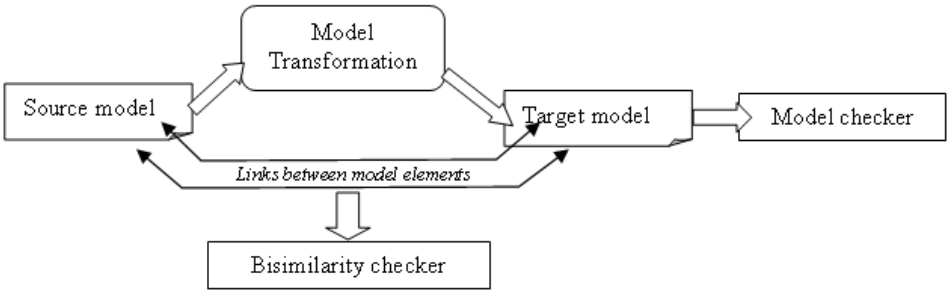


Figure 24: Architecture for verifying reachability preservation in a transformation

Verifying Instances of Graph Transformations

In an earlier work [45], we proposed a method for using bisimulation to verify each instance of a graph transformation. We believe that the strategy to verify each instance is less complex and more practical, as opposed to devising a correctness proof for the transformation itself, which may be intractable.

In this approach, we use bisimilarity to check if a reachability property is preserved by a certain instance of a transformation. Figure 24 shows the basic architecture used for the verification. The model transformation creates temporary associations between the source and target elements, which are used to trace the equivalence relation R. These links are then used by a bisimilarity checker, to check if this particular instance of input and output models preserve the same reachability behavior. If this check succeeds, the results of a model checker on the output model instance will be valid for the input model instance as well.

Verifying Behavior Preservation

Our approach to verifying behavior preservation relies on establishing a weak bisimilarity between the transition systems representing the behaviors of the source and target models. Figure 25 shows an overview of our approach. We would like to

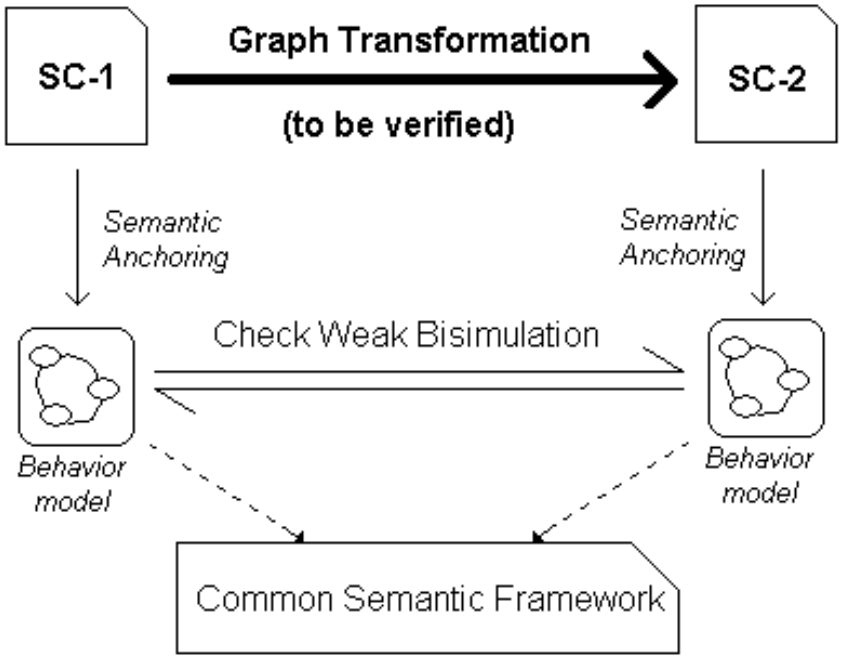
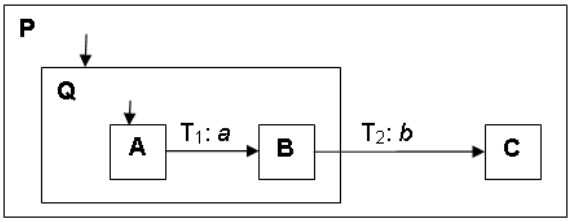


Figure 25: Framework for verifying behavior preservation

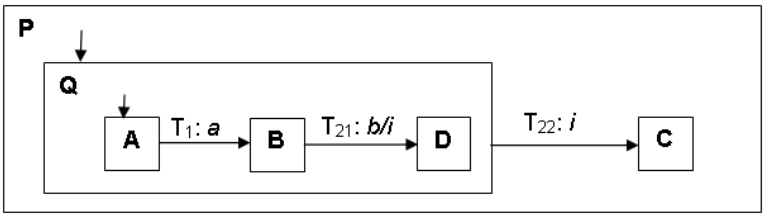
verify the graph transformation instance that transforms the *Variant 1* Statechart model *SC-1* into a *Variant 2* Statechart model *SC-2*. We generate the behavior models of both the source and target instances using semantic anchoring. We will use non-hierarchical FSMs as a semantic unit to represent the behavior of the source and target DSMLs. The semantic anchoring process will result in flat FSM representations of the source and target instances. We will then verify if there is a weak bisimulation based on some criteria which will be explained in the following sub-sections.

The Source and Target Languages

The popular Statecharts variants used currently vary in a number of subtle issues. We conjecture that a case study will be most useful if we consider two hypothetical variants, lets call them *Variant 1* and *Variant 2*, which differ in a small set of features



(a) A sample *Variant 1* Statechart model



(b) A sample *Variant 2* Statechart model

Figure 26: Sample Models

that can be defined clearly. For this case study, we will only consider the synchronous time model. We will now list these features and their semantics.

Compositional Semantics

A language is said to have compositional semantics if the semantics of a compound object is completely defined by the semantics of its subcomponents. In other words, we do not have to look at the internal syntactical structure of the subcomponents. Such semantics are useful in verification. Compositional semantics are violated by allowing inter-level transitions and state references. Inter-level transitions are transitions that cut across levels of hierarchy. State references are a mechanism to allow the execution of a transition based on whether a certain parallel component is active, expressed as trigger conditions such as *in(State)*. In our case study, we will allow *Variant 1* to have inter-level transitions and state references, and not allow them in *Variant 2*.

Figure VI.26(a) shows a *Variant 1* Statechart, where transition T_2 is inter-level. Figure VI.26(b) shows a *Variant 2* Statechart, which tries to simulate the semantics of the *Variant 1* Statechart, by using a self-termination state to replace the inter-level transition. What happens if the transition T_2 in Figure VI.26(a) has an action E? In *Variant 1*, E will be active when the system enters state C, but in *Variant 2*, E will not be available when the system enters state C (since events are available only in the step following the one in which they occur, and not in subsequent steps[1]). The semantics will be better reproduced if state D in Figure VI.26(b) is an *instantaneous state*. Instantaneous states will be discussed in the next subsection.

We must note that it may not be possible to represent all *Variant 1* Statecharts as *Variant 2* Statecharts. Our objective is not to find a *Variant 2* representation of any *Variant 1* Statechart. Rather, it is to verify whether a *Variant 2* Statechart generated by a graph transformation can be accepted as behaviorally equivalent to the *Variant 1* Statechart that was the input to the transformation.

Instantaneous States

An instantaneous state is a state that can be simultaneously entered and exited in a single step. Instantaneous states are not allowed in most common Statecharts variants. We will allow instantaneous states in *Variant 2* Statecharts, with the semantics that a step is not complete until there are no instantaneous states in the final state configuration of that step. The sequence of transitions leading to the final state configuration constitute a macro step, which is considered to be executed in zero time[2]. Events available at the start of the macro step are available throughout the

[1] The durability of events is itself an issue that Statecharts variants may differ on, and is explained in [13]

[2] For the purposes of this case study, we will not go into the issues of infinite sequences of transition executions.

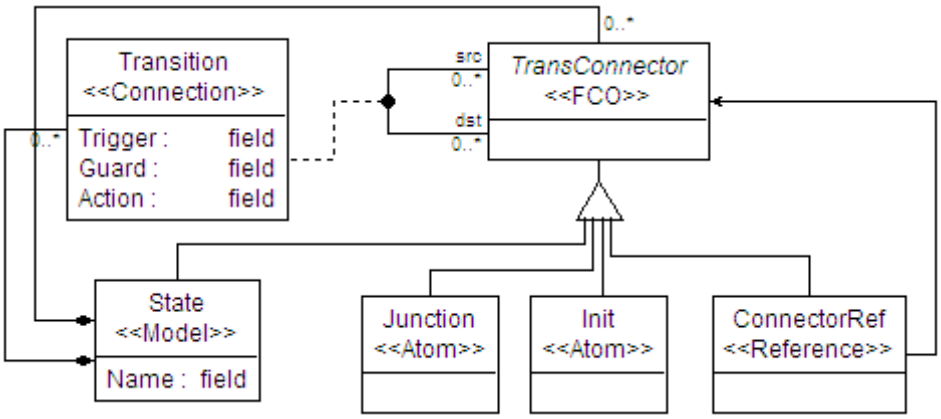
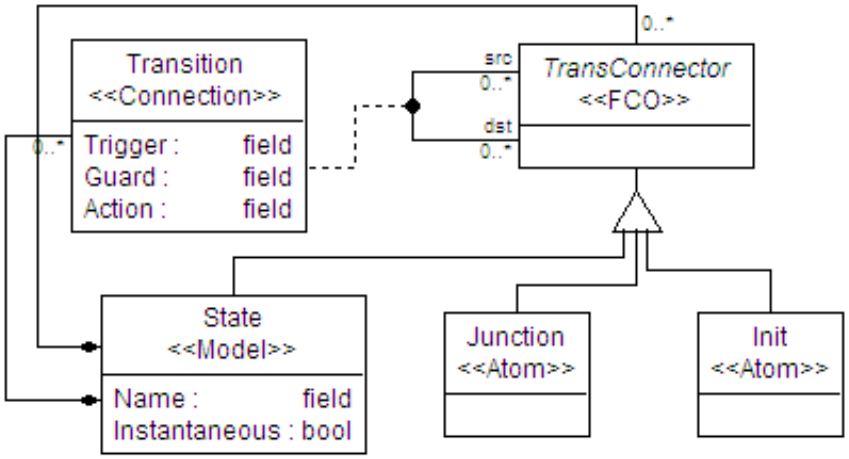
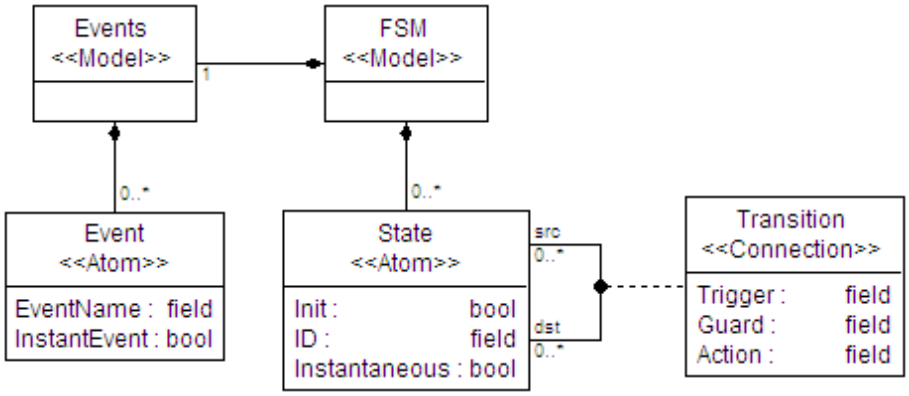


Figure 27: GME Meta-model for Statechart *Variant 1*

macro step, and actions on any of the transitions in the macro step will be available in the step following the macro step. For providing finer semantics, we introduce the notion of instantaneous actions to *Variant 2* Statecharts, which will be available in the same step. In Figure VI.26(b), D will be an instantaneous state, and i will be an instantaneous action. The sequence T_{21}, T_{22} will form a macro step, with B as the starting state and C as the ending state. To an external observer, the instantaneous state D and instantaneous action i will not be visible, and the macro step will appear to be a transition whose triggers and actions will be the aggregate of the triggers and actions of the sequence of transitions. In other words, the macro step will be identical to a transition whose trigger is the conjunction of the triggers for each micro step, and all the trigger events must be enabled at the start of the macro step. The conjunction of all the non-instantaneous actions will be available for the step following the macro step. The effect of the macro step (T_{21}, T_{22}) in Figure VI.26(b) will be identical to transition T_2 in Figure VI.26(a).



(a) GME Meta-model for Statechart *Variant 2*



(b) GME Meta-model for FSM semantic unit

Figure 28: GME Meta-models

State References

Some Statecharts variants allow triggers to be specified by referencing the activity of other, parallel states. For instance, the condition *in(S)* is true when state S is active, and entering or exiting S will result in events *en(S)* and *ex(S)* respectively. We will allow state references in *Variant 1*, and not in *Variant 2*. The graph transformation that will generate the *Variant 2* Statechart will replace the events for *en(S)* and *ex(S)* when necessary, but will not handle the condition *in(S)*. The replaced events will be considered equivalent when verifying behavior preservation. *Variant 1* Statecharts which use the condition *in(S)* cannot be transformed by the graph transformation which we will consider in the case study (*in(S)* can be simulated by a system of self loops, but we chose not to implement it in the case study, to avoid overly complicating the transformation).

Figures 27 and VI.28(a) show the GME meta-models for the two Statechart variants for the case study. The GME modeling environment allows connections to only reside under a single parent. Thus, inter-level transitions must be represented by using references to states that are under another parent. The absence of a *ConnectionRef* prohibits inter-level transitions in Variant 2. States can be marked as instantaneous in Variant 2. The absence of state references in Variant 2 is enforced by the operational semantics specification by semantic anchoring for this variant.

Operational Semantics Using Semantic Anchoring

We will use a flat FSM as the semantic unit to represent the behavior of the Statecharts variants. The meta-model of the semantic unit is shown in Figure VI.28(b). The FSM model allows instantaneous states and events, and can model the behavior of both the Statecharts variants discussed above. The semantics of this FSM semantic unit is defined in AsmL [47], the Abstract State Machine Language developed by

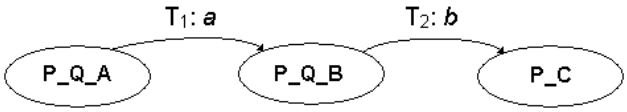
Microsoft Research. This is done by specifying an Abstract Data Model in AsmL, corresponding to the constructs of the FSM semantic unit. For instance, *Events* are modeled as below:

```
interface Event

structure ModelEvent implements Event

structure LocalEvent implements Event

structure InstantEvent implements Event
```

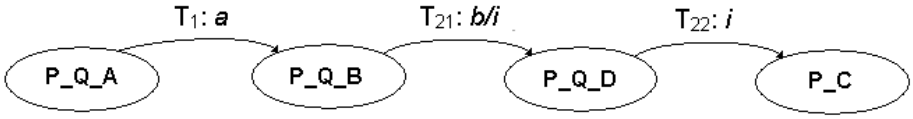
ModelEvent models events input to the model, *LocalEvent* models the events generated in a step, and *InstantEvents* models events generated by instantaneous actions, which will be available in the same step.

The FSM itself, the states and the transitions are modeled using AsmL *class* constructs:

```
class FSM

id as String

var outputEvents as Seq of ModelEvent

var localEvents as Set of LocalEvent

...

class State

id as String

var active as Boolean = false

var instantaneous as Boolean

var outTransitions as Set of Transition

...
```



(a) FSM semantic model 1



(b) FSM semantic model 2

Figure 29: FSM Semantic Models

```
class Transition

...
```

The step semantics are modeled using operational rules. The AsmL model of the instances can be obtained by instantiating the states and transitions based on the model. The details of creating the AsmL abstract model can be found in [46].

The specification of the behavior of the Statecharts variants will be expressed via semantic anchoring, as a transformation from the Statechart model to the FSM semantic unit. Figure VI.29(a) shows the FSM behavior model of the Statechart in Figure VI.26(a). The first step in the transformation will to convert the hierarchical Statechart model into a non-hierarchical FSM model, by enumerating all possible state configurations. Each legal state configuration of the Statechart will be represented by a unique state in the FSM. For instance, the state P_Q_A in Figure VI.29(a) represents the state configuration consisting of the states P, Q and A in Figure VI.26(a). The next step is to transform the transitions. The transitions are extracted from the Statecharts model, and the source and target state configurations

are determined. The corresponding unique states in the FSM are located, and a transition is constructed between them. The trigger, action and guards are then updated in the FSM model.

This transformation is a behavior specification, and itself is not verified. We may use the technique described in [45] to verify the behavior model by bisimulation if necessary, but the behavior specification may be considered straightforward enough to not require further verification.

Setting up the Transformation

We will now describe the graph transformation for our case study. This transformation will take a *Variant 1* Statechart and convert it into a *Variant 2* Statechart.

The main tasks of this transformation will be to convert inter-level transitions in *Variant 1* into normal transitions in *Variant 2* and to replace any state referencing actions by regular actions. Compositional semantics can be developed by replacing inter-level transitions with concepts called self-start and self-termination. In Figure VI.26(b), the state D can be thought of as a self-termination state, with the help of which the sequence of transitions T_{21}, T_{22} replace an interlevel transition. Similarly, self-start states can be used in the case of interlevel transitions entering a state and terminating in one of its substates.

Every state in the input Statechart is first copied on to the output. Figure 30 shows a simple graph transformation rule in GReAT, where a new state is created in the *Variant 2* Statechart (indicated by the tick mark) for a child state found in the *Variant 1* Statechart. The *Attribute Mapping* block is a special construct in GReAT that allows us to perform additional functions, used in this case to set up the label of the state. We also track the equivalent states by creating a direct link between them. The transitions are extracted one by one, and if the source and destination

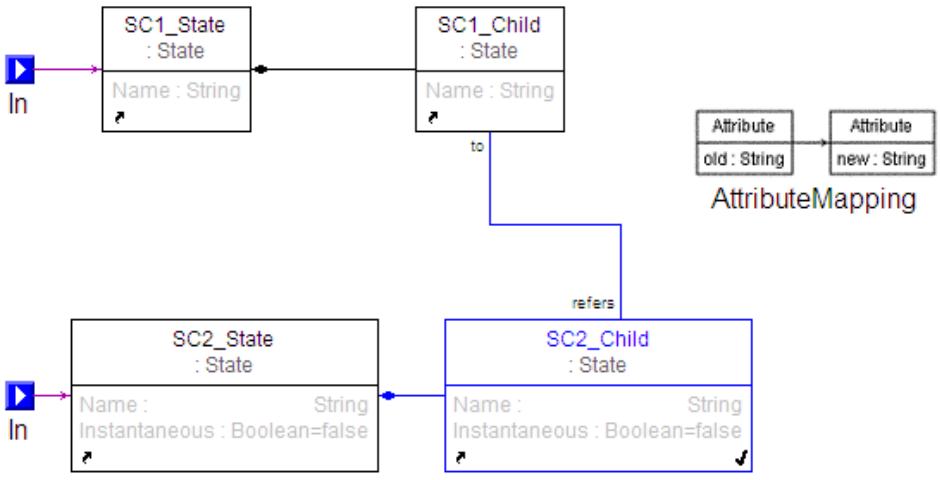


Figure 30: Sample GReAT rule

states are contained in the same parent, they are constructed in the output between the corresponding source and destination states. If the transition is inter-level, then a self-start or self-termination state is added to the deeper of the two states, and the parent is marked as the source or target. The process is repeated until the source and target states are under the same parent. An automatic system of naming and numbering the intermediate instantaneous states and actions will ensure that each inter-level transition is reproduced uniquely.

If there are state references in the input model, they will be copied in the output model as normal events, using a special naming convention for identification. All such events will then be added as actions to all transitions into or out of the respective states being referenced. For instance, if a trigger *en(S)* is used in a *Variant 1* Statechart, it will be replaced by *en_S* in the output, and the action *en_S* will be added to all the transitions into state S in the output. The occurrence of *in(S)* in the input model will be flagged as an error.

Our objective here is to verify if a certain instance of the transformation produced a *Variant 2* Statechart that preserved the behavior of the *Variant 1* Statechart given as input. To verify this, we get the semantic model of the input instance and the semantic model of the output instance, and check if they are behaviorally equivalent. This is done by establishing a weak bisimulation relation between the two behavior models.

Weak Bisimulation

According to the semantics we have described, we consider the two Statechart models in figures VI.26(a) and VI.26(b) to be behaviorally equivalent. But it is obvious that the two behavior models in Figures VI.29(a) and VI.29(b) are not bisimilar. We thus turn to a practically useful notion of bisimilarity that will help us in this scenario, the notion of *weak bisimilarity*.

In Figure VI.29(b), the state P_Q_D and the action i are instantaneous. To an external observer, the sequence of transitions T_{21}, T_{22} will appear identical to the transition T_2 in Figure VI.29(a). To the observer, the two systems are behaviorally equivalent, even though they may vary internally. Weak bisimulation allows us to weaken the notion of what constitutes a transition, allowing us to set the granularity at which we accept two systems as behaviorally equivalent. In our scenario, we argue that it is acceptable to consider only non-instantaneous states as states of the transition system, and a transition as one that goes from one non-instantaneous state to another, passing through any number of instantaneous states. We consider such transitions as a single *weak* transition, whose trigger and action are the aggregate of the series of transitions, disregarding all instantaneous states and actions.

We now rephrase our earlier definition of bisimulation, to define a weak bisimulation for FSM models. We define an equivalence relation R between the non-instantaneous states of two FSM models, for the current study, by simply stating that two states p and q are in R if they have the same label. Given this equivalence relation, we define a somewhat novel definition of weak bisimulation that is useful for our purpose:

$$\forall\ (p,\ q) \in R \text{ and } \forall \alpha\text{: } p \overset{\alpha}{\Rightarrow} p',\ \exists\ q' \text{ such that } q \overset{\alpha}{\Rightarrow} q' \text{ and } (p',\ q') \in R,$$

and conversely,

$$\forall \alpha\text{: } q \overset{\alpha}{\Rightarrow} q',\ \exists\ p' \text{ such that } p \overset{\alpha}{\Rightarrow} p' \text{ and } (p',\ q') \in R.$$

where p, q, p', q' are all non-instantaneous states, the transition $\Rightarrow$ is from one non-instantaneous state to another, and α is the aggregate of the events for the transition, disregarding instantaneous states and actions (The label α constitutes both the cause of a transition and its effect. In our implementation, we represent α as a comma separated list of the events that are the triggers and the actions of the transition. In the case of a weak transition, this list will include all the non-instantaneous events that are the triggers and the actions of the sequence of transitions which constitute the weak transition). According to this definition, the FSM models in Figurers VI.29(a) and VI.29(b) are weakly bisimilar.

Checking for Weak Bisimulation

To verify weak bisimulation, we will first reduce the FSM into non-instantaneous states and weak transitions. This is done by tracing through the FSM model, and aggregating any sequence of transitions through instantaneous states. In this case, only *Variant 2* Statecharts will result in FSM models with instantaneous states, and the reduction step is not required for FSM models of *Variant 1* Statecharts.

The next step is to establish the equivalence relation R. For this study, we will consider two states p and q to be in R if they have the same label. Our transformation is set up in such a way that newly created states in the *Variant 2* Statechart model are labeled depending upon the *Variant 1* state they were created from. The FSM states are similarly labeled depending on the labels of the states in the Statechart state-configuration that they represent. We will consider that two states are equivalent if they represent the same configuration in the two Statecharts.

After these two steps, for each of the FSMs, we list all the states in a table, followed by all the outgoing transitions for each of these states. We then step through the states one by one, verifying that the weak bisimulation holds according to the definition above. After stepping through the list of states for both the FSMs, we can conclude whether the generated *Variant 2* Statechart is behaviorally equivalent to the input *Variant 1* Statechart.

Related Work

In this section, we review some related work.

Graph Transformation Based Operational Semantics

[65] and [48] present approaches to specify the operational semantics of DSMLs using graph transformations. [48] presents a meta-level analysis technique where the semantics of a modeling language are defined using graph transformation rules. A transition system is generated for each instance model, which can be verified using a model checker. This is an alternative to the semantic anchoring approach. We find the semantic anchoring approach easier to use, as we can choose a semantic unit such as non-hierarchical FSMs, and use that to verify weak bisimulation. But as long as the semantics of the input and output languages can be specified in terms of a

common representation, and we define the equivalence relation and rules for weak bisimulation appropriately, behavior preservation can be verified.

Certifiable Program Generation

[52] considers the problem of verification of generated code by focusing on each individual generated program, instead of verifying the program generator itself. The generator is extended such that it produces all logical annotations that are required for formal safety proofs in a Hoare-style framework. These proofs certify that the program does not violate certain conditions during its execution. While the proofs in this case are not related to semantic correctness, the idea of providing an instance level certificate of correctness instead of proving the correctness of the generator has been a great motivation for our ideas.

Conclusions and Future Work

Semantic anchoring is a useful method for formally specifying the dynamic semantics of DSMLs. We have shown here that it allows us to use bisimulation to verify behavior equivalence across a transformation. This technique is especially useful in cases where it is hard to compare the source and target languages directly. As long as their behavior can be specified using a common semantic framework, we can verify behavioral equivalence by using bisimulation. We have also shown that weak bisimulation is a practical way to determine acceptable behavioral equivalence. As in [45], we believe that it is more practical to verify whether an instance of a transformation succeeded in preserving behavior, instead of providing a proof of correctness for the transformation itself.

Further research in determining semantic units that can represent a wide variety of DSMLs will allow us to use this technique for a larger range of transformations.

We may also consider representing a specific behavioral property using semantic anchoring, as opposed to the complete operational semantics of the language. This will allow us to check the preservation of a specific behavior in languages that are otherwise very different. The checking of weak bisimulation may also be refined to be more efficient.

CHAPTER VII

VERIFYING MODEL TRANSFORMATIONS BY STRUCTURAL CORRESPONDENCE

Overview of this Paper

The previous papers suggested approaches to verify if an execution of a transformation resulted in models that preserved certain behavioral properties. There are also a large class of transformations that perform structural modifications on the instance models that may not have a clearly defined behavior. In such cases, correctness is determined by whether the desired structural transformations were effected (as opposed to preservation of a previously defined property). This paper addresses the instance based verification of such transformations.

This paper proposes a specification of *structural correspondence* between the source and target models, which is a set of rules that captures the correctness of the desired structural transformation. Models are essentially typed, attributed graphs, that conform to a type graph (the meta-model). A correct output graph must have a specific structure depending on the corresponding input graph. We can describe this correctness by relating specific nodes and edges of the source and target type graphs. If the transformation framework is extended to map the correspondences between source and target instance nodes, the correctness conditions can be verified on the instance models.

This paper uses a transformation from UML Activity Diagrams to Communicating Sequential Process (CSP) specifications as a case study to demonstrate the verification of transformations by structural correspondence. The source and target languages, and the transformation are explained in the backgrounds section. Specific portions

of the transformation are then described separately, along with how a structural correspondence can be established between them.

Introduction

Model transformations that translate a source model into an output model are often expressed in the form of rewriting rules, and can be classified according to a number of categories [36]. However, the correctness of a model transformation depends on several factors, such as whether the transformation terminates, whether the output model complies with the syntactical rules of the output language, and others. One question crucial to the correctness of a transformation is whether it achieved the intended result of mapping the semantics of the input model into that of the output model. For instance, a transformation from a Statechart model to a non-hierarchical FSM model can be said to be correct if the output model truly reproduces the behavior of the original Statechart model.

Models can also be seen as attributed and typed graph structures that conform to an abstract syntax. Model transformations take an input graph and produce a modified output graph. In a majority of these cases, the transformation matches certain graph structures in the input graph and creates certain structures in the output graph. In such cases, correctness may be defined as whether the expected graph structures were produced in the output corresponding to the relevant structures in the input graph. If we could specify the requirements of such correspondences and trace the correspondences easily over instance models, a simple model checking process at the end of a transformation can be used to verify if those instances were correctly transformed. In this paper, we explore a technique to specify *structural correspondence* rules, which can be used to decide if the transformation resulted in an output model with the expected structure. This will be specified along with the

transformation, and evaluated for each execution of the transformation, to check whether the output model of that execution satisfies the correspondence rules.

Background

In this section, we review some background information.

GReAT

GReAT [58] is a language framework for specifying and executing model transformations using graph transformation. It is a meta-model based transformation tool implemented within the framework of GME [58]. One of the key features of GReAT is the ability to define cross-language elements by composing the source and target meta-models, and introducing new vertex and edge types that can be temporarily used during the transformation. Such cross-meta-model associations are called *cross-links*. Note that a similar idea is present in Triple Graph Grammars [66]. This ability of GReAT allows us to track relationships between elements of the source and target models during the course of the transformation, as the target model is being constructed from the source model. This feature plays a crucial role in our technique to provide assurances about the correctness of a model transformation.

Instance Based Verification of Model Transformations

Verifying the correctness of model transformations in general is as difficult as verifying compilers for high-level languages. But for practical purposes, a transformation may be said to have 'executed correctly' if a certain instance of its execution produced an output model that preserved certain properties of interest. We call that instance 'certified correct'. This idea is similar to the work of Denney and Fischer in

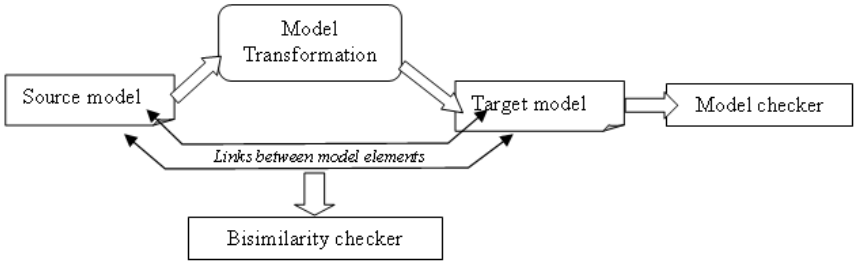


Figure 31: Architecture for Verifying Reachability Preservation in a Transformation

[52], where a program generator is extended to produce logical annotations necessary for formal verification of certain safety properties. An automated theorem prover uses these annotations to find proofs for the safety properties for the generated code. Note that this does not prove the safety of the code generator, but only of a particular instance of generated code.

In our previous effort [45], we have shown that it is both practical and prudent to verify the correctness of every execution of a model transformation, as opposed to finding a correctness proof for the transformation specification. This makes the verification tractable, and can also find errors introduced during the implementation of a transformation that may have been specified correctly.

This technique was applied to the specific case of preservation of reachability related properties across a model transformation. Reachability is a fairly well-understood property, and can be verified easily for a labeled transition system (LTS), for instance by model checking [16]. If two labeled transition systems are *bisimilar*, then they will have the same reachability behavior. In our approach, we treated the source and target models as labeled transition systems, and verified the transformation by checking if the source and target instances were bisimilar.

Given an LTS (S, Λ, $\rightarrow$), a relation $R \subseteq S \times S$ is a *bisimulation* [56] if:

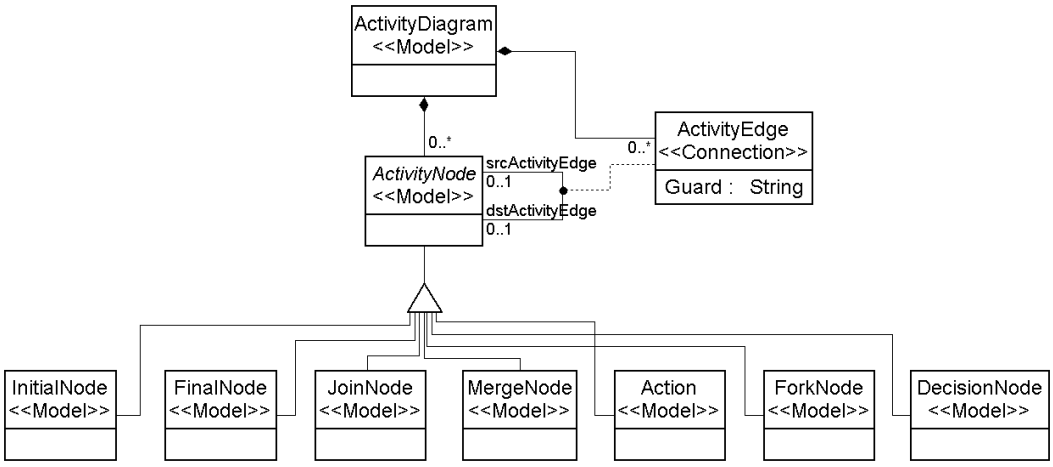


Figure 32: Meta-model for UML Activity Diagrams

$$(p,\, q) \in R \text{ and } p \xrightarrow{\alpha} p' \text{ implies that there exists a } q' \in \mathrm{S},$$
$$\text{such that } q \xrightarrow{\alpha} q' \text{ and } (p',\, q') \in R,$$

and conversely,

$$q \xrightarrow{\alpha} q' \text{ implies that there exists a } p' \in \mathrm{S},$$
$$\text{such that } p \xrightarrow{\alpha} p' \text{ and } (p',\, q') \in R.$$

We used cross-links to relate source and target elements during the construction of the output model. These relations were then passed to a bisimilarity checker, which determined whether the source and target instances were bisimilar. If the instances were determined to be bisimilar, we could conclude that the execution of the transformation was correct. Figure 31 shows an overview of the architecture for this approach.

UML to CSP Transformation

The UML to CSP transformation was presented as a case study at AGTIVE '07 [67], to compare the various graph transformation tools available today. We provide

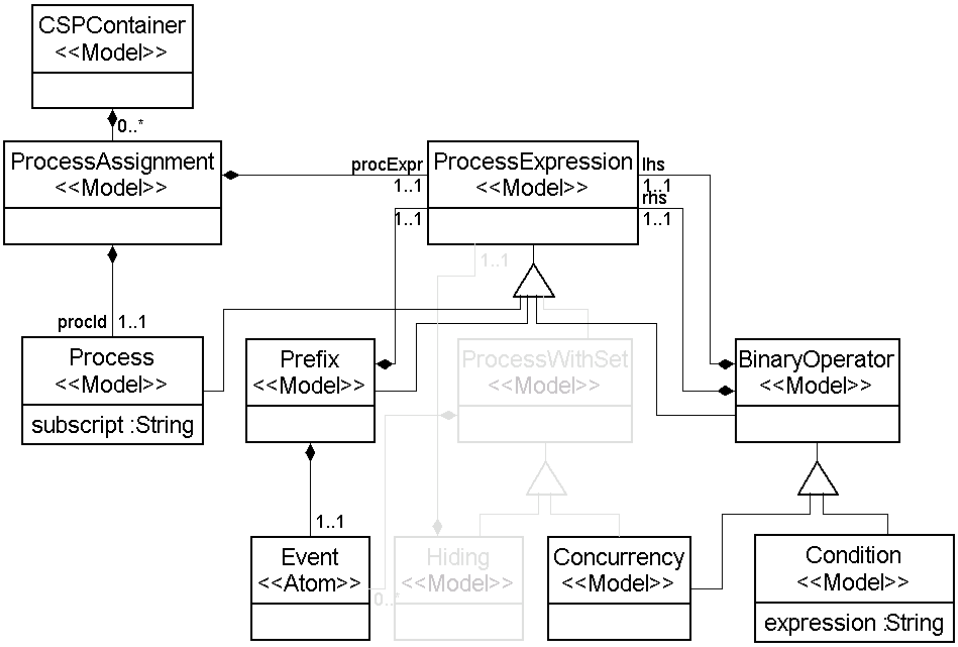


Figure 33: Meta-model for CSP

an overview of this case study here from a GReAT point of view, and we will use this as an example to explain our technique for verifying model transformations.

The objective of this transformation is to take a UML Activity Diagram [68] and generate a Communicating Sequential Process [69] model with the equivalent behavior. The Activity Diagram consists of various types of *Activity Nodes*, which can be connected by directed *Activity Edges*. Figure 32 shows the GME meta-model for UML Activity Diagrams. A CSP *Process* is defined by a *Process Assignment*, which assigns a *Process Expression* to a *Process Id*. *Process Expressions* can be a simple *Process*, a *Prefix* operator or a *BinaryOperator*. Figure 33 shows the GME meta-model for CSP, highlighting the relevant parts for our example.

The UML to CSP mapping assigns a *Process* for each *Activity Edge*. For each type of *Activity Node*, a *Process Assignment* is created, which assigns the Process corresponding to the incoming Activity Edge to a *Process Expression* depending on

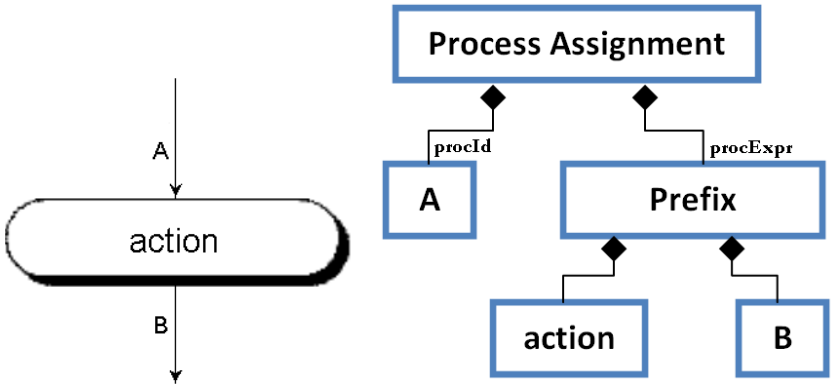
the type of the Activity Node. Figure 34 shows one such mapping, for the type *Action Node*. This assigns the incoming Process to a *Prefix* expression. The resulting CSP expression can be written as $A = action \longrightarrow B$, which is shown in Figure 34 as a model instance compliant with the CSP meta-model.

Structural Correspondence

As in the UML to CSP case, model transformations can be used to generate a target model of a certain structure (CSP) from a source model of a different structure (UML). Specific structural configurations in the source model (such as an *Action Node* in the UML model) produce specific structural configurations in the target model (such as a *Prefix* in the CSP model). The rules to accomplish the structural transformations may be simple or complicated. However, it is fairly straightforward to compare and verify that the correct structural transformation was made, if we already know which parts of the source structure map to which parts of the target structure.

In our technique to verify a transformation by structural correspondence, we will first define a set of structural correspondence rules specific to a certain transformation. We will then use cross-links to trace source elements with the corresponding target elements, and finally use these cross-links to check whether the structural correspondence rules hold.

In essence, we expect that the correspondence conditions are independently specified for a model transformation, and an independent tool checks if these conditions are satisfied by the instance models, after the model transformation has been executed. In other words, the correspondence conditions depend purely on the source and target model structures and not on the rewriting rules necessary to effect the transformation. Since the correspondence conditions are specified in terms of simple



(a) Activity Diagram (b) CSP

Figure 34: CSP Process Assignment for Action Node

queries on the model around previously chosen context nodes, we expect that they will be easier to specify, and thus more reliable than the transformation itself. We also assume that the model transformation builds up a structure for bookkeeping the mapping between the source and target models.

Structural Correspondence Rules for UML to CSP Transformation

A structural correspondence rule is similar to a precondition-postcondition style axiom. We will construct them in such a way that they can be evaluated easily on model instances. We will use the UML to CSP example to illustrate structural correspondence. Consider the case for the *Action Node*, as shown in Figure 34. The *Action Node* has one incoming edge and one outgoing edge. It is transformed into a *Process Assignment* in the CSP. The CSP structure for this instance consists of a *Process Id* and a *Prefix*, with an *Event* and a target *Process*. This is the structural transformation for each occurrence of an Action Node in the Activity Diagram.

We can say that for each Action Node in the Activity Diagram, there is a *corresponding* Process Assignment in the CSP, with a Prefix Expression. When our

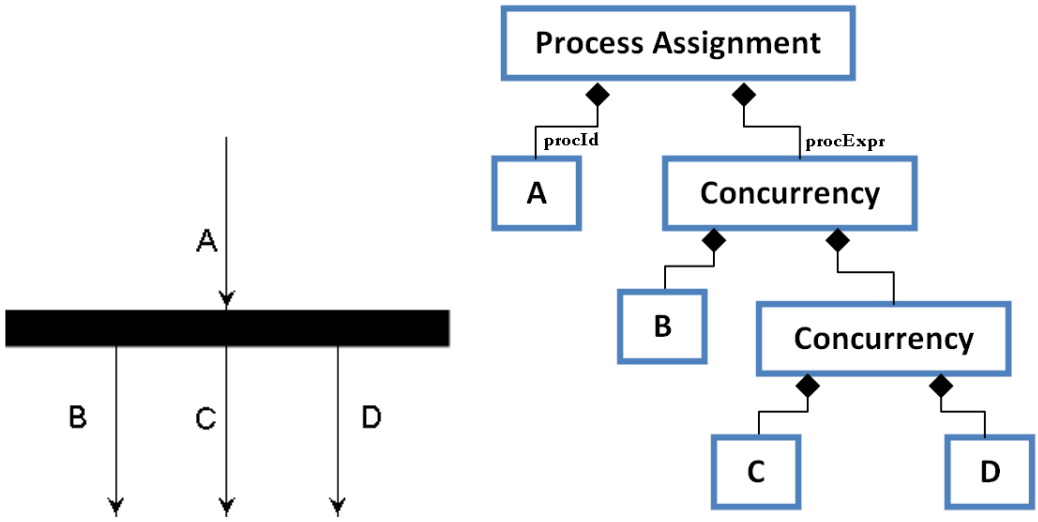
Rule	Path expression
The *Action Node* corresponds to a Process Assignment with a Prefix	$PA.procExpr.type = Prefix$
The incoming edge in the UML corresponds to the Process Id	$AN.inEdge.name = PA.procId.name$
The outgoing edge corresponds to the target Process	$AN.outEdge.name = PA.procExpr.Process.name$
The *action* of the Action Node corresponds to the Event	$AN.action = PA.procExpr.event$

Table 1: Structural Correspondence Rules for Action Node

transformation creates this corresponding Process Assignment, we can use a cross-link to track this correspondence. The structural correspondence is still not complete, as we have to ensure that the Process Id and the Prefix Expression are created correctly. We use a kind of *path expression* to specify the correctness of corresponding parts of the two structures, and the correspondence is expressed in the form $SourceElement = OutputElement$. Let us denote the Action Node by AN, and the Process Assignment by PA. Then the necessary correspondence rules can be written using path expressions as shown in Table VII.

These rules together specify the complete structural correspondence for a section of the Activity Diagram and a section of its equivalent CSP model. The different types of Activity Nodes result in different structures in the CSP model, some of which are more complex than the fairly straightforward case for the Action Node. Next, we look at the structural mapping for some of the other nodes.

A *Fork Node* in the Activity Diagram is transformed into a Process Assignment with a *Concurrency* Expression. Figure 35 shows a Fork Node with an incoming edge A and three outgoing edges B, C and D. This is represented by the CSP expression $A = B \parallel (C \parallel D)$, where $\parallel$ represents concurrency (the actual ordering of B, C and D is immaterial). The structural representation of this expression as an instance of



(a) Activity Diagram (b) CSP

Figure 35: CSP Process Assignment for Fork Node

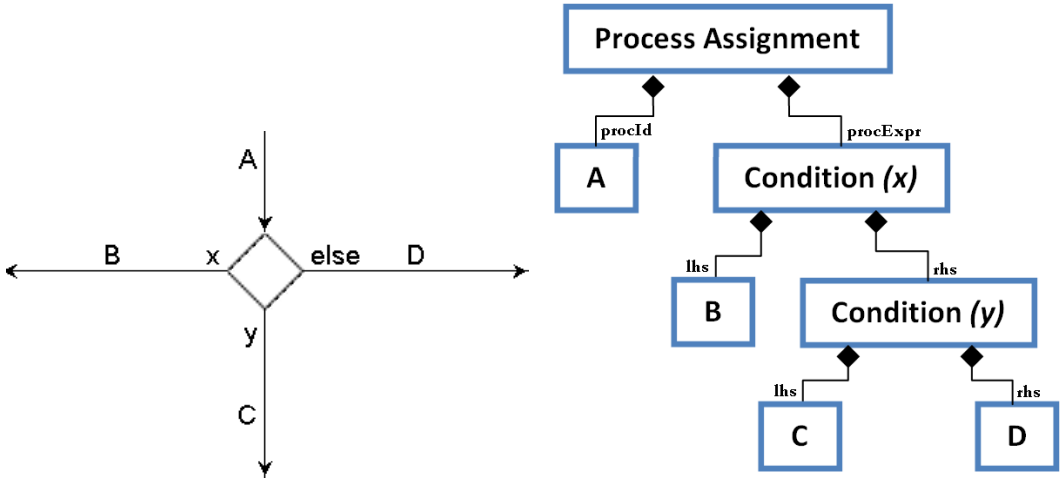
the CSP meta-model is shown in Figure 35. The Fork Node is transformed into a CSP Process Assignment that consists of a Process Id corresponding to the incoming Activity Edge of the Fork Node, and a Process Expression of type *Concurrency*. The Concurrency Expression consists of Processes and other Concurrency nodes, depending on the number of outgoing Activity Edges.

If we denote the Fork Node by FN and the Process Assignment by PA, the structural correspondence rules can be described using path expressions as shown in Table VII. We will use the double-dot '..' to denote the *descendant* operator (similar to '//' in XPath queries), to specify ancestor-descendant relationships.

By evaluating these rules on the Activity Diagram and the CSP models, we can determine whether the structural correspondence was satisfied for Fork Nodes. Another type of node in the Activity Diagram is the *Decision Node*. The transformation mapping for the Decision Node is a slight variation of the Fork Node. Figure 36 shows a Decision Node with an incoming edge A, and three outgoing edges B, C and D, with guards x, y and *else* respectively (in this case study, we will assume that the

Rule	Path expression
The *Fork Node* corresponds to a Process Assignment with a Concurrency	$PA.procExpr.type = Concurrency$
The incoming edge in the UML corresponds to the Process Id	$FN.inEdge.name = PA.procId.name$
For each outgoing edge, there is a corresponding Process in the Process Expression	$\forall o \in FN.outEdge$ $\exists p \in PA.procExpr..Process :$ $o.name = p.name$

Table 2: Structural Correspondence Rules for Fork Node



(a) Activity Diagram (b) CSP

Figure 36: CSP Process Assignment for Decision Node

Decision Node always has exactly one 'else' edge). This is represented by the CSP expression $A = B \not< x \not> (C \not< y \not> D)$, where the operator $C \not< y \not> D$ is the *condition* operator with the meaning that if y is *true*, then the process behaves like C, else like D.

The Decision Node is transformed to the CSP model shown in Figure 36 as a model instance of the CSP meta-model. The Decision Node is transformed into a Process Assignment that consists of a Process Id corresponding to the incoming Activity Edge of the Decision Node, and a Process Expression of type *Condition*. The Condition's

Rule	Path expression
The *Decision Node* corresponds to a Process Assignment with a Condition	$PA.procExpr.type = Condition$
The incoming edge in the UML corresponds to the Process Id	$DN.inEdge.name = PA.procId.name$
For each outgoing edge, there is a corresponding Condition in the Process Expression and a corresponding Process in the Condition's LHS	$\forall o \in DN.outEdge \wedge o.guard \neq else$ $\exists c \in PA.procExpr..Condition :$ $c.expression = o.guard \wedge$ $c.lhs.name = o.name$
For the outgoing 'else' edge, there is a Condition in the Process Expression with a corresponding Process as it's RHS	$\forall o \in DN.outEdge \wedge o.guard = else$ $\exists c \in PA.procExpr..Condition :$ $c.rhs.name = o.name$

Table 3: Structural Correspondence Rules for Decision Node

expression attribute is set to the *guard* of the corresponding Activity Edge, and a Process corresponding to the Activity Edge is created as it's LHS. For the final 'else' edge, a Process is created in the last Condition as it's RHS. The structural correspondence rules for this mapping are shown in Table VII.

Specifying Structural Correspondence Rules in GReAT

Specifying the structural correspondence for a transformation consists of two parts:

1. Identifying the significant source and target elements of the transformation
2. Specifying the structural correspondence rules for each element using path expressions

The first step is accomplished in GReAT by using cross-links between the source and target elements. A composite meta-model is added to the transformation, by associating selected source and target elements using a temporary 'Structural Correspondence' class. There will be one such class for each pair of elements. This class will

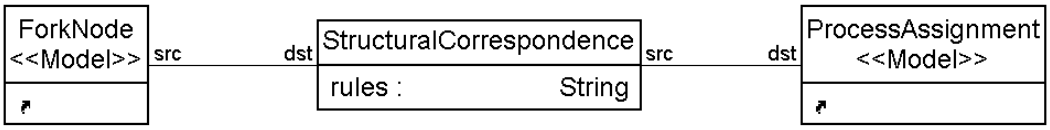


Figure 37: Composite Meta-model to Specify Structural Correspondence

have a string attribute, which is set to the path expressions necessary for structural correspondence for that pair. Figure 37 shows a composite meta-model specifying the structural correspondence for Fork Nodes. The *Fork Node* class comes from the Activity Diagram meta-model, and the *Process Assignment* class comes from the CSP meta-model.

Once the structural correspondence has been specified for all the relevant items, the transformation is enhanced to create the cross-link when creating the respective target elements. Figure 38 shows the GReAT rule in which the Process Assignment for a Fork Node is created, enhanced to create the cross-link for structural correspondence. Note that in incoming Activity Edge is represented as an *Association Class* named `ActivityEdgeIn`. The transformation for the Fork Node is actually accomplished in a sequence of several rules executed recursively, as shown in Figure 39. First all the Fork Nodes are collected, and a sequence of rules are executed for each node. These rules iterate through the out-edges of each Fork node, creating the Concurrency tree. Though several rules are involved in the transformation for Fork nodes, the cross-link needs to be added to one rule only.

It must be noted that it is necessary to specify the structural correspondence rules only once in the composite meta-model. The cross-link must however be added to the transformation rules, and in most cases will be required only once for each pair of source and target element.

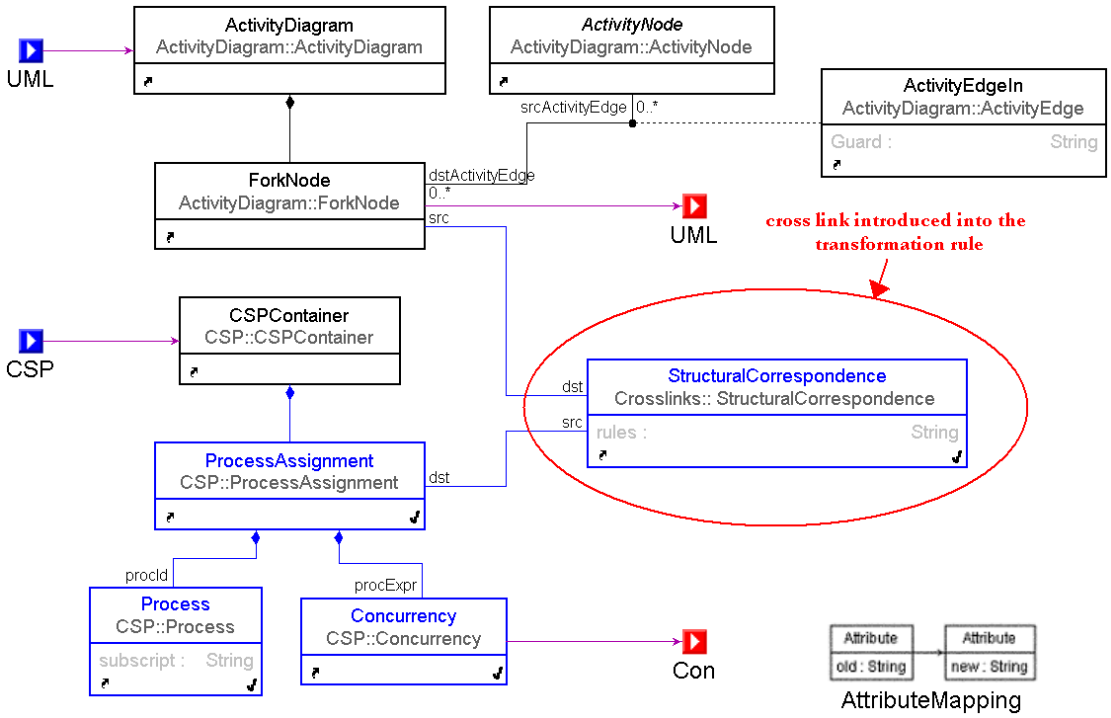


Figure 38: GReAT Rule with Cross-link for Structural Correspondence

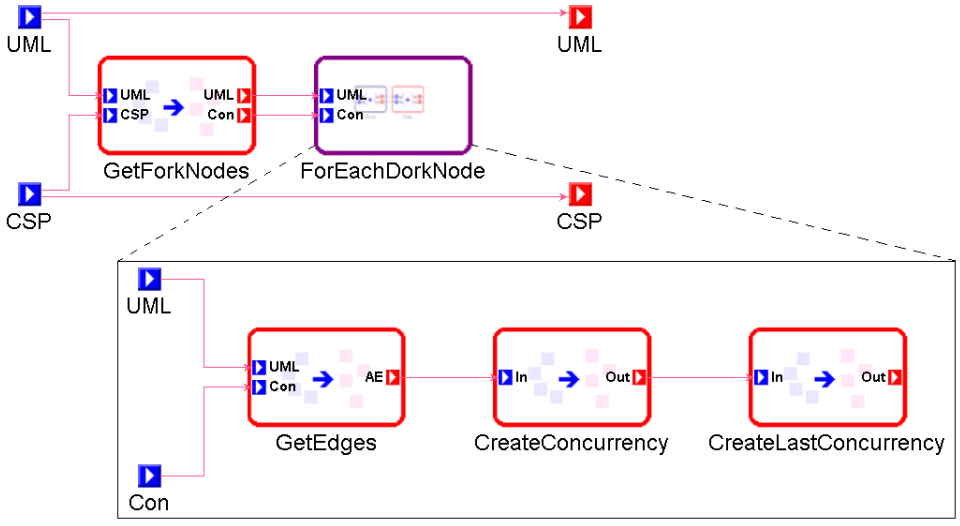


Figure 39: Sequence of GReAT Rules for Fork Node Transformation

Evaluating the Structural Correspondence Rules

Once the structural correspondence rules have been specified, and the cross-links added to the transformation, the correspondence rules are evaluated on the instance models after each execution of the transformation. These rules can be evaluated by performing a simple depth first search on the instance models, and checking if the correspondence rules are satisfied at each relevant stage.

This consists of two phases. The first phase is to generate the code that will traverse the instance models and evaluate the correspondence rules. Since the meta-models of both the source and target languages are available with the transformations, and the path expressions are written in a standard form that can be parsed automatically, the model traverser code can be automatically generated from the structural correspondence specification. This needs to be done only once each time the structural correspondence specification changes. The second phase is to call the model traverser code at the end of each execution of the transformation, supplying to it the source and target model instances along with the cross-links.

In the case of the UML to CSP transformation, we traverse the input Activity Diagram model and evaluate the correspondence rules at each activity node. For each Activity Node, the cross-link is traversed to find the corresponding Process Assignment. If a correspondence rule has been defined for an Activity Node, and no corresponding Process Assignment is found, then this signals an error in the transformation. After locating the corresponding Process Assignment, the path expressions are evaluated. If any of the rules are not satisfied, the error is reported. If all the rules are satisfied for all the nodes, then we can conclude that the transformation has executed correctly.

The instance model is traversed in a depth-first manner. The corresponding elements are located using the cross-links, which will take constant time. The path

expressions are evaluated on the instances, which will take polynomial time in most cases. Thus, the overall verification process does not incur a significant performance overhead in most cases.

Remarks

The structural correspondence based verification described here can provide important assurances about the correctness of transformations, while being practically applicable in most common transformations.

The use of path expressions to specify correspondence rules makes it easy to specify correctness. The path expressions use a simple query language that can be easily evaluated on the instance models. Our future research concentrates on the requirements of such a query language. Most complex transformations may involve multiple rules executing recursively to transform a particular part of a model. However, it may be possible to specify the correspondence for that part of the model using a set of simple path expressions. Such a specification would be simpler and easier to understand than the complex transformation rules.

The structural correspondence is also specified orthogonal to the transformation specification. Thus, these rules may be written by someone other than the original transformation writer, or even supplied along with the requirements document.

Related Work

[70], [71] present ideas on validating model transformations. In [71], the authors present a concept of rule-based model transformations with control conditions, and provide a set of criteria to ensure termination and confluence. In [70], Küster focuses on the syntactic correctness of rule-based model transformations. This validates whether the source and target parts of the transformation rule are syntactically

correct with respect to the abstract syntax of the source and target languages. These approaches are concerned with the functional behavior and syntactic correctness of the model transformation. [72] also discusses validation of transformations on these lines, but also introduces ideas of syntactic consistency and behavior preservation. Our technique addresses semantic correctness of model transformations, addressing errors introduced due to loss or mis-representation of information during a transformation.

In [73], Bisztray and Heckel present a rule-level verification approach to verify the semantic properties of business process transformations. CSP is used to capture the behavior of the processes before and after the transformation. The goal is to ensure that every application of a transformation rule has a known semantic effect. We use path expressions to capture the relation between structures before and after a transformation. These path expressions are generic (they do not make any assumptions about the underlying semantics of the models involved), and can be applied to a wide variety of transformations.

Ehrig et. al. [74] study bidirectional transformations as a technique for preserving information across model transformations. They use triple graph grammars to define bi-directional model transformations, which can be inverted without specifying a new transformation. Our approach offers a more relaxed framework, which will allow some loss of information (such as by abstraction), and concentrates on the crucial properties of interest. We also feel that our approach is better suited for transformations involving multiple models and attribute manipulations.

In other related work, [75] presents a model level technique to verify transformations by model checking a selected semantic property on the source model, and transforming the property and validating it in the target domain. The validation requires human expertise. After transforming the property, the target model is model

checked. In our approach, since the properties are specified using cross links that span over both the source and target languages, we do not need to transform them. [76] discuss an approach to validate model transformations by applying OCL constraints to preserve and guarantee certain model properties. [77] is a language level verification approach which addresses the problem of verifying semantic equivalence between a model and a resulting programming language code.

MOF QVT Relations Language

The MOF 2.0 Query / View / Transformation specification [29] addresses technology pertaining to manipulation of MOF models. A *relations language* is prescribed for specifying relations that must hold between MOF models, which can be used to effect model transformations. Relations may be specified over two or more domains, with a pair of *when* and *where* predicates. The *when* predicate specifies the conditions under which a relation must hold, and the *where* predicate specifies the condition that all the participating model elements must satisfy. Additionally, relations can be marked as *checkonly* or *enforced*. If it is marked *checkonly*, the relation is only checked to see if there exists a valid match that satisfies the relationship. If it is marked *enforced*, the target model is modified to satisfy the relationship whenever the check fails.

Our approach can be likened to the *checkonly* mode of operation described above. However, in our case, the corresponding instances in the models are already matched using cross links, and the correspondence conditions are evaluated using their context. The cross links help us to avoid searching the instances for valid matches. Specifying the correspondence conditions using context nodes simplifies the model checking necessary to evaluate the conditions, thus simplifying the verification process. Since we verify the correspondence conditions for each instance generated by the transformation, these features play an important role.

Triple Graph Grammars

Triple Graph Grammars [66] are used to describe model transformations as the evolution of graphs by applying graph rules. The evolving graph complies with a graph schema that consists of three parts. One graph schema represents the source meta model, and one represents the target meta-model. The third schema is used to track correspondences between the source and target meta models. Transformations are specified declaratively using triple graph grammar rules, from which operational rules are derived to effect model transformations.

The schema to track correspondences between the source and target graphs provides a framework to implement a feature similar to cross links in GReAT. If the correspondence rules can be encoded into this schema, and the correspondence links persisted in the instance models, our verification approach can be implemented in this scenario.

Conclusions and Future Work

In this paper, we have shown how we can provide an assurance about the correctness of a transformation by using structural correspondence. The main errors that are addressed by this type of verification is the loss or misrepresentation of information during a model transformation.

We continue to hold to the idea that it is often more practical and useful to verify transformations on an instance basis. The verification framework must be added to the transformation only once, and is invoked for each execution of the transformation. The verification process does not add a significant overhead to the transformation, but provides valuable results about the correctness of each execution.

The path expressions must use a simple and powerful query language to formulate queries on the instance models. While existing languages such as OCL may be suitable

for simple queries, we may need additional features, such as querying children to an arbitrary depth. Our future research concentrates on the requirements of such a language.

While the path expressions can be parsed automatically and evaluated on the instances, the cross-link for the relevant elements must be manually inserted into the appropriate transformation rules. However, in most cases, it may be possible to infer where the cross-links must be placed. If the cross-links could be inserted into the rules automatically, the transformation can remain a black box. The main concern with this is that the cross-links are crucial to evaluating the correspondence rules correctly and also to keep the complexity down.

We have seen simple string comparisons added to the path expressions in this paper. Some transformations may require more complex attribute comparisons, or structure to attribute comparisons such as counting. We wish to explore such situations in further detail in future cases, to come up with a comprehensive language for specifying the path expressions.

CHAPTER VIII

SPECIFYING THE CORRECTNESS PROPERTIES OF MODEL TRANSFORMATIONS

Overview of this Paper

The previous paper introduced the idea of structural correspondence for verifying transformations that performed largely structural changes on the instance models. This paper treats the subject of structural correspondence in greater detail, with special attention to the language for correctness specification and the details of checking them on the instance models.

The specification of the correctness conditions uses correspondence information between the two domains along with structural information from the models. This requires a query language that is both easy to use, and is powerful enough to describe correspondences between complex structures. This includes the ability to navigate the structure of the models, and the ability to use quantifiers to specify correspondence requirements. The requirements of such a query language are studied in this paper.

This paper introduces some background information, and describes in detail all the steps involved in verifying a transformation using structural correspondence. The requirements of the query language are described by looking at various types of correspondence rules.

Introduction

Model based software development has advanced to a level of maturity where most artifacts in the design and development stages could be produced by automated model transformations. The success of such a development effort hinges on the correctness of these automated model transformations. The term *correctness*, in this context, involves several facets such as termination of the transformation algorithm, confluence of the transformation rules and conformance to a language syntax or meta-model. One important criterion for correctness is whether the transformation achieved its objective, whether it resulted in the desired output model. While the former properties can be stated implicitly within the framework of the transformation language, the notion of correctness in the latter case involves a more detailed knowledge of the source and target domains and the transformation objective from a domain-specific point of view. We present a technique to specify such a correctness, using a language framework that can easily be incorporated into a variety of domains.

Models can be seen as typed, attributed graphs, and model transformations as manipulations on such graph structures. In other words, a source model of a certain graph structure is transformed into a target model of a different structure. In most transformation cases where the correctness problem is significant, there is a correlation or *correspondence* between parts of the input model and parts of the output model. If we can specify these correlations in terms of the abstract syntax of the source and target languages, and have a framework to verify whether the correlations hold, then we can verify whether the desired output model was created. We call these correlations *structural correspondence*. Our thesis for the approach is as follows: if a transformation has resulted in the desired output models, there will be a verifiable structural correspondence between the source and target model instances that is decidable. This idea was introduced in [78] with a simple case study. In this paper, we

will concentrate on the requirements of a query language for specifying correctness in this form, and look at more detailed examples that illustrate these requirements.

Background

In this section, we review some background information.

Instance based verification

In our previous work on verifying model transformations [79], we used the underlying behavior models of the source and target languages, and framed the verification problem in terms of finding a bisimulation between the source and target models. One key feature of the approach was that we focused on verifying each execution of the transformation, as opposed to finding a correctness proof for the transformation specification. We believe that this approach is more simple and pragmatic. Figure 40 shows the framework we used.

We extended the transformation framework to trace relations between elements of the source and target instances. At the end of the transformation's execution, we passed these links to a bisimilarity checker, which checked if there was a bisimulation between the source and target models. This helped us to conclude whether a behavioral property was preserved by that execution of the transformation. The instance model is then said to be 'certified correct'. Note that while the verification is performed for each execution of the transformation, the transformation program had to be augmented with the additions for the verification framework only once.

While this example is concerned with behavioral properties, we will consider verification from a structural viewpoint in the rest of this paper. However, we will continue to focus on verifying each execution of a transformation instead of finding general correctness proofs.

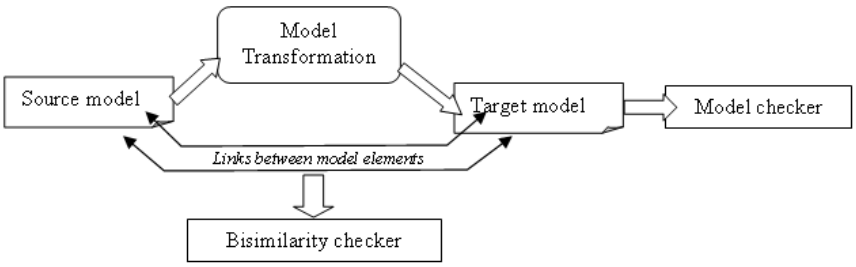


Figure 40: Instance Level Verification of Transformations

GReAT

GReAT [58] is a language and transformation framework for specifying and executing meta-model based model transformations using graph transformation rules. GReAT is implemented with the framework of GME [57]. The source and target meta-models of the transformation are specified using UML class diagrams, and transformation rules are constructed using these classes. GReAT also allows transformation designers to create new vertex and edge types in addition to those defined in the source and target meta-models. The transformation can thus be written over a composite meta-model that consists of the source and target languages, and some cross-language elements. This allows us to track associations between elements of the source and target instances during the course of a transformation. Such associations are called *cross links*.

Class to RDBMS Transformation

The "Class to RDBMS" transformation example [80] was presented as a challenge problem in the 2005 Model Transformations in Practice Workshop. We will use this example to explain some of the ideas in this paper. A short description of the problem is presented here.

Figure 41 shows the meta model for Classes and Associations. Classes can contain one or more Attributes, with the additional constraint that there is at least one

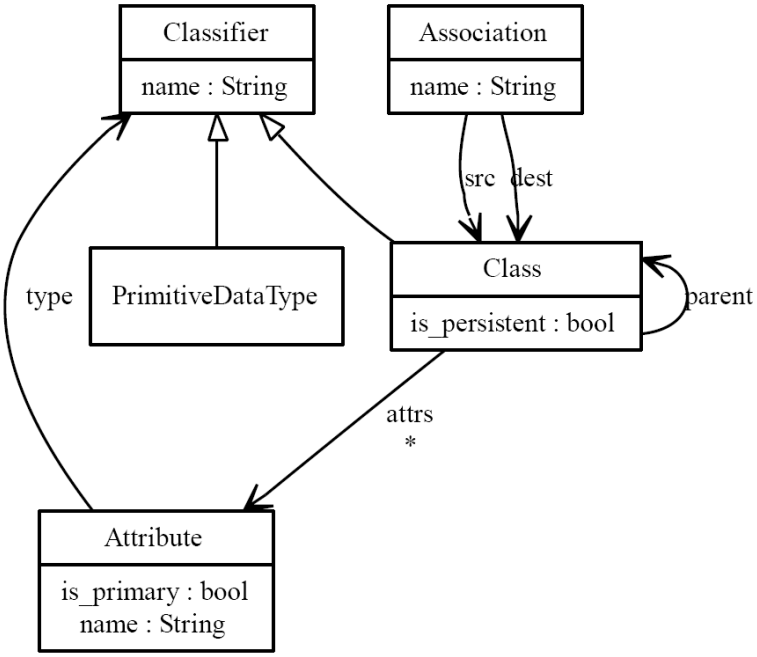


Figure 41: Class meta-model [80]

primary attribute. Figure 42 shows the meta model for RDBMS. An RDBMS model contains one or more Tables, each with one or more Columns. One or more of these Columns forms the *primary key* of the table. A table may contain zero or more *foreign keys*.

The goal of the transformation is to create an RDBMS representation from a class diagram, based on a number of rules. A *Table* is created for each top level *Class* in the source model which has its *is_persistent* attribute set to *true*. The *Attributes* and *Associations* of the class are transformed into *Columns* of the corresponding table, and the primary and foreign keys are also set appropriately. The complete details of the transformation requirements can be found in [80].

We will not attempt to provide a solution for this transformation here, we refer the reader to [81] for some solutions to the transformation. We will simply use it below to illustrate how a verification framework can be built around such a transformation.

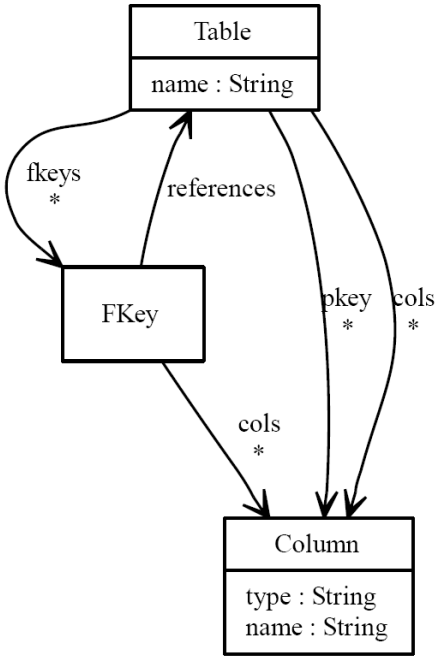


Figure 42: RDBMS meta-model [80]

Specifying Correctness by Correspondence

We wish to ensure that a model transformation executed correctly, by verifying whether it produced the desired output model. To accomplish this, we need a specification of what constitutes a desired model. This specification must not be confused with the specification of the transformation itself. In most cases, it is sufficient that the output model satisfies a small number of constraints based on the source model elements and attributes, to be accepted as correct. However, the transformation that produces such an output model may have to deal with intricate issues, and thus be much more complex. Further, with a simplified language for specifying such conditions, and the use of cross links, we can construct a pragmatic framework to verify the correctness of the output model.

In typical model transformations, we frequently wish to create a structure in the output model corresponding to some structure in the input model. The transformation can be accepted as correct, if a node in the source model and its corresponding node in the target model satisfy some *correspondence conditions*. In this case, a tractable solution to the verification problem can be provided under the following conditions: 1. A map is maintained to match the corresponding nodes in the source and target model instances; 2. Correspondence conditions are specified in terms of these previously identified and matched nodes. Our approach consists of the following steps.

Identifying Correspondence Structures

The first task is to identify a sufficient set of node types from the source language that must have a corresponding element in the target model. We call these *pivot nodes*. While all the node types being transformed can be considered here, it may be pragmatic to only choose a smaller set of significant nodes - which either undergo complex transformations or are critical to the correctness of the output model.

For instance, in the Class to RDBMS transformation example, *Class* nodes are transformed into *Table* nodes, and *Attribute* nodes are transformed into *Column* nodes. These two pairs can be identified as the pivot nodes for this problem. The verification problem is to ensure that the *Tables* and *Columns* are correctly created in the output model corresponding to the *Classes* and *Attributes* in the source model. Cross links are defined between such pairs, which will carry the correspondence specifications and will later assist in the verification of the correspondence conditions.

Specifying Correspondence Conditions

Once the pivot nodes have been identified, the correctness condition must be specified for each pair of pivot nodes. The correctness condition can use some form of query that can be performed on the instance models. This query could involve traversing the immediate hierarchy of the nodes, access the nodes' attributes and associations etc. This is explained in greater detail in Section VIII.

Nodes of type *Class* in the source model correspond to nodes of type *Table* in the target model. The correspondence rule must ensure that for every such pair in the model being transformed, the *is_persistent* attribute of the *Class* node is set to *true*. The correspondence must further state that there exists a *Column* for each *Attribute* of the class and one for each *Association* where the class acts as the source, which are in turn related by correspondence conditions. This is specified using existential quantifiers that make use of the cross link relation.

Creating cross links

Cross links are used to construct a look-up table (i.e. a map) that matches the corresponding pivot nodes of any instance of a transformation execution. The cross links are crucial to have a tractable and reliable verification framework. The transformation must be extended by creating a new cross link between the pivot nodes in every transformation rule that creates the relevant target node. The transformation has to be extended this way only once.

The cross links are specified at the meta level, by drawing a link between the source node and its corresponding target node. To clarify the specification of the correspondence conditions, we will use a *StructuralCorrespondence* class that carries the correspondence rules for a specific pair of source and target nodes. We can then connect the source and corresponding target nodes through this intermediate class

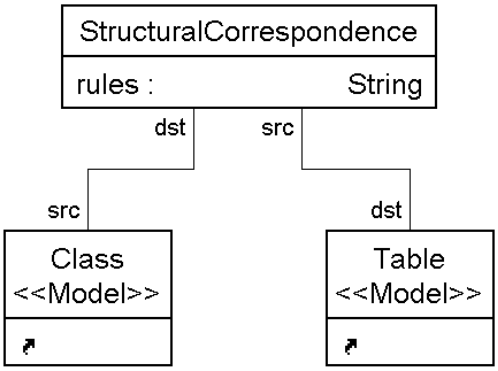


Figure 43: A cross link to specify structural correspondence

using cross links, as shown in Figure 43. This gives rise to a composite meta model that includes the source and target languages along with the cross links. Since the transformation rules are defined by referencing the meta types in the pattern matching rule, it may be possible to insert the cross links automatically into the relevant rules of the transformation. For instance, a cross link is created between *Class* in the source meta and it Table in the target meta. Similarly, cross links are created between *Attribute* and *Column*, and *Association* and *Column*.

Checking the Correspondence Conditions

At the end of each transformation run the source and target instance models, along with the cross links, are passed to a model checker. The model checker uses these cross links to check if the verification conditions hold for all pairs of pivot nodes through out the model instances. This is explained in Section VIII.

Design of a Query Language for Specifying Correspondence

It is important to note that the correspondence conditions are not intended to specify complete transformations using complex pattern matching, but simply list some conditions to help to satisfactorily conclude that the correct transformation was

made. This simplifies the query language to a degree. Also, the verification conditions are checked on specific nodes as opposed to arbitrary subgraphs, and have the option of looking up matches using cross links. The instance model is traversed exhaustively, and a check is performed on each node that has a correspondence specification. Thus, there is a specific context in which each condition is evaluated. The query language is expected to have the following features.

Querying Attributes and Associations

It must be possible to reference the attributes and associations of the pivot nodes based on the abstract syntax of the languages. A simple OCL style path query of the form *class.attribute* or *class.association* suffices here. Since they are executed per instance of the pivot node, it is always expected to terminate with a finite result.

Querying Up or Down a Containment Hierarchy

We may need to reference the pivot nodes' parent or child nodes to specify certain correspondence conditions. Most query languages (such as OCL) allow querying child nodes using the child role names. A similar notation for finding the pivot nodes' parents is required. Further, some structural correspondence conditions may require querying child nodes at an arbitrary depth. The query language can be extended to use a double dot '..' notation to query all contained child nodes at any level of depth in the hierarchy. Since model hierarchies cannot contain cycles, this query will always terminate with a finite result.

Using Quantifiers with Queries

Finally, we may need to add quantifiers to the query string to frame the correspondence conditions. For instance, we may want to use statements like 'a Column

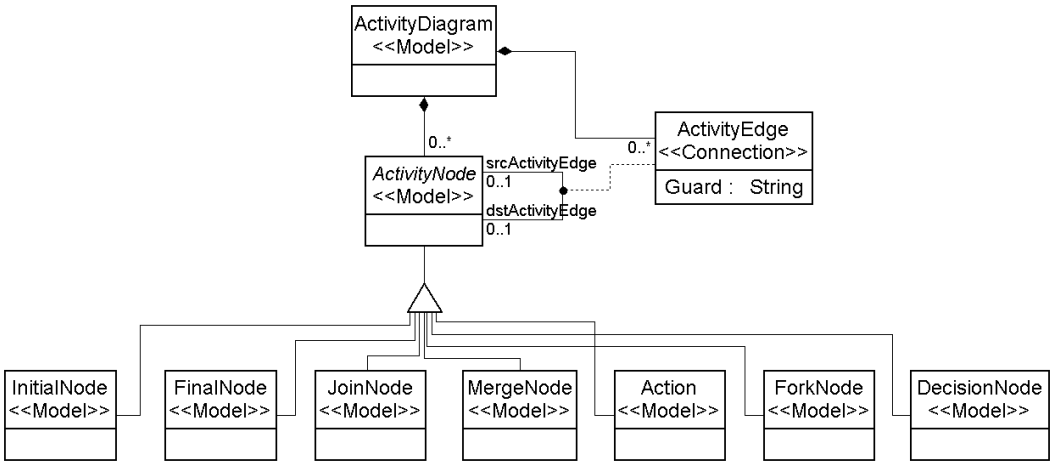


Figure 44: Meta-model for UML Activity Diagrams

is created *for each* Attribute of a Class'. The standard quantifiers such as $\exists$ (there exists at least one), $\forall$ (for all) and $\exists!$ (there exists exactly one) can be used to specify most of the correspondence conditions. We conjecture that it is useful to also have specialized quantifiers that take advantage of correspondence information stored in the cross links, by imposing some restrictions on the standard quantifiers. For instance, $\exists_C$ can be used to represent 'there exists, attached via a cross link'. Such a statement is useful in cases where an object in the source model must correspond to a number of objects in the target model. This reduces the search space and simplifies the verification.

Checking the Verification Conditions

At the end of each execution of the transformation, the verification conditions are checked on the instance model. This is performed by a model checker that performs an exhaustive scan of the instance models, applying the verification conditions on all the relevant nodes. This model checker must be tailored to traversing instance models

of the relevant domains and evaluating the relevant verification conditions. We propose the use of a generic model checker, that can be customized automatically using the meta-models of the source and target languages, and the verification conditions specified in a standardized form. The steps of the generic model checker are listed in Algorithm VIII.0.1.

Case Study: UML to CSP Transformation

In this section, we look at some application examples of verification by structural correspondence, using a more detailed case study. We will consider a transformation from UML Activity Diagrams [68] to a CSP (Communicating Sequential Processes) [69] specification. Figure 44 shows the meta model for UML Activity Diagrams. Activity Diagrams consist of Activity Nodes and Activity Edges. Figure 45 shows the meta model for CSP. A CSP model consists of a number of Process Assignments, which assign a Process Id to some Process Expression. The Process Expression can be of various types such as Prefix, Concurrency, Condition etc.

The objective of the UML to CSP transformation is to obtain a CSP specification from an activity diagram, by creating CSP process assignments that mimic the behavior of the activity diagram. A CSP Process represents an Activity Edge, and is assigned to a Process Expression based on the type of the Activity Node involved. The complete description of this transformation can be found in [67]. We will look at some specific portions of the transformation here.

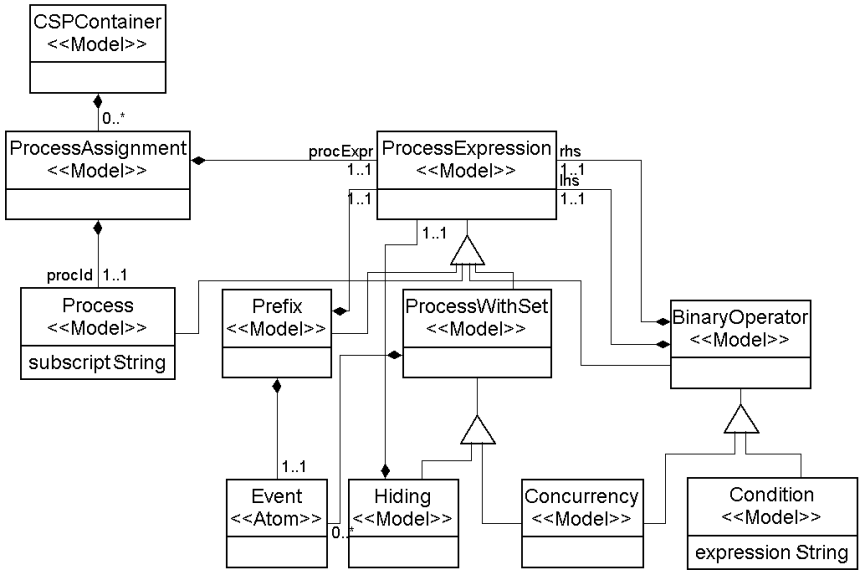


Figure 45: Meta-model for CSP

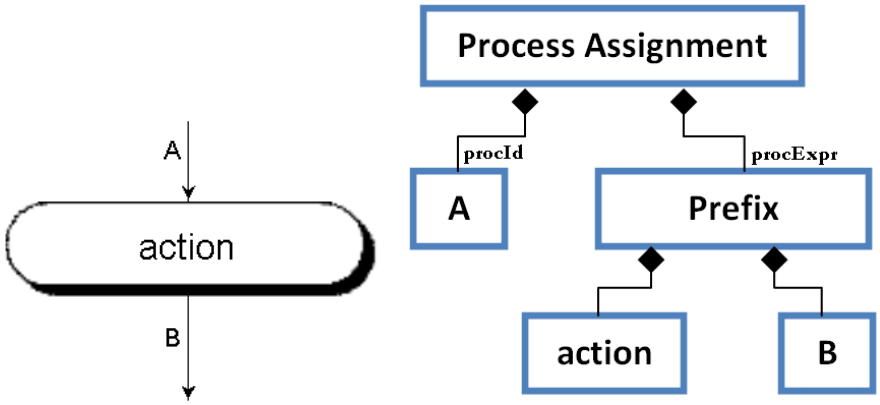
Algorithm VIII.0.1: CHECKCORRESPONDENCE(*instances*, *crosslinks*, *metamodels*, *correspondence − rules*)

---

for each instance node
  if type has correspondence rules **then**
    find corresponding target nodes
    if failed **return** (*false*)
    for each *rule*
      Evaluate (rule)
      if failed **return** (*false*)
    end for
  end if
end for
return (*true*)

Action Nodes

Action Nodes in the activity diagram are represented by Process Assignments using Prefix expressions. Action Nodes have one incoming Activity Edge and one outgoing Activity Edge. The expected transformation for Action Nodes in the activity diagram is shown in Figure 46.



(a) Activity Diagram (b) CSP

Figure 46: CSP Process Assignment for Action Node

The condition for correspondence in this case is simply that each (UML) Action node has a corresponding (CSP) Process Assignment, with the Process Id set to the incoming edge of the Action Node, and the Process Expression is a Prefix expression that encodes the *action* of the Action node as its *event* and the outgoing edge of the Action node as its target process. To specify this correspondence, we create a cross link between the Action node and the Process Assignment, and encode the correspondence conditions for the cross link. This relates each instance of type Action Node in the activity diagram model to an instance of type Process Assignment in the CSP model, which is directly identified by the cross link between the instances. If we denote the (UML) Action node by AN and the corresponding (CSP) Process Assignment by PA, we will have the following conditions:

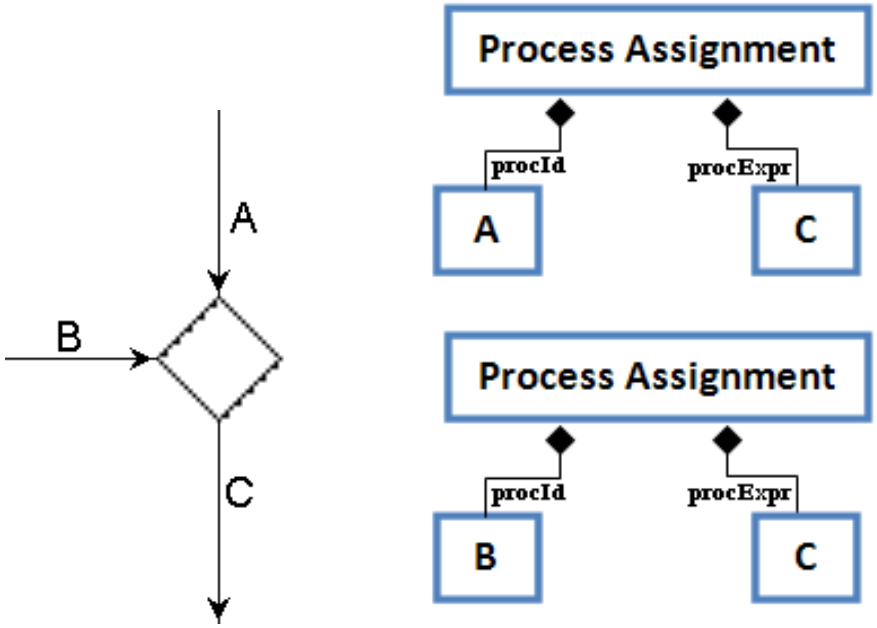
- $PA.procExpr.type = Prefix$
- $AN.inEdge.name = PA.procId.name$
- $AN.action = PA.procExpr.event$
- $AN.outEdge.name = PA.procExpr.process.name$

Note that this set of verification conditions is checked for each instance of an Action node in the model being transformed. At the end of transformation, the instance models are traversed, and for each AN found, the corresponding PA is located through the cross link (if no PA is found, that itself indicates an error in the transformation). With the AN and its corresponding PA, the verification conditions are checked. If all conditions are satisfied, we move on to the next object instance. If a condition is not satisfied, then an error has been found in the transformation.

Merge Nodes

Merge nodes merge multiple activity edges into a single activity edge. This is performed by simply assigning the process corresponding to each incoming edge to the process corresponding to the outgoing edge. Figure 47 shows the transformation for Merge nodes.

In this transformation, as many Process Assignments must be created as there are incoming edges for the merge node. To establish the correspondence, each of the Process Assignment nodes must be cross linked to the Merge node. When specifying the correspondence, however, we need to create only one cross link between the Merge Node type (in the Activity Diagram meta) and the Process Assignment type (in the CSP meta). The correspondence condition must ensure that the required number of Process Assignments were created, and that a Process corresponding to each of the incoming activity edges is assigned to a Process corresponding to the outgoing activity



(a) Activity Diagram (b) CSP

Figure 47: CSP Process Assignment for Merge Node

edge. To specify this correspondence, we make use of the specialized quantifier $\exists_C$. If a Merge Node is denoted by MN and a Process Assignment is denoted by PA:

- $\forall\ MN.inEdge\ \exists_C\ PA:$

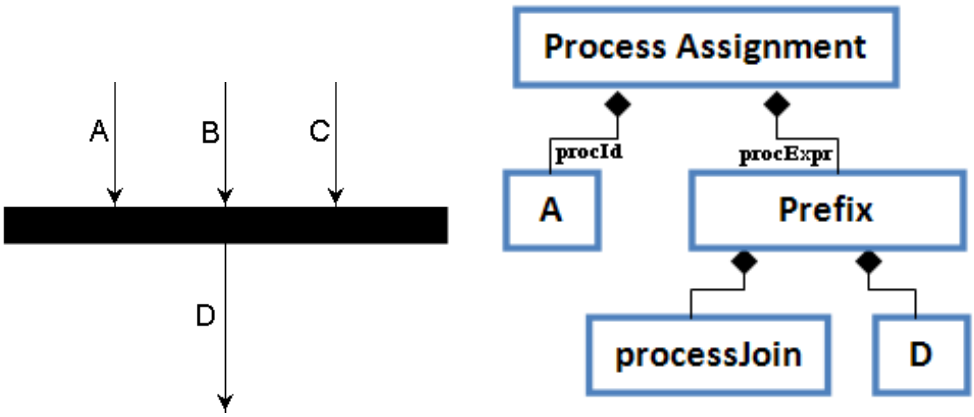
  $PA.procId.name = MN.inEdge.name$

  $\wedge PA.procExpr.name = MN.outEdge.name$

The verification condition is checked for each occurrence of a Merge node in the instance model being transformed. Each MN will have a finite number of $inEdges$, and a finite number of cross-linked PAs, and the condition can be verified by a model checker by traversing the instance models. This example shows the advantage of having a specialized quantifier like $\exists_C$ that can use the information represented by cross links.

Join Nodes

Join Nodes can be thought of as the 'synchronized' version of Merge nodes. Join nodes also have multiple incoming activity edges and one outgoing activity edge, but represent the semantics that the associated processes must wait for each other before joining. This is represented in the CSP by using a Prefix expression and a special event *processJoin* that synchronizes the processes involved. The Join Node is transformed such that one Process (corresponding to some incoming edge) is chosen to go on with the continuation Process (corresponding to the outgoing activity edge), while the other Processes simply terminate in a $SKIP$. For the Join Node shown in Figure VIII.48(a), the CSP process assignments would be: $A = processJoin \longrightarrow D; B = processJoin \longrightarrow SKIP; C = processJoin \longrightarrow SKIP$. Figure 48 shows the transformation for Join Nodes, with the expected CSP model structure for the the first Process Assignment. For the remaining Process Assignments, the target process D is replaced by $SKIP$.



(a) Activity Diagram (b) CSP

Figure 48: CSP Process Assignment for Join Node

Like in the case of Merge nodes, multiple Process Assignments must be created for each Join node, depending on the number of incoming edges incident on the Join

node. But unlike the previous case, we have the additional condition that the target process corresponds to the outgoing edge for exactly one of the Process Assignments (it does not matter which), and is $SKIP$ for the rest of the Process Assignments. If a Join Node is denoted by JN and a Process Assignment by PA, we have the following conditions:

- $\forall \;\; JN.inEdge \;\; \exists_C \;\; PA :$

  $PA.procId.name = JN.inEdge.name$

  $\wedge\, PA.procExpr.type = Prefix$

  $\wedge\, PA.procExpr.event = \text{`}processJoin\text{'}$

- $\forall \;\; JN.inEdge \;\; \exists! \;\; PA :$

  $PA.procExpr.process.name = JN.outEdge.name$

- $\forall \;\; JN.inEdge \;\; \exists! \;\; PA :$

  $PA.procExpr.process.name \neq SKIP$

These conditions are checked for each occurrence of a Join node in the instance model being transformed. The first condition checks whether a PA has been created corresponding to each incoming activity edge in the source model. The second condition checks if exactly one of these has the target expression set to a Process corresponding to the outgoing activity edge. The last condition ensures that there is only one such PA, and the rest of the processes terminate in a $SKIP$.

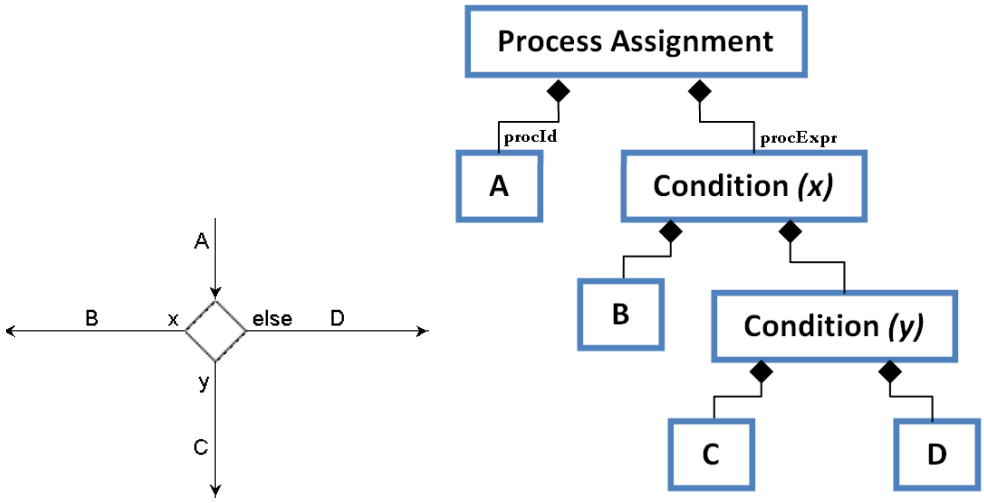
Decision Nodes

Decision nodes have one incoming edge and several outgoing edges, with a *guard* on each outgoing edge. We also assume that at least one of these guards is *else*. Decision nodes are represented using Condition expressions in the CSP. The CSP Condition is a binary expression, with an *lhs* and an *rhs* Process expression, and a

condition expression that captures the guard for the condition. Due to the binary nature of the condition expression, decisions involving more than one condition must be represented using a binary tree structure. The Decision node shown in Figure VIII.49(a) is represented by the CSP process assignment $A = B \not< x \not> (C \not< y \not> D)$. The statement $C \not< y \not> D$ represents a binary condition which states that if y is *true*, then the process behaves like C, else like D. Figure VIII.49(b) shows the CSP structure for this transformation.

Each Decision node is transformed into a single Process Assignment, which can be identified using the cross link. To verify if the Decision node was transformed correctly, we must check that each outgoing activity edge is represented as the *lhs* of a corresponding Condition which has its *expression* set to the *guard* of the activity edge, and there exists a Condition whose *rhs* captures the *else* activity edge. Since we must traverse the binary tree to a child node at an arbitrary depth to verify these conditions, we will make use of the '..' notation. If a Decision Node is represented by DN and its corresponding Process Assignment is represented by PA, then we will have the following conditions:

- $PA.procId.name = DN.inEdge.name$
- $PA.procExpr.type = Condition$
- $\forall o \in DN.outEdge \wedge o.guard \neq else$
  $\exists c \in PA.procExpr..Condition :$
  $c.expression = o.guard$
  $\wedge\, c.lhs.name = o.name$
- $\forall o \in DN.outEdge \wedge o.guard = else$
  $\exists c \in PA.procExpr..Condition :$
  $c.rhs.name = o.name$



(a) Activity Diagram (b) CSP

Figure 49: CSP Process Assignment for Decision Node

These conditions are verified for each occurrence of a DN in the instance model being transformed. The corresponding PA is located using the cross link, and its children are navigated to check the verification conditions. Since the nodes are identified using the cross links, the complexity is not expected to be high.

Finally, the model checker is customized to evaluate these correspondence rules on the instance models. The generic model checker listed in Algorithm VIII.0.1 is extended for this domain and these rules. The listing in Algorithm VIII.0.2 shows an overview of the customized model checker steps.

We have seen a number of cases where the correctness of the transformation has been specified using a structural correspondence. The examples we have seen are only representative of the kind of conditions that can satisfactorily determine correspondence. The actual number and depth of detail of these conditions depends on the domain and the application. This brings us to the question of completeness of the correspondence specification. Further research is needed to study the nature of these correspondence conditions in complex scenarios. Completely specifying semantic properties is always a difficult problem due to the variabilities of different

domains. We believe that our approach of using context nodes and correspondences simplifies the problem to an extent, and could prove to be sufficient to completely specify correspondences in most cases.

Algorithm VIII.0.2: checkCorrespondence(*instances*, *crosslinks*, *metamodels*, *correspondence_rules*)

```
for each activity node N
  if type is Action  then
    PA := follow_cross_link(N)
    if failed, return (false)
    /*Evaluate Rules*/
    if not (PA.procExpr.type = Prefix)
      return (false)
    if not (PA.procId.name = N.inEdge.name)
      return (false)
    if not (PA.procExpr.event = N.action)
      return (false)
    if not (PA.procExpr.process.name =
      N.outEdge.name) return (false)
    continue
  else if type is Merge  then
    ...
  end if
end for
return (true)
```

Note that in each of these cases, the rules necessary to accomplish the transformation may be quite complex, requiring several transformation steps and possibly requiring recursion. However, the correspondence conditions are relatively simple to specify and check using a simple brute force model checking approach. Moreover, the same verification conditions are independent of the algorithm chosen for the transformation, and depend only on the source and target structures. This makes it possible to plug them into existing transformation solutions to verify their correctness.

Related Work

In this section, we review some related work.

The OMG QVT Relations Language

The MOF 2.0 Query / View / Transformation specification [29] provides a language for declaratively specifying transformations as a set of relations that must hold between models. A relation is defined by two or more domains, and is declared either as *Checkonly*, meaning that the relation is only checked, or *Enforced*, meaning that the model is modified if necessary to satisfy the relation. It is augmented by a *when* clause that specifies under what conditions the relation must hold, and a *where* clause that specifies a condition that must be satisfied by all the model elements participating in the relation.

Our approach provides a solution similar to the *Checkonly* mode of QVT relations. The main difference is our use of *pivot nodes* to define correspondence conditions and the use of cross links. This allows us to use a look up table to match corresponding nodes. Our approach takes advantage of the transformation framework to provide a pragmatic and usable verification technique that can ensure that there are no critical errors in model instances produced by automated transformations.

Triple Graph Grammars

Triple Graph Grammars [66] can be used to transformations on models as the evolution of a graph by applying graph rules. The evolving graph must comply with a graph schema at all times. This graph schema consists of three parts, one describing the source meta-model, one describing the target meta-model, and one describing a correspondence meta-model which keeps track of correspondences between the other two meta-models. Triple graph grammar rules are declarative, and operational graph grammar rules must be derived from them.

The correspondence meta-model can be used to perform a function similar to the cross links used here. This provides a framework in which a map of corresponding nodes in the instance models can be maintained, and on which the correspondence conditions can be checked. This makes it suitable for our verification approach to be applied.

Other Verification Approaches

Some ideas on validating model transformations are presented in [70] and [71]. In [71], the authors present a concept of rule-based model transformations with control conditions, and provide a set of criteria to ensure termination and confluence. In [70], Küster focuses on the syntactic correctness of rule-based model transformations. This validates whether the source and target parts of the transformation rule are syntactically correct with respect to the abstract syntax of the source and target languages. These approaches are concerned with the functional behavior and syntactic correctness of the model transformation. We focus on the semantic correctness of model transformations, addressing errors introduced due to loss or mis-representation of information during a transformation. It is possible for a transformation to execute completely and produce an output model that satisfies all syntactic rules, but which

may still not have accomplished the desired result of porting essential information from the source model to the output model. Our approach is directed at preventing such semantic errors.

Ehrig et. al. [74] study bidirectional transformations as a technique for preserving information across model transformations. They use triple graph grammars to define bi-directional model transformations, which can be inverted without specifying a new transformation. Our approach offers a more relaxed framework, which will allow some loss of information (such as by abstraction), and concentrates on the crucial properties of interest. We also feel that our approach is better suited for transformations involving multiple models and attribute manipulations.

Conclusions and Future Work

In this paper we have introduced an approach for the instance-based verification of model transformations based on the concept of structural correspondence. We have illustrated its use on examples and described informally how the structural correspondence could be specified in a language. We conjecture that the technique is viable and pragmatic for a wide range of transformations, where correctness can indeed be captured in such structural form, but the wider applicability of the approach needs to be verified on industry-grade examples.

Our approach is based on the assumptions that (1) it is relatively easy to specify the correspondence criteria for a transformation, and (2) the correspondence rules are 'complete' in the sense that they cover all the relevant semantic aspects of the transformation. As these properties are always domain- and transformation-specific, it is hard to quantify what it costs to satisfy them. However, we believe it is feasible

to integrate such checks into practical transformation applications and, if the specifications are indeed independently developed, they could serve a solid foundation for verifying the results of transformations.

The most important future work in the approach is the full implementation of the correspondence language and the model checker that evaluates the correspondence expressions. Since the correspondence specification does not make any assumptions about the transformation algorithm, it will be possible to plug it into any implementation of the transformation. Checking each execution of the transformation makes the verification tractable, but care must be taken to not create a significant overhead. However, the guarantee provided about the correctness of the output model can prove to be extremely useful in critical scenarios.

The approach described here will suit a majority of *exogenous* transformations [36] where a structural view can be taken to verify correctness, it may also prove to be useful in *endogenous* transformations (where the source and target models belong to the same meta model) such as optimization or refactoring. Cross links are directed links, and can be created within the same meta. Thus they can be used to specify structural correspondence within the same meta-model. Further research is needed to study the applicability and issues (such as possibility of cycles) of using this technique for verifying such transformations .

CHAPTER IX

CONCLUSIONS AND FUTURE WORK

The semantic correctness of model transformations is a critical factor in the success of a model based software development effort. This important area has long been overlooked in favor of ease of implementation of the transformation specification. This has adversely affected the use of automated model transformations in safety critical applications. The main contribution of this dissertation is to introduce tractability and a sense of practical usefulness to the problem of semantic verification of model transformations. This could possibly lead to greater acceptance of automated model transformations in safety critical applications. We now look at some concluding remarks on the techniques described in this dissertation, and some areas for future work.

Concluding Remarks

Model transformers are similar to compilers, in the sense that they take an input model and produce an output model based on a set of transformation rules. Therefore, the techniques used in compiler verification may also be applicable in the verification of model transformations. Compiler verification is in general concerned with safety criteria, such as initialization of variables and array bounds safety. In the verification of model transformations, we are often more interested in the semantic correctness of the models. This dissertation has illustrated practical and useful techniques to verify model transformations, based on two ideas: (1) The verification problem can be simplified by verifying each execution of a transformation instead of providing a general

correctness proof, (2) We can take advantage of the restricted syntax semantics of domain specific languages to specify correctness in terms of preservation of behavior or correspondence of structure.

Instance Based Verification

This dissertation has introduced instance based verification of model transformations as a practical alternative to correctness proofs. The main idea behind instance based verification is the generation of a certificate of correctness along with each output instance produced by a model transformation. While each execution must be independently verified and certified, the framework for generating the certification needs to be specified only once.

Instance based verification of model transformations consists of two steps:

1. Extending the transformation specification to generate annotations that will simplify the verification of the correctness properties of the instances. This needs to be done only once. The annotations store information about the transformation, such as relations between elements of the source and target instance models, typically in some tabular form. From the experience of the case studies described in this dissertation, cross-links in the GReAT transformation framework provide an excellent method of storing these annotations. Having these annotations on the instance models simplifies the problem verifying of correctness properties, from the difficult problem of theorem proving to the less difficult problem of exhaustive model checking.

2. Once the transformation has been extended, the annotations are automatically generated for each execution of the transformation. These annotations are passed to an automated checker which uses this information to verify the

correctness of the instance based on the property being verified. In the case of verifying behavior preservation, a bisimulation checker is used.

The main challenge in this kind of a semantic verification are the correctness of the correctness specification itself. This is similar to verification in most cases, where unless the property being verified is specified correctly, its verification is useless. However, I believe that concentrating on instances, and using cross-links to store annotations simplifies the specification of the correctness properties. Under these circumstances, there is less chance of an error being introduced into the correctness specification.

The second concern is the overhead introduced by creating annotations and checking correctness for each execution of the transformation. This overhead is mitigated greatly by the use of cross links, which reduce the look up times to retrieve the annotation information. By choosing a good algorithm to check the correctness properties using the annotations, the overhead can be kept low. From initial experiences, the certification framework does not seem to add a significant overhead compared to the execution of the transformation itself.

Behavioral and Structural Verification

The aim of this dissertation is to cover the verification of a wide class of transformations. The instance based approach can be applied to a number of cases, as illustrated by the case studies in this dissertation. The verification problem can be separated into two categories, behavioral and structural verification.

In the verification of preservation of behavioral properties across a transformation, bisimulation or weak bisimulation are suitable for most cases. It is often useful to have an underlying behavior model, which can be used with the instance based certification framework. In cases where the semantic correctness is not entirely concerned with

behavioral properties, structural correspondence can be used to specify the semantic correctness. In the most general case, it must be possible to specify the preservation of behavioral properties also by structural correspondence.

Trusted Components

In any verification framework, there are certain components that must be *trusted*. No verification is performed on these components, and their result is accepted as correct. In the instance based certification framework, the specification of the correctness properties, and the model checker or bisimulation checker that checks the correctness on the instance models, must be trusted. Errors in these components will produce in false results on the correctness of the model transformation.

However, the simplified framework in which the correctness is specified and checked allows us to have more faith in these components. These components are, in general, simple and straightforward enough to be trusted without the need for a formal verification to be performed on them.

Future Work

Model based software engineering is a developing area, with new technologies and formalisms emerging constantly. The diverse nature of the various domain specific technologies presents interesting possibilities. The semantic correctness of model transformations remains a crucial issue, especially with the greater acceptance of model based development in safety critical applications. This dissertation has presented some initial ideas towards a practical approach to address the verification problem, and also opened the doors to many avenues for future research.

The techniques presented here for a framework for the instance based verification of model transformations offers a strong foundation to address the verification of model transformations. The main constituents of this framework are the specification of the correctness properties, the generation of annotations to assist verification, and the evaluation of the correctness rules on the instances. Improvements in the expressibility, automation and efficiency of evaluation of these will aid the acceptance of the verification techniques described here.

Correctness Specification

As noted earlier, the correctness specification is a trusted component - it is assumed that the user has specified the correctness properties correctly, before they can be verified on the instances. It is therefore important that the correctness properties can be specified easily and clearly, with little chance of error in their specification.

An underlying behavior model is very useful in specifying behavioral properties of domain specific languages. This has been illustrated by the use of semantic anchoring in the case studies in this dissertation. An underlying FSM behavior model was used to specify reachability related behavioral properties. Similar techniques in specifying

other more complex properties will make their verification easier. Future research in the area of semantic anchoring will also help the instance based certification approach.

In the absence of a behavior model, the correctness is specified using structural correspondence rules. In this case, the simplicity and power of the query language plays a significant role. The main requirements of a query language for this purpose have already been discussed. The actual set of features appropriate for this requirement can only be accumulated over a very large number of actual transformation cases. Extensive application of the structural correspondence technique to a large number of transformations will provide more insight into the optimum query language for this purpose.

Annotation Generation

The annotations generated during a transformation greatly simplify the final task of verifying the correctness properties on the instance models. In the case studies described in this dissertation, the cross-links feature of GReAT was used to store the annotation information.

The cross-links were specified between the meta-models of the source and target languages, and the transformation rules were extended to create the cross-links in the instances. This second step of extending the transformation is a tedious process, were each appropriate transformation rule must be manually extended to create the cross-links. Though this must be done only once for each correctness specification, the automation of this step could simplify the verification process. Extending the transformation rules automatically based on cross link information specified at the meta-level is an important avenue for future research.

Evaluation of Correctness Properties

A final step of model checking on the instance models is required to verify the correctness properties on the instances. This step is necessary for each execution of the transformation. While this check is automated, it adds a small overhead to each execution of the transformation. Having an efficient implementation to perform this check can decrease the overhead and improve the efficiency of the verification process.

The model checker may have to perform different tasks, depending on the properties being verified, such as checking bisimulation or verifying certain rules specified using some queries. For the case studies presented here, custom hand-coded model checkers were used. Future work may concentrate on consolidated model checkers that can check for a number of different correctness properties efficiently.

Other Future Work

The verification of semantic correctness is a crucial part of the verification of model transformations. Other interesting properties about the model transformation framework are also candidates for a formal analysis. These include proof of termination, proof of confluence etc. These specific properties have not been formally proven for the GReAT transformation framework, and are candidates for future research.

BIBLIOGRAPHY

[1] OMG's Model Driven Architecture Homepage. http://www.omg.org/mda/

[2] Karsai, G. Sztipanovits, J. Ledeczi, A. Bapty, T., "Model-integrated development of embedded software", In: Proceedings of the IEEE, Jan 2003, Vol. 91, Issue 1, pp 145- 164

[3] Ledeczi A., Maroti M., Karsai G., Nordstrom G., "Metaprogrammable Toolkit for Model-Integrated Computing", Engineering of Computer Based Systems (ECBS) , pp. 311-317, Nashville, TN, March, 1999.

[4] Matlab, Simulink and Stateflow tools: www.mathworks.com

[5] *Model of Computation* at The Dictionary of Algorithms and Data Structures, National Institute of Standards and Technology. http://www.nist.gov/dads/HTML/modelOfComputation.html

[6] *Finite State Machines* at The Dictionary of Algorithms and Data Structures, National Institute of Standards and Technology. http://www.nist.gov/dads/HTML/finiteStateMachine.html

[7] K. L. McMillan, "Symbolic Model Checking: an approach to the state explosion problem", CMU Tech Rpt. CMU-CS- 92-131.

[8] Gerard J. Holzmann "The Spin Model Checker: Primer and Reference Manual" Addison-Wesley, ISBN 0-321-22862-6.

[9] L. A. Cortes, P. Eles, and Z. Peng, "A Survey on Hardware/Software Codesign Representation Models", SAVE Project Report, Dept. of Computer and Information Science, Linkping University, Sweden, June 1999.

[10] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming, 8(3):231-274, 1987.

[11] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: A working environment for the development of complex reactive systems", IEEE Transactions on Software Engineering, 16(4):403–414, April 1990.

[12] D. Harel, A. Naamad, "The STATEMATE semantics of statecharts", ACM Transactions on Software Engineering and Methodology (TOSEM), v.5 n.4, p.293-333, Oct. 1996

[13] M. von der Beeck, "A Comparison of Statecharts Variants", In ProCoS: Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems. Pp.128148. Springer-Verlag, London, UK, 1994.

[14] Mikk E., Lakhnech Y., and Siegel M., "Hierarchical automata as model for statecharts", In R. Shyamasundar and K. Euda, editors, ASIAN97 Third Asian Computing Conference. Advances in Computer Science, volume 1345 of *LNCS*, pages 181196. Springer-Verlag, 1997.

[15] Latella D., Majzik I., and Massink M., "Automatic verification of a behavioral subset of UML statechart diagrams using the SPIN model-checker", *Formal Aspects of Computing*, 11(6), pp. 637 664, 1999.

[16] Holzmann G., "The model checker SPIN", *IEEE Transactions on Software Engineering*, 23(5), pp. 279-295, 1997.

[17] "Technical Guide to Model Driven Architecture: The MDA Guide v1.0.1", Adopted by OMG's Architecture Board as an official description of the MDA, June, 2003.
`http://www.omg.org/cgi-bin/doc?omg/03-06-01`

[18] J. Rumbaugh, I. Jacobson, and G. Booch, "The Unified Modeling Language Reference Manual", Addison-Wesley, 1998.

[19] T. Quatrani, "Visual Modeling with Rational Rose 2000 and UML", Addison-Wesley,

[20] Visio Home Page, Microsoft Office Online. `www.microsoft.com/office/visio/`

[21] I. Paltor and J. Lilius. "Formalising UML state machines for model checking". In R. France et al., editor, UML '99, volume 1723 of LNCS. Springer.

[22] M. Gogolla, P. Ziemann, and S. Kushke. "Towards an Integrated Graph Based Semantics for UML". In Electronic Notes in Theoretical Computer Science, volume 72. Elsevier, 2003.

[23] J.Sztipanovits, G.Karsai: "Model-Integrated Computing", Computer, Apr.1997, pp110-112

[24] Model-Integrated Computing at ISIS, Vanderbilt University.
`http://www.isis.vanderbilt.edu/research/research.html`

[25] Agrawal A., Karsai G., Shi F., "Interpreter Writing using Graph Transformations", Technical Report, ISIS-03-401, 2003.

[26] OMG's Meta-Object Facility Homepage. `http://www.omg.org/mof/`

[27] "Request For Proposal: MOF 2.0 Query/Views/Transformations", OMG Document: ad/2002-04-10, 2002, OMG, Needham, MA.

[28] "A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard", OMG Document: ad/03-08-02, 2003, OMG.
www.omg.org/docs/ad/03-08-02.pdf

[29] "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification", OMG Document: ptc/05-11-01, OMG. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01

[30] K. Czarnecki and S.Helsen, "Classification of Model Tranformation Approaches". In Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, Anaheim, October 2003.

[31] Jamda: The Java Model Driven Architecture 0.2, May 2003.
http://sourceforge.net/projects/jamda/

[32] A. Bakay, "The UDM Framework". http://repository.escherinstitute.org/Plone/tools/suites/mic/udm

[33] A. Gerber, M. Lawley, K. Raymond, J. Steel, A. Wood, "Transformation: The Missing Link of MDA", In A. Corradini, H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.): Graph Transformation: First International Conference (ICGT 2002), Barcelona, Spain, October 7-12, 2002. Proceedings. LNCS vol. 2505, Springer-Verlag, 2002, pp. 90 - 105

[34] J. Bézivin, G. Dupé, F. Jouault, and J. E. Rougui, "First experiments with the ATL model transformation language: Transforming XSLT into XQuery". In the online proceedings of the OOPSLA03 Workshop on Generative Techniques in the Context of the MDA.
http://www.softmetaware.com/oopsla2003/mda-workshop.html

[35] H. Ehrig, K. Ehrig, U. Prange and G. Taentzer, "Fundamentals of Algebraic Graph Transformation", Springer-Verlag New York, Inc., Secaucus, NJ, 2006.

[36] T.Mens, P.Van Gorp: A Taxonomy of Model Transformation, International Workshop on Graph and Model Transformation (GraMoT), Tallinn, Estonia, September 28, 2005.

[37] Karsai G., Agrawal A., Shi F., "On the Use of Graph Transformations for the Formal Specification of Model Interpreters", Journal of Universal Computer Science, Volume 9, Issue 11, pp. 1296-1321, November, 2003.

[38] Corradini, A., Montanari, U., and Rossi, F., "Graph processes", *Fundamenta Informaticae* 26, 3-4 (Jun. 1996), 241-265.

[39] Agrawal A., "A Formal Graph-Transformation Based Language for Model-to-Model Transformations", PhD Dissertation, Vanderbilt University, Dept of EECS, August, 2004.

[40] The AGG Homepage. `http://tfs.cs.tu-berlin.de/agg/`

[41] G. Taentzer. AGG: A Tool Environment for Algebraic Graph Transformation. In LNCS 1779, pages 481-490, Springer, 2000.

[42] PROGRES - A Graph Grammar Programming Environment. `http://www-i3.informatik.rwth-aachen.de/tikiwiki/tiki-index.php?page=Research%3A+Progres`

[43] A. Schürr, "PROGRES for Beginners".

[44] Gogolla, M. and Parisi-Presicce, F., 1998, "State Diagrams in UML - A Formal Semantics using Graph Transformation", Proceedings ICSE'98 Workshop on Precise Semantics of Modeling Techniques (PSMT'98), Broy, M., Coleman, D., Maibaum, T., and Rumpe, B., Eds., Technical University of Munich, Technical Report TUM-I9803, pp. 55-72.

[45] A. Narayanan and G. Karsai. Towards Verifying Model Transformations. In R.Bruni and D. Varro, editors, Graph Transformation and Visual Modeling Techniques GT-VMT 2006, Electronic Notes in Theoretical Computer Science, pages185194, 2006.

[46] Kai Chen, Janos Sztipanovits, Sherif Abdelwahed, and Ethan K. Jackson. Semantic anchoring with model transformations. In ECMDA-FA, pages 115129, 2005.

[47] The abstract state machine language. `http://www.research.microsoft.com/fse/asml`

[48] Varró D., "Automated Formal Verification of Visual Modeling Languages by Model Checking", *Journal of Software and Systems Modeling*, col. 3(2), pp. 85-113.

[49] A. Schmidt and D. Varro. CheckVML: A tool for model checking visual modeling languages. In Proc. UML 2003: 6th Intern. Conf. on the Unified Modeling Language.

[50] Hoare, C. A. 1969. An axiomatic basis for computer programming. Commun. ACM 12, 10 (Oct. 1969), 576-580.

[51] M. Fitting, First-Order Logic and Automated Theorem Proving. Springer, 2nd edition, 1996.

[52] Denney E., FischerB., "Certifiable Program Generation ", GPCE 2005, LNCS, vol. 3676, pp. 17-28.

[53] Necula, G. C. 1997. "Proof-carrying code". In Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Paris, France, January 15 - 17, 1997). POPL '97. ACM Press, New York, NY, 106-119.

[54] Pitts, A. M. 2002. Operational Semantics and Program Equivalence. In Applied Semantics, international Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures (September 09 - 15, 2000). G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, Eds. Lecture Notes In Computer Science, vol. 2395. Springer-Verlag, London, 378-412.

[55] Plotkin, G. 1981. A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Computer Science Dept., Aarhus University, Aarhus, Denmark.

[56] D. Sangiorgi. Bisimulation: From the origins to today. In LICS 04: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, pages 298302, Washington, DC, USA, 2004.

[57] Ldeczi A. et. al., "Composing Domain-Specific Design Environments", *IEEE Computer*, November 2001, pp. 44-51.

[58] Agrawal A., Karsai G., Ledeczi A., "An End-to-End Domain-Driven Software Development Framework", 18th Annual ACM SIGPLAN Conference on Object- Oriented Programming, Systems, Languages, and Applications (OOPSLA), Anaheim, California, October 26, 2003.

[59] Gottler H., "Attributed graph grammars for graphics", H. Ehrig, M. Nagl, and G. Rosenberg, editors, Graph Grammars and their Application to Computer Science, LNCS 153, pages 130-142, Springer-Verlag, 1982.

[60] Nickel U., Niere J., and Zündorf A.. Tool demonstration: The FUJABA environment. In The 22nd International Conference on Software Engineering (ICSE). ACM Press, Limerick, Ireland, 2000.

[61] Varró D. "A Formal Semantics of UML Statecharts by Model Transition Systems", Proc. ICGT 2002: 1st International Conference on Graph Transformation, *LNCS*, vol. 2505, pp. 378-392.

[62] Schürr A., Winter A. J., and Zündorf A., In [20], chap. The PROGRES Approach: Language and Environment, pp. 487 550. World Scientific, 1999.

[63] Aditya Agrawal, Gyula Simon, and Gabor Karsai. Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. *Electr. Notes Theor. Comput. Sci.*, 109:43–56, 2004.

[64] W. Harwood, F. Moller, and A. Setzer. Weak bisimulation approximants. In *CSL'06: The 15th International Conference on Computer Science Logic*, Szeged, Hungary, 2006. Lecture Notes in Computer Science.

[65] Andrea Corradini, Reiko Heckel, and Ugo Montanari. Graphical operational semantics. In *ICALP Satellite Workshops*, pages 411–418, 2000.

[66] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *WG '94: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*. Pp. 151–163. Springer-Verlag, London, UK, 1995.

[67] D. Bisztray, K. Ehrig, R. Heckel. Case Study: UML to CSP Transformation. In *Applications of Graph Transformation with Industrial Relevance (AGTIVE)*. 2007.

[68] OMG. Unified Modeling Language, version 2.1.1. 2006. `"http://www.omg.org/technology/documents/formal/uml.htm"`

[69] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM* 21(8):666–677, 1978.

[70] J. M. Küster. Systematic Validation of Model Transformations. In *Proceedings 3rd UML Workshop in Software Model Engineering (WiSME 2004)*. October 2004.

[71] J. M. Küster, R. Heckel, G. Engels. Defining and validating transformations of UML models. In *HCC '03: Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments*. Pp. 145–152. IEEE Computer Society, Washington, DC, USA, 2003.

[72] de Lara, J. and Taentzer, G. Automated Model Transformation and its Validation with AToM3 and AGG. In *Lecture Notes in Artificial Intelligence, 2980*. Pp. 182–198. Springer.

[73] D. Bisztray, R. Heckel. Rule-Level Verification of Business Process Transformations using CSP. In *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'07)*. 2007.

[74] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, G. Taentzer. Information Preserving Bidirectional Model Transformations. In *Fundamental Approaches to Software Engineering*. Pp. 72–86. 2007.

[75] D. Varró, A. Pataricza. Automated Formal Verification of Model Transformations. In Jürjens et al. (eds.), *CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop*. Technical Report TUM-I0323, pp. 63–78. Technische Universität München, September 2003.

[76] L. Lengyel, T. Levendovszky, G. Mezei, H. Charaf. Model-Based Development with Strictly Controlled Model Transformation. In *The 2nd International Workshop on Model-Driven Enterprise Information Systems, MDEIS 2006*. Pp. 39–48. Paphos, Cyprus, May 2006.

[77] H. Giese, S. Glesner, J. Leitner, W. Schfer, R. Wagner. Towards Verified Model Transformations. October 2006.

[78] A. Narayanan and G. Karsai. Verifying Model Transformations by Structural Correspondence. In *7th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008)*, March 2008.

[79] A. Narayanan and G. Karsai. Using semantic anchoring to verify behavior preservation in graph transformations. *Electronic Communications of the EASST*, 4(2006), January 2006.

[80] J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt. Model Transformations in Practice Workshop Announcement, MoDELS 2005, 2005. `http://sosym.dcs.kcl.ac.uk/events/mtip05/long_cfp.pdf`.

[81] K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, G. Taentzer, D. Varró, and S. Varró-Gyapay. Model transformation by graph transformation: A comparative study. In *MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005)*, 2005.