DOMAIN-SPECIFIC MODELS, MODEL ANALYSIS, MODEL TRANSFORMATION

By

Tivadar Szemethy

Dissertation
Submitted to the Faculty of the
Graduate School of Vanderbilt University
in partial fulfillment of the requirements
of the degree of

DOCTOR OF PHILOSOPHY

in

Electrical Engineering

August, 2006

Nashville, Tennessee

Approved:

Professor Gabor Karsai

Professor Janos Sztipanovits

Professor Gautam Biswas

Professor John T. Koo

Professor Sherif Abdelwahed

ACKNOWLEDGEMENTS

Undertaking this research with ISIS has been a great learning experience. First and foremost, I'd like to thank the ISIS community for giving all what it takes to earn a PhD: motivation, support, the always necessary criticism, and most importantly for never stopping to ask questions.

Of the ISIS community, I'd like to thank my advisor, Dr. Gabor Karsai. His insight and dedication are unparalleled, so are his mentoring and guidance. He is able to unify science and engineering in a way only a select few can.

I am also grateful to my committee members. Thank you, John and Sherif for always answering my questions, and for always having time for me. Janos Sztipanovits and Gautam Biswas also gave me all the help and support I could ask for. I could not have wished for a better PhD committe.

I owe special thanks to Nag Mahadevan for the many inspiring conversations we had, and for always asking the right question. I would also like to thank all my friends at ISIS I had the opportunity to work with.

The NSF ITR on "Foundations on Hybrid and Embedded Software Systems" has supported, in part, the activities described in this paper.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS i

LIST OF FIGURES v

LIST OF ABBREVIATIONS viii

Chapter

I. INTRODUCTION 1
- Platform Modeling 6
- Model Transformations 6
- Outline 8

II. BACKGROUND: FORMAL VERIFICATION 9
- Modeling 10
  - Modeling approaches 11
  - Kripke structures and first-order logic representations 11
  - Composing Kripke structures 13
  - Fairness 15
  - Discrete untimed models 15
  - Discrete models with sparse (quantized) time 16
  - Timed Automata mapped onto clock regions 16
  - Symbolic state space representation in finite-state systems 18
  - Tool Example: The UPPAAL Model Checker 20
- Specification 22
  - Computational Tree Logic CTL* 22
  - CTL and LTL 24
- Verification 26
  - Theorem Proving 26
  - Model Checking for finite-state systems 27
- Conclusions 31
  - Modeling 31
  - Specification 32
  - Verification and Model Checking 32

III. BACKGROUND: MODEL TRANSFORMATIONS 34
- Basic concepts 34
  - Terminology 35
  - Model Transformation Approaches 37
  - Properties of Transformations 39

The Unified Modeling Language with OCL . . . 40
UML Class Diagrams . . . 41
The OCL language . . . 42
Conclusions . . . 46
Graph Transformations . . . 46
The Generic Modeling Environment (GME) . . . 51
The GME Metamodeling Environment . . . 52
The GReAT Graph Rewriting and Transformation System . . . 53
Graph Transformation Language . . . 55

IV. IMPLICIT PLATFORM MODELING: CASE STUDY . . . 59
The Platform concept . . . 60
Design-time model analysis . . . 62
The SMOLES design and synthesis environment in GME . . . 63
The SMOLES Language . . . 63
The DFK platform . . . 66
Implementing SMOLES over DFK . . . 68
Modeling and synthesis environment . . . 70
Mapping SMOLES models onto UPPAAL Timed Automata . . . 71
Modeling component-kernel interaction . . . 74
Component TA models . . . 76
Analysis model for the Kernel . . . 79
Implementing the transformation with GReAT . . . 81
Metamodeling UPPAAL . . . 82
Transformation setup and phases . . . 83
Flattening the assembly hierarchy . . . 85
Creating the TA templates . . . 88
Verification via model checking Timed Automata . . . 90
Conclusions . . . 92

V. EXPLICIT PLATFORM MODELING . . . 95
The DSML→Platform→Analysis transformation chain . . . 95
The DSML → Platform mapping . . . 96
The Platform→Analysis mapping . . . 98
Platform metamodeling for synthesis . . . 100
Example: The SMOLES → DFK transformation . . . 103
The Platform → Analysis transformation . . . 106
The Platform Modeling Language (PML) . . . 109
Metamodels, Crosslinks and the Kernel skeleton . . . 110
Component Skeletons . . . 111
Mappings . . . 113
PML block hierarchy . . . 115
Mappings semantics . . . 117
Execution semantics for actions . . . 118
A detailed example: mapping triggers for IF . . . 118

Comparing the DFK → UPPAAL mapping in PML and GReAT . . 122
Implementing PML over GReAT 123
Extending the transformation configuration 125
Generating rules for component skeleton instantiation 126
Implementing PML Mappings in GReAT 127
The compiler transformation 130
Conclusions 136

VI. RESULTS AND FUTURE WORK 138
Platform Modeling 138
Explicit platform models 139
Graph Transformation 140
Future Work 141
Platform Modeling 141
Graph Transformation 142

Appendix

A. THE GME MODELING FRAMEWORK FOR PML 144
The modeling language (metamodel) 144
UML fundamentals 144
PML concepts 145
Pattern hierarchy 146
Pattern basics 146
Auxiliary software in the PML framework 147
The ImportMeta tool 147
Decorator 147

B. DETAILED PML EXAMPLES FROM THE DFK → UPPAAL MAPPING 148
Kernel Skeleton 148
Component skeletons 149
Filter and action patterns 151

C. DETAILED EXAMPLES FOR THE PML → GREAT TRANSFORMATION 153
Component Skeleton instantiation examples 153
Examples for the SelectAction rule-block 157

REFERENCES 160

LIST OF FIGURES

Figure Page

1. Example of a Kripke structure 12
2. Truth table, full and reduced decision tree for a boolean function 18
3. OBDDs for function $f = (x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_3 \wedge y_3)$ with different ordering 20
4. UPPAAL model for the mutex example 21
5. Basic CTL Operators 25
6. UML Class representation and relations 43
7. Graph Concepts: L is a subgraph of G and has an occurrence in H 47
8. Illustration of a graph transform step $G \Rightarrow H$ 49
9. Four-layer modeling architecture with GME 53
10. Example GReAT rule 56
11. Component-based system implemented over a platform 60
12. Sample SMOLES systems in GME 65
13. Parts of the SMOLES metamodel in GME 71
14. TA Templates for platform modeling 76
15. SMOLES model and Timed Automata for component *Processing* 78
16. Kernel TA model for the system in Fig 12(a) 81
17. The UPPAAL metamodel in GME 83
18. Top-level blocks of the SMOLES → UPPAAL transformation 85
19. Rule blocks flattening the SMOLES hierarchy 87
20. Rule blocks creating UPPAAL TA 88
21. Creating the Kernel TA 90

22. Advantages of having an explicit platform abstraction in MIC 98
23. The DFK metamodel in the case study 102
24. Illustrations for the SMOLES → DFK transformation: Queue and Timer . 103
25. SMOLES → DFK trigger mapping 105
26. Creating an analysis model fragment in GReAT 107
27. Visualization of the ComponentModel concept 112
28. PML Block hierarchy to map Nodes and Methods 116
29. The simplified IF metamodel 118
30. Trigger condition $(a \vee b \vee c) \wedge d$ in DFK 119
31. Trigger expression $((a \vee b) \vee c) \wedge d$ in IF 120
32. Progress of the trigger mapping process 121
33. Block mapping a Port trigger source (steps 2-3)) 121
34. Block mapping an upstream trigger onto an Expr (steps 5-6) 122
35. Mapping a subsequent trigger argument onto an ExprR (steps 7-8) 122
36. High-level overview of the PML → GReAT transformation 123
37. Configuration of a generated GReAT transformation 125
38. Schematics of the rules implementing component skeleton instantiation . . 126
39. Schematics of PML Mappings implementation 128
40. Mapping a zero-cardinality pattern 129
41. Major phases of the compiler transformation 131
42. AssociateClasses rule-block from the compiler 132
43. MapCrosslinks rule-block in the compiler 133
44. Rule-block Components with sub-block InputSkeleton 134
45. Rule-block BuildRules 135
46. Rule-block CopyPattern within MapMappings 136

47. PML concepts in the metamodel 145

48. Hierachy of patterns in the PML metamodel 145

49. Patterns metamodel for PML 146

50. The DFK Kernel Skeleton for UPPAAL in GME 148

51. Component Skeleton for a Node in PML 149

52. Component Skeleton part for a PID constant in PML 151

53. Dataflow→variable PML mapping block 151

54. Component skeleton instantiation overview 154

55. Generated rule-blocks for the PID skeleton 156

56. Example for GReAT rules generated for PML mapping patterns 157

57. Details of GReAT Test cases and rules 158

LIST OF ABBREVIATIONS

(G/M)T	(Graph/Model) Transformation
(L/R)HS	(Left/Right) Hand Side (of an expression/rule)
(O)BDD	(Ordered) Binary Decision Diagram [17]
API	Application Program Interface
CBS	Computer-Based System
CSP	Communicating Sequential Processes
DFK	DataFlow Kernel [51]
FSM	Finite State Machine
GME	Generic Modeling Environment [37]
GReAT	Graph Rewriting And Transformations [30]
KS	Kripke structure(s) [23]
MBD	Model-Based Design
MD(A/D)	Model Driven (Architecture/Design) [1]
MIC	Model Integrated Computing [54]
OCL	Object Constraint Language [57]
OS	Operating System
QVT	Query / View / Transformation [43]
SMOLES	Small MOdeling Language for Embedded Systems [53]
TA	Timed Automata [11]
UDM	Universal Data Model [14]
UML	Unified Modeling Language [2]

CHAPTER I

INTRODUCTION

Modeling is a traditional engineering practice for mitigating complexity. Models are mathematically precise *abstractions* of the system, focusing on *relevant* aspects. According to the basic assumption, creating and analyzing abstractions is easier than building and studying the systems themselves. *Design models* describe the abstraction of the system's *structure*, whereas *analysis models* capture the abstraction of its *behavior*. (The fact that those are both *abstractions* is very important.)

In order to better understand the role of modeling, let us examine the process of creating engineering artifacts in general. The first step is *specification*, consisting of two parts:

Requirements specification: *what* is expected of the system.

Design specification: *how* the system is going to be constructed or operate.

During the first steps, these specifications are abstract and not necessarily precise, as they leave many details unspecified. They capture high-level aspects, and give a sort of "executive summary" about what and how the system is supposed to do. At this level, descriptions are usually given in human language.

Design models are subsequent refinements of the design specification: they assign precise semantics to their building blocks, and enrich the level of detail. Analysis models form the counterparts of design models. At each level of abstraction, they capture the behavior of the structures described by the design model.

Using a blunt example, it is possible to construct a hammer by its design model (physical dimensions and material specifications). On the other hand, for determining the impact the hammer can make (to verify against requirements specifications), we either have to build and test it by experiments, or use an analysis model based on Newtonian physics. Although

the “build and test” method might be feasible for hammers, it is less and less applicable for complex engineering systems. Thus, analysis models become more and more important.

The analysis model is derived from a precise design model. As we can see by the example of the hammer, deriving the analysis model takes a very different knowledge (physics) from what is required for implementing (building) the design (craftmanship). In most cases, generalized analysis models can still be provided, automatically customizable for a specific design (i.e. by substituting in values from the design model). Algorithms for checking these customized analysis models can also be automated. Thus, the design can be validated (against *precisely* captured requirements) with a minimal knowledge about the analysis model (e.g. before commencing to construct the hammer, the craftsman can validate the design by computing a basic formula on values from the design model).

For computer-based systems (CBS) the end product is software (source code and other artifacts, such as configuration or documentation), or a hardware-software combination. Source code is very hard to analyze, and testing the end system is limited by complexity. Thus, methodologies providing *design-time analysis*, such as model-based design bring significant improvements over traditional “build and test” methodologies.

A leading approach of model-based design is *Model-Driven Architecture* (MDA [1]), promoted by the Object Modeling Group (OMG). MDA provides a broad set of guidelines and technologies for specification, design and implementation.

In MDA, system functionality is defined though a platform-independent model (PIM), given in an appropriate specification language. *Platforms* in MDA are specific technologies implementing abstract concepts used by the design. For example, Oracle is a specific database technology implementing the concept of relational database.

A given PIM is translated into platform–specific model(s) (PSMs) for the actual implementation. MDA defines an architecture (modeling standards, guidelines, tools) for structuring specifications expressed as models. (Here, “architecture” does not refer to the architecture of the actual design, but rather to the architecture of enabling stardards, representations

and technologies that form the basis of MDA). The PIM→PSM transformation is usually automated by tools such as code generators. The MDA architecture defines and relates multiple standards, such as the Unified Modeling Language (UML [2]), the Meta-Object Facility (MOF [3]), the XML Metadata interchange (XMI [4]).

MDA provides the architecture for using precise models during design and implementation. Thus, analysis models can be derived, and design-time analysis performed.

Model Integrated Computing (MIC [54]) is also a model-based design approach for CBS (and for embedded systems in particular). In MIC, similar to MDA, a model is more than a mere "blueprint" of the system: MIC elevates the scope and usage of models to form the integrating "backbone" of the development process. Visual, multi-aspect models capture the information relevant to the system being developed. Models explicitly represent the designer's understanding of the system at multiple levels (functional, system architecture and system environment).

Traditionally in CBS design, designers use abstractions of the computer's resources, such as memory address→variable or machine instructions→ assembly language→C language. The solution to a given problem is specified in terms of these abstractions. These are abstractions of the *solution space*, and it takes a computer expert (programmer) to understand each individual problem and code a solution for them. MDA facilitates a high level of automation for this coding task, but the solution is still specified in a (high-level) abstraction of the solution space. A MDA "user" is a programmer aided by high-level, sophisticated tools.

MIC promotes the use of *domain-specific modeling languages* (DSMLs), which provide abstractions of the *problem space* [48]. In MIC, the problem space is captured by *meta-modeling*: a visual modeling language specific to the problem's domain is defined using meta-level tools. The *metamodeler* creates the DSML, and also provides *model interpreters* which map a model given in the particular DSML onto structures in the solution space (e.g. code generators mapping interaction diagrams onto function calls).

Also, MIC facilitates the automatic synthesis of visual, user-friendly model editors based on the presentation rules defined in metamodels. Domain experts, such as control systems engineers can comfortably work in the environment built around DSMLs, and compose design models using domain-specific elements like sensors, actuators, PID controllers etc.

In its implementation, MIC leverages on MDA and uses core MDA approaches and technologies. Instead of PIMs, MIC incorporate designer's knowledge into *domain-specific* (design) *models* (DSMs), and uses automated synthesis tools on those to analyze them and generate the implementation for the system. Proving the correctness of the generator tools ensures the correctness of a family applications generated — assuming that the abstraction captured in the DSML is correct in the first place.

MIC promises better designs by allowing the designer (domain expert) to focus on relevant issues by providing domain-specific modeling (design) environments tailored carefully for the particular area. Advanced modeling tools, such as metamodel-configurable visual editors, constraint checkers etc. can significantly improve the (syntactical) correctness of the resulting designs. As each individual system has its own specific set of requirements, these requirements cannot be included with metamodel "correctness" rules and thus have to be verified for each implementation.

Validating a system against its requirements is a challenging task. For complex CBS (and especially for embedded systems interacting with the physical environment), the traditional validation method of *testing* is generally prohibitively costly (if possible at all) due to the astronomical number of test cases to be verified. On the other hand, many of these systems operates in a high-risk environment, and failure has dire (often unacceptable) consequences.

Thus, *design-time verification* becomes important. Design-time verification methods, both *formal verification* and *simulation* rely on mathematically precise *analysis models*, with well-defined semantics. High-level design models usually adhere to a formal *Model of Computation* (MoC [38]). MoC precisely defines components and their interactions within a model. Since these languages are formal, it is usually possible to perform some level of

formal verification (mathematical analysis) on the design. Such examples include, analyzing application-level event chains in event-triggered systems or checking token emission / consumption balance at the actor level in a dataflow language.

These high-level languages, though formal, still abstract away many "implementation details". Typically, they assume concurrent components / threads executing "truly" in parallel. They might assume zero-delay communication or instantaneous events at certain parts of the system. They also might neglect certain inherent resource requirements, such as OS overhead or scheduling policy. These assumptions are made in order to reduce design complexity, and to let the designer focus on the "relevant" aspects. This is perfectly understandable as domain experts should not be burdened with implementation-level details unless those have system-wide consequences.

For verification, these details must not be ignored. Quite often, these properties get overlooked during the design and lead to hard-to-identify problems such as priority inversion or internal OS resource starvation. Thus, these properties have to be accurately captured in the analysis model.

A "true" analysis model should model all details of the system. Hence, the behavior predicted by the analysis model is equivalent to the system's "true" (physical) behavior. Unfortunately verifying or simulating this analysis model becomes equivalent to testing the implementation (end system), which is not feasible. For example, embedded systems consist of software, hardware and often mechanical parts (mechatronics). Analyzing the behavior of a system of this kind requires the analysis of a *hybrid system* (a mathematical abstraction exhibiting both continuous and discrete behavior). In general, hybrid systems analysis is an NP-complete problem, and the solution is known only for certain subsets of hybrid systems.

Thus, providing design-time analysis models, <u>at the right level of abstraction</u> is a crucial problem.

Platform Modeling

The solution proposed in this work is *platform modeling*: modeling the *implementation platform*, and the design implemented by this platform, at an appropriate level of abstraction. This level should be high enough to make formal verification or simulation feasible, and it should be low enough to capture key properties of the platform. In this work, the word "platform" is used in this sense: it is defined as the *collection of hardware, software or middleware services / resources the system is implemented over and by.* The platform is always the appropriate layer below the DSML used for the design, and its position in the layer hierarchy is relative to the DSML. For an applications programmer, it can be the collection of OS services and the `libc`; for an Internet service designer it can be the TCP/IP protocol; for a signal processing engineer it can be the dataflow architecture used to implement the design. Platform-based design [47] uses the platform concept in a very similar role.

The *platform-level analysis model* captures

1. how the MoC is implemented by platform-level structures
2. the particular configuration / composition of these platform-level structures forming the implementation of the particular design in question

An *explicit platform model* is a concise representation of mapping platform-level structures onto analysis models, and the composition / configuration of these structures to model particular designs.

It needs to be emphasized that the platform-level model is still an abstraction. The validation results obtained by its analysis are only as relevant to the implementation as the platform model is relevant to the platform implementation.

Model Transformations

By its very nature, MDA/MIC relies on model transformations, such as model-to-code transformations to generate source code or model-to-model transformations forming steps

in the process of system synthesis. Model transformations is a very active research area, as developers are seeking higher-level methods to cope with the overwhelming complexity of today's CBS.

Deriving analysis models from design models and platform descriptions is also a model transformation step. Hence, the *correctness of the transformation* is of particular concern. The validity of the analysis results obviously depends on the correctness of the way the analysis model is obtained.

Graph transformation (GT [12]) is promising model transformation approach, since annotated graphs provide a natural abstraction format for many modeling languages. It is also a relatively mature, well-researched and understood area of discrete mathematics, as active research in this area is performed since the '70s. Theoretical results on proving key properties (e.g. termination and confluence) of graph transformation systems promise a higher level of confidence in the correctness of transformation specifications.

The goal of this research is to develop a model transformation framework to automatically derive analysis models using *explicit platform models.* Explicit platform models enable the *design-time* verification of implementations over different platforms.

Platform-level analysis models provide important details about the *implementation* of the design, and enable their verification. The designer is free to choose the appropriate level of abstraction when providing the platform models. This way, fine control is obtained over the level of details represented in the analysis model. Due to the state of the art in formal verification (scalability issues, e.g. state space explosion), this level of control over the complexity of the analysis model is very much necessary, as it makes the difference in the feasibility of a verification. As a consequence, properties which were subject to designer intuition or "robust design" become formally verifiable.

Outline

The second chapter reviews the state of the art in formal verification. Dominant modeling approaches suitable for verification are enumerated along with their mathematical background. The problem of state space explosion is explained and the most popular method for representing large state spaces is discussed (OBDDs). Formal requirement specifications for reactive systems are also discussed (temporal logic), as well as a few important model checking algorithms

The third chapter gives an overview of model transformations. As this are is less mature than formal verification, the chapter starts with the discussion of basic concepts and terminology, followed by an introduction to model transformation approaches. Relevant parts of the UML standard are discussed next. These are central to the modeling approach for the rest of the dissertation. A summary of graph transformations is also presented, and the GME modeling environment and the GReAT graph transformation engine are detailed. These are representative examples for metamodeling and model transformation frameworks used in MDA, and the examples of this work will be prepared in GME / GReAT.

The fourth chapter covers a motivational case study in (implicit) platform modeling: the basic concepts are introduced and the process is demonstrated on a non-trivial example.

In chapter V. the approach of *explicit platform modeling* is discussed in detail. The DSML → Platform → Analysis transformation chain is reviewed. Platform Modeling Language (PML) a new language for explicit platform models is introduced. An implementation for PML is also discussed. Finally, the case study of chapter IV. is revisited, and the advantages of explicit platform modeling are demonstrated.

CHAPTER II

BACKGROUND: FORMAL VERIFICATION

In order to assign semantics to design models for requirement verification, first we need to know *what* can be verified, and *how* the model transformation could be done. To this end, the following two chapters provide overviews of the state-of-the-art in *formal verification* and *model transformation*.

In our context, *formal verification is the act of proving or disproving the correctness of a system with respect to a certain formal specification or property, using formal methods* [1], where *formal* means "mathematically precise".

This chapter focuses on formal verification, where the traditional order is modeling → specification (formalizing requirements in terms of the model / language)→verification. In *design*, the first step is specification (requirements and architecture). Subsequently, design models refine the basic architecture specification. When discussing formal verification in this chapter, it is assumed that this basic specification already exists."Modeling" in this context means formalizing the design model in a language suitable for verification.

Formal verification of an (embedded) system consists of three major tasks:

Modeling Provide an abstract representation of the design, and establish our *understanding* about the system. The model should faithfully represent the decomposition, known properties, behavior, relations to its environment etc. of the design. Models are also used to reduce complexity and aid human understanding. Therefore the above goals are usually achieved by using hierarchical, *multi-aspect* models. Multi-aspect models decompose the system into different "views" (aspects). An aspect groups a subset of associated model elements.

[1] this definition is due to Wikipedia, `http://en.wikipedia.org/wiki/Formal_verification`

Specification Formalize the desirable properties of the system. The specification establishes the goals and requirements the design should satisfy. Requirements are specified in each aspect of the design: physical, functional, temporal etc., and each aspect has its own way of formally specifying those. *Safety* properties are usually formulated as "bounds" on the system's state trajectory. Thus, state reachability analysis is important in verifying safety properties, formulated either using sequences of state changes or state sets (regions) reachable through trajectories.

Verification Check if the system satisfies the specification: Depending on the modeling and specification, many methods can be employed for verification. All of them have to cope with representation of the system's states, and group them into subsets which either satisfy or do not satisfy the specifications. The methods we are going to examine are *formal*, i.e. based on mathematical foundations.

Modeling

During **modeling**, we provide an abstract, simplified view of the system on some level of abstraction. In this context, a *model* is a mathematical abstraction, which explains and/or predicts the behavior of a physical artifact (in our context, this is often a computer executing the software being modeled). A *design model* is a formal specification of the function, structure, and/or behavior of a system. Formal (vs. informal) means that every modeling element has a well-defined, unambiguous meaning associated with it.

It is important to observe that not all models are suitable for formal verification. The discussed methods have strong restrictions on their inputs, and therefore the *verification model* is usually an *aspect* of the comprehensive design model. This aspect or *view* is derived through *model transformation*.

Modeling approaches

In this section, the modeling approaches suitable for formal verification are discussed — it is a different problem to transform a design model represented in a different formalism into one of these.

For *concurrent, reactive* systems, the following modeling approaches are used:

discrete untimed the system has a finite number of discrete states and time is not modeled.

discrete with sparse (quantized) time representation the system has a finite number of discrete states and time is divided into slices indexed by natural numbers, i.e. clocks can be represented by integer variables.

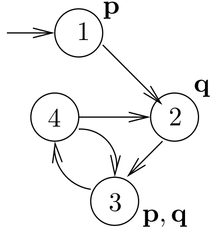
discrete with dense time (Timed Automata) the system has a finite number of discrete states extended with a set of real-valued variables modeling clocks. Time is represented as a continuous (”dense”) quantity, i.e. there are (infinite) time points between any two (different) points in time.

hybrid systems have both discrete and continuous variables. Time is continuous and most models in this area use the subclass *linear hybrid systems.*

In the next secions, an introduction of the mathematical foundations of the modeling approaches discussed will be given, followed by a short introduction to each.

Kripke structures and first-order logic representations

Embedded systems interact with their environment frequently, change their *state* based on these interactions (or internal events), and typically do not terminate. Therefore, they cannot be sufficiently modeled only by their input-output behavior. Typically, we do want to model the system’s (internal) state and its evolution over time, and this timespan is often unbounded.



- $S = \{1, 2, 3, 4\}$
- $S_0 = \{1\}$
- $R = \{(1,2), (2,3), (3,4), (4,2), (4,3)\}$
- $AP = \{p, q, \neg p, \neg q\}$
- $L = \left\{ \begin{array}{ll} (1, \{p, \neg q\}), & (2, \{\neg p, q\}), \\ (3, \{p, q\}), & (4, \{\neg p, \neg q\}) \end{array} \right\}$

Figure 1: Example of a Kripke structure

Such systems are modeled by *Kripke structures* (KS) [23], which describe a class of finite-state automata extended with first-order logic propositions. A state of the automaton is defined by a unique subset of these logic propositions. More precisely a state is a *valuation* which associates each system variable with a value within its domain. The state transition relation then can be defined by (*present states*, *next states*) pairs, denoted as $R(s, s')$. Figure 1 demonstrates a simple Kripke structure. As a further property, since the number of states is finite, a KS also constitutes a *finite transition system*, and can be used in the analysis of *hybrid systems*.

Formally: Let AP be a set of atomic propositions defining possible system conditions. A *Kripke structure* M over AP is a four tuple $M = (S, S_0, R, L)$ where

1. S is a finite set of states.
2. $S_0 \subseteq S$ is the set of initial states.
3. $R \subseteq S \times S$ is a total transition relation
4. $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions which are true in that state.

AP, the global set of atomic propositions is usually understood from the context — if not, it may be included with the definition of the KS e.g. $M = (S, S_0, R, L, AP)$.

A *path* in structure M is defined as an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$.

A state s is *reachable* if there is a path from the initial states to s.

Reactive systems are sometimes described by a first order formula-pair $(\mathcal{S}_0,\ \mathcal{R})$, where $\mathcal{S}_0$ describes the initial states and $\mathcal{R}$ describes the translation relation in the following way:

A *valuation* for $V = \{v_1, \ldots, v_n\}$ is a function which associates each system variable with a value in its domain D. For example, if V is defined as $V = \{v_1, v_2, v_3\}$ and the domain is $\mathcal{N}$, one valuation is $\langle v_1 \leftarrow 5, v_2 \leftarrow 6, v_3 \leftarrow 9\rangle$, then the subset of propositions describing this particular state is $(v_1 = 5) \wedge (v_2 = 6) \wedge (v_3 = 9)$. In this manner a formula can be constructed for $\mathcal{S}_0$ which evaluates *True* iff the system variables evaluate according to the initial conditions.

For transitions, we introduce the formula (V, V') to represent (*current state, next state*) pairs. $\mathcal{R}$ evaluates *True* for every (V, V') pairs representing a valid transition in the system.

The above two representations for reactive systems, a KS or a $(\mathcal{S}_0, \mathcal{R})$ pair are equivalent, and can be derived from each other as shown in [23].

Composing Kripke structures

Complex Kripke structures are usually presented by a structured language, where the global system is given as a composition of smaller modules. In this case AP is usually global and models are given as $M = (S, S_0, R, L)$.

In *synchronous* parallel composition, all FSMs execute a transition at the same time (akin to being driven by the same hardware clock signal). At each clock pulse every component performs a transition. Thus components evolve in parallel.

Formally, the synchronous composition of two Kripke structures is defined as follows: let $K_1 = (S^1, S^1_0, R^1, L^1, AP^1)$ and $K_2 = (S^2, S^2_0, R^2, L^2, AP^2)$ represent two Kripke structures. Their *synchronous composition* $K = (S, S_0, R, L, AP)$ is defined as:

- $S = S_1 \times S_2$
- $S_0 = \{(s^1, s^2) : s^1 \in S^1_0 \text{ and } s_2 \in S^2_0\}$

- $R = \{((s_i^1, s_i^2), (s_{i+1}^1, s_{i+1}^2)) : (s_i^1, s_{i+1}^1) \in R^1 \text{ and } (s_i^2, s_{i+1}^2) \in R^2\}$

- $AP = AP^1 \cup AP^2$

- $L = \{((s^1, s^2), a) : (s^1, a) \in L^1 \text{ or } (s^2, a) \in L^2\}$

In *asynchronous* composition, the member automata's transitions are independent of each other: the system evolves by *interleaving* the evolution of its components. At each execution cycle one component is selected and performs a transition. The interleaving semantics are a simplification for parallel execution: since transitions can follow each other arbitrarily rapidly, parallel execution can be assumed.

Formally, the asynchronous composition of two Kripke structures is defined as follows: let $K_1 = (S^1, S_0^1, R^1, L^1, AP^1)$ and $K_2 = (S^2, S_0^2, R^2, L^2, AP^2)$ two Kripke structures. Their *asynchronous composition* $K = (S, S_0, R, L, AP)$ is defined as:

- $S = S_1 \times S_2$

- $S_0 = \{(s^1, s^2) : s^1 \in S_0^1 \text{ and } s_2 \in S_0^2\}$

- $R = \left\{ ((s_i^1, s_i^2), (s_{i+1}^1, s_{i+1}^2)) : \begin{array}{c} (s_i^1, s_{i+1}^1) \in R^1 \wedge s_i^2 = s_{i+1}^2 \\ \text{or} \\ s_i^1 = s_{i+1}^1 \wedge (s_i^2, s_{i+1}^2) \in R^2 \end{array} \right\}$

- $AP = AP^1 \cup AP^2$

- $L = \{((s^1, s^2), a) : (s^1, a) \in L^1 \text{ or } (s^2, a) \in L^2\}$

This composition results in a larger total state space, since the full cross product of the automata's states need to be considered in the global automaton at each step, while in the synchronous case the state set for the nth transition is the union of each individual automata's states for their nth transition.

Fairness

Asynchronously composed models are usually augmented by *fairness constraints* to prevent situations in which only one automaton executes transitions while the others wait forever. These conditions can enforce a *fair* execution environment by specifying predicates which have to evaluate infinitely often during the infinite execution of the automaton. The most common example is the *running* fairness condition: the automata has to execute transitions infinitely often. In other words, the *running* condition must be true infinitely often.

Formally, a *fair Kripke structure* is a 4-tuple $M = (S, R, L, F)$ where $F \subseteq 2^S$, $F = \{P_1, \ldots, P_k\}$ is a set of fairness constraints (also known as generalized Büchi acceptance conditions).

If $\pi = s_0, s_1, \ldots$ is a path in M, we define $inf(\pi) = \{s | s = s_i \text{ for infinitely many } i\}$. Then π is *fair* iff for every $P_j \in F$, $inf(\pi) \cap P_j \neq \emptyset$.

Discrete untimed models

These models are explicit *finite state machine* specifications, where the system variables are interpreted over a finite domain.

Discrete untimed models can efficiently represent hardware designs and communications protocols. A complex design contains several FSMs, and the total state space can be represented by the *parallel composition* of these FSMs.

With asynchronous composition, the language has to provide a synchronization mechanism. For this, all languages studied have shared (global) variables, and some offer CSP-style (rendezvous channel) synchronization primitives as well.

The state space of the composed global automaton is still finite, but often huge, especially if asynchronous composition is used. As a consequence, exhaustive state-space exploration

is (theoretically) possible, and the model checking (state space search) algorithms are guaranteed to finish. The practical size of the system is limited by the number of states, and effective state space representation has become a major problem in this area.

Discrete models with sparse (quantized) time

A logical extension of the previous modeling language is to associate the state transitions with unform delays in real time as computer hardware works on uniformly timed clock pulses.

There are two possible approaches here, the first one assumes that as in computer hardware, every transition happens on a clock tick. In this manner, we can describe real-time behaviors, and we can create models that states "it takes n time units to reach state A from startup". This approach is taken with the real-time extensions of the NuSMV [8] language.

The other approach, taken by the *simply timed* systems [39], extends Kripke structures into Timed Kripke structures, where transitions carry durations, which can be arbitrary natural numbers (i.e. even zero, which is not possible with the previous approach).

Time representation is very simple in these systems, but even this simple introduction of time makes composition difficult [33], unless all the participating automata are synchronized by a global clock.

It is worth observing here, that these models can have a global counter representing time units only if the system is periodic, or if we are interested in the system's behavior up to a limited amount of time, since all variables of the system have to be finite, including this global clock.

Timed Automata mapped onto clock regions

Finite-state models are useful for verifying a large class of systems, especially — as we've seen — hardware and other systems where transitions are governed by clock ticks, and time is represented as a discrete counter.

In order to model real-time systems more faithfully, representation of continuous time must be supported. One such extension of the FSM is the *Timed Automata* [11] formalism, where the FSM is extended with real-valued clock variables (with certain restrictions on them). These restrictions are:

- Clock constraints may contain only the $\leq$ and $<$ operators and only against (bounded) nonnegative rational numbers
- The only two operations allowed on clocks are read, and reset to zero

As Alur and Dill have shown in their fundamental paper cited above, the infinite state space of the Timed Automata (TA) can mapped onto a FSM using the *region automaton* whose states correspond to clock value regions in the TA.

Parallel composition is naturally asynchronous for these systems, as transitions can happen with arbitrary density in real time.

The number of clock regions in the resulting region automata is bounded by

$$|X|! \cdot 2^{|X|} \cdot \prod_{x \in X} (2c_x + 2)$$

where X is the set of clock variables, and c_x is the largest constant clock x is compared to. In order to compose the region automaton, these clock regions have to be combined with the discrete states of the original system.The system's size grows exponentially with the number of clocks, so efficient state space representation is a major problem here as well.

An extension of the above formalism is the work of Fersman, Peterson and Wang Yi in [24] in which they show that the set of allowed clock operations can be extended with subtraction of constants without losing the ability to map the automaton onto a region automaton. In certain key areas such as schedulability analysis, this significantly enhances the usefulness of the timed automata [31].

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

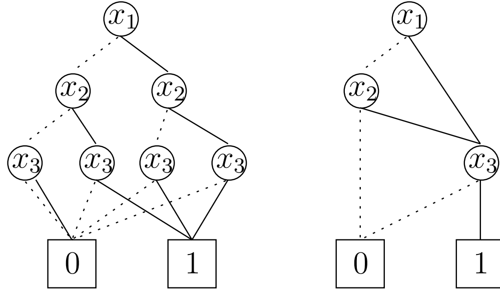


Figure 2: Truth table, full and reduced decision tree for a boolean function

Symbolic state space representation in finite-state systems

As we had seen earlier, overcoming state space explosion is a problem in the representation and analysis of system models.

Kripke structures over the finite domain D can be encoded as logical functions: first we create an encoding $\phi : \{0,1\}^m \rightarrow D$ to represent D with a set of binary vectors of length m. Then, a state can be represented by a binary function $f(x_1, \ldots, x_m)$ which evaluates to 1 if the given state's encoding is given, and the set of states is simply the conjunction of the corresponding functions. A single transition is a binary function of $f(x_1, \ldots, x_m, x'_1, \ldots, x'_m)$ accepting *(present state, next state)* pairs, and the transition relation $\mathcal{R}$ is the conjunction of all transition functions.

The labeling function L of the Kripke structure can be represented as well: we encode each predicate as a set of states where it holds.

Here, the crucial point is that how compactly and efficiently binary functions can be represented. Truth tables have exponential storage (and computation) requirements with the number of variables, so they quickly become inefficient. Conjunctive and disjunctive normal form representations are not much better, since they are closely related to truth tables.

Ordered Binary Decision Diagrams (OBDDs) [17] are a canonical representation of boolean formulas, effective both in space and operations for a large class of functions and have been successfully used in the context of modeling and verification.

OBDDs were first used for representing large FSM state spaces by McMillan in 1987 [19], and this lead to the development of the SMV tool [40]. Since then, OBDDs and their various extensions have been in mainstream use in practically every area of *model checking*: in timed and untimed systems, probabilistic and hybrid models using various state and transition representation schemes. Model checking, as it will be discussed later in this chapter, is an automated approach for property verification for models.

OBDDs are directed acyclic graphs with a root node, internal nodes and exactly two terminal nodes (representing 0 and 1). This structure is also known as binary decision tree. Each nonterminal node is labeled with a variable of the function, and has two outgoing edges (for 0 and 1, respectively). Evaluation starts from the root node, and one of the edges is taken according to the valuation of the associated variable. The evaluation finishes when a terminal node is reached, and the resulting value is the value of the terminal node. This tree is usually constructed by drawing the full binary decision tree for the function and performing OBDD *reduce* operations on it to decrease the number of nonterminal nodes. Figure 2 shows a simple boolean function with it's full and reduced decision tree (dotted edges denote 0 valuation and solid ones stand for 1).

OBDDs are *ordered* because the order of variables (the sequence they are encountered starting from the root) is important regarding their size, as illustrated by Figure 3.

Finding the optimal ordering is an NP-complete problem [15], so implementations resort to heuristics to improve efficiency, or use simple *dynamic reordering* rules which iteratively reduce the number of nodes for certain graph configurations by changing the variable ordering while an operation is being performed on the graph [46]. One such method is *sifting* which swaps the order of adjacent variables. Such *local reordering* techniques usually lead to sub-optimal ordering. For better gains (at the cost of more complex algorithms) some implementations offer *global reordering* using global heuristics resulting in a completely new variable ordering [41].

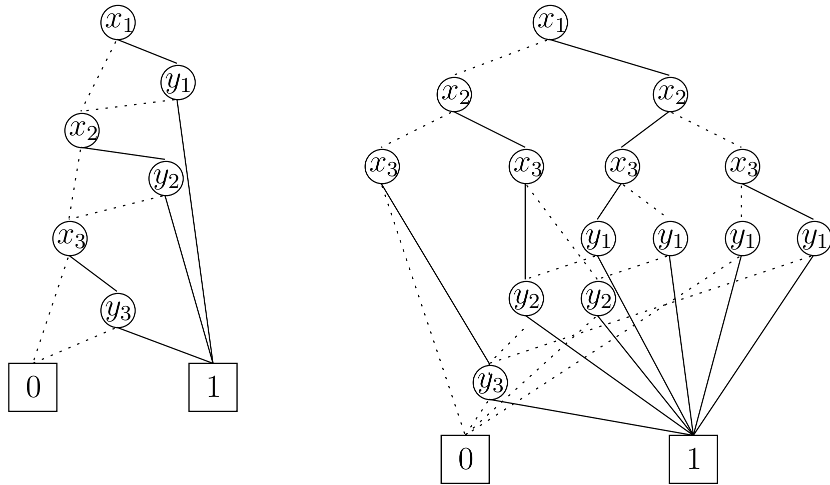


Figure 3: OBDDs for function $f = (x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_3 \wedge y_3)$ with different ordering

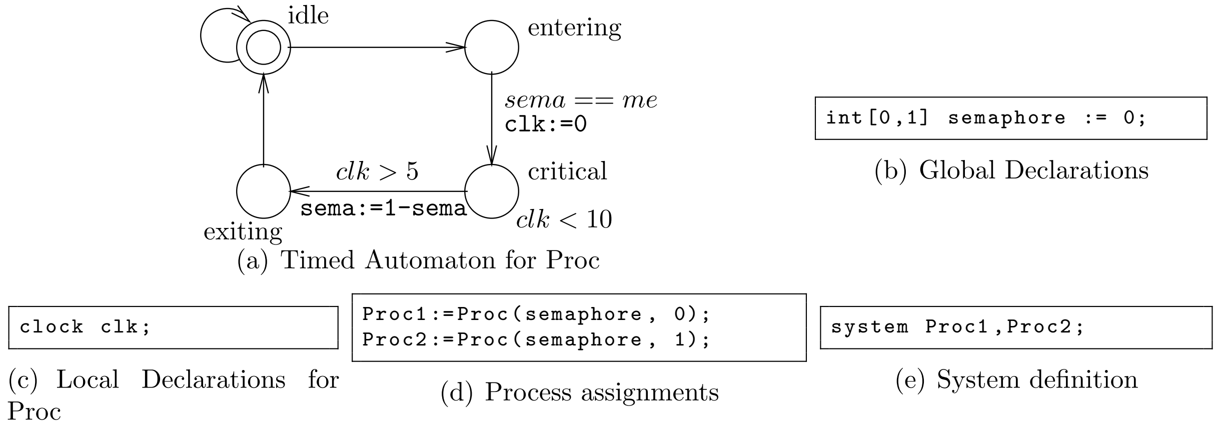
There are known functions where OBDD representation results in a very large tree (e.g binary multiplication-like and XOR-like functions), but for the average case OBDDs yield a significant reduction in representation cost.

It has also been shown that all 16 two-argument logical operators can be efficiently implemented over OBDDs. The complexity is linear in the product of the argument graphs' sizes [18].

A straightforward extension of the OBDD idea is the use of MTBDDs (*multi-terminal binary decision diagrams*) [21] which are used to represent functions whose arguments are boolean but their domain is integer. The BDD idea can also be generalized into using any discrete-domain variables instead of booleans.

Tool Example: The UPPAAL Model Checker

A good, comprehensive modeling framework based on the TA formalism can be found in the UPPAAL [34] [35] [5] toolset. UPPAAL supports the visual description of TA as a network of FSM processes. An FSM consists of states with invariants and execution attributes (`urgent` or `comitted`) and transitions with guards, synchronization labels and assignments. An `urgent` state is left immediately after any outgoing transition guard becomes true (a



(a) Timed Automaton for Proc

(b) Global Declarations

(c) Local Declarations for Proc

(d) Process assignments

(e) System definition

Figure 4: UPPAAL model for the mutex example

transition is taken as soon as possible). No time can be spent in a `comitted` state: once entered, it must be left immediately. Committed states can be used to ensure atomicity by preventing interleaving.

Variables can be global or local, and a variable can be either a clock or a bounded integer. Arrays of integer variables are also implemented. Supported operations on clock variables are reset (to zero) or comparison to an integer constant. For synchronization, rendezvous channels are provided.

Figure 4 demonstrates the mutual exclusion process in UPPAAL's visual Timed Automata language with the necessary declarations. The model was extended for the sake of the example with clock constraints saying that a process stays in state *critical* for no less than 5 units and no more than 10 time units.

Based on the software developed for UPPAAL, the DARTS team at Uppsala University has developed the Times Tool [24]. This is a less general-purpose tool than UPPAAL, and it is mainly geared towards schedulability analysis of embedded systems. It incorporates novel results for schedulability checking with timed automata, and contains algorithms for checking TA with constant subtractions on clock variables.

Specification

Specification describes properties of the system to be verified. In this area, we are mostly concerned about *safety* properties, and those are described using conditions on the system's trajectory.

Generally, we are asking questions such as: "Starting from state s_0, can the system ever reach state q ?", or "Does the system always arrive to state set Q in finite number of transitions ?" To ask these questions, we need to provide universal and existential quantifiers (*always* or *ever*), we need ways to specify state trajectories and we need logical predicates to designate states and state sets.

For systems modeled as finite-state systems and Timed Automata, *Temporal Logic* is used, and for hybrid systems the safety properties are specified as state set reachability statements based on state and trajectory operators like *Post*. The *Post* family of operators describe the set of "next" (successor) states (for a given state set) along possible trajectories of the system. The closure of *Post* gives the set of all the states reachable from the current state (regardless of the time or number of transitions required).

In the following sections, temporal logics formalisms CTL, LTL and CTL*, are going to be discussed, along with their various extensions into sparse and dense time representation and probabilistic modeling. tate set reachability specifications for hybrid systems are also introduced briefly.

Computational Tree Logic CTL*

CTL* is a branching tree logic describing properties of *computation trees* [23]. The tree has an initial state as its root, and then unwinds infinitely representing the possible state trajectories.

Formulas are composed of *path quantifiers* and *temporal operators*. Path quantifiers are to describe the branching structure of the tree and temporal operators describe properties of a path.

CTL* has two path quantifiers:

A *for all paths* — the following property is true for all paths from the given state

E *there exists a path* — for which the following property is true

In addition, there are five basic temporal operators:

X *next state* — the property is true for the next state

F *eventually in the future* — the property will be hold eventually along this path

G *always, globally* — the property holds for all states on the path

U *until* — this operator has two properties as arguments: the first one holds at a given path until eventually the second one will hold

R *release* — the dual of *until*: the second property holds until a state in which the first one will hold

CTL* statements are combined of *state formulas* (predicates being true in a given state), their negations, conjunctions and disjucntions, and *path formulas* which are CTL* operators applied to state formulas.

Temporal logics can also be extended with *fairness*. In this case, fairness constraints are interpreted as a set of states, and only those paths which contain elements of this set infinitely often are considered.

The formalism for stating that state formula p holds at state s in the Kripke structure is $M, s \models p$ (which implies that $p \in L(s)$ if p is an atomic predicate). For path formulas, we write $M, \pi \models \mathbf{Op}_1\ f_1$ or $M, \pi \models f_1\ \mathbf{Op}_2\ f_2$ to indicate that the given path formula holds for π ($\mathbf{Op}_1$ and $\mathbf{Op}_2$ are unary and binary CTL* path operators).

For Kripke structures with fairness, the operator $\models_F$ is used, and it always indicates the existence of *fair paths*, i.e. if applied to a state (for example $M, s \models_F p$, $p \in L(s)$) it means that there exists a fair path starting from s.

It is worth noting that operators $\vee$, $\neg$, **X**, **U**, and **E** are sufficient to express any other CTL* formula.

Unrestricted CTL* is very general and hard to analyze. Thus, in practice usually a subset is used, for example CTL and LTL as we'll see in the following section. Historically, CTL and LTL were developed independent of each other and CTL* was developed later to unify the two.

About notation: there are two styles for denoting temporal operators throughout the literature. The one used in this document denotes temporal operators with capital Latin-letter mnemonics. The other style uses a mathematical symbol for each operator: $\forall$ for **A**, $\exists$ for **E**, $\bigcirc$ for **X**, $\diamond$ for **F** and $\square$ for **G**. For **U**, some kind of $\mathcal{U}$ symbol is used.

CTL and LTL

CTL and LTL (Linear Time Logic) are two sub-languages of CTL*. CTL is a restricted subset of CTL* where a temporal operator must be immediately proceeded by a path qualifier, or more precisely, where the *path formulas* can only be constructed by the following rule:

- If f and g are state formulas, then $\mathbf{X}f$, $\mathbf{F}f$,$\mathbf{G}f$,$f\mathbf{U}g$, and $f\mathbf{R}g$ are path formulas.

There are ten basic CTL operators (**AF**, **EF**, **AX**, **EX**, **AG**, **EG**, **AU**, **EU**, **AR**, **ER**), and similarly to CTL*, they can be expressed with **EX**, **EG**, **EU**, and $\neg$, $\vee$.

Figure 5 illustrates the four most common CTL operators.

Some examples for CTL specifications:

- from each reachable state, there is always a path back to s_0: $\mathbf{AG}(\mathbf{EF}(s_0))$
- any path will eventually reach s_0: $\mathbf{AG}(\mathbf{AF}(s_0))$
- mutual exclusion: $\mathbf{AG}\neg(critical_1 \wedge critical_2)$

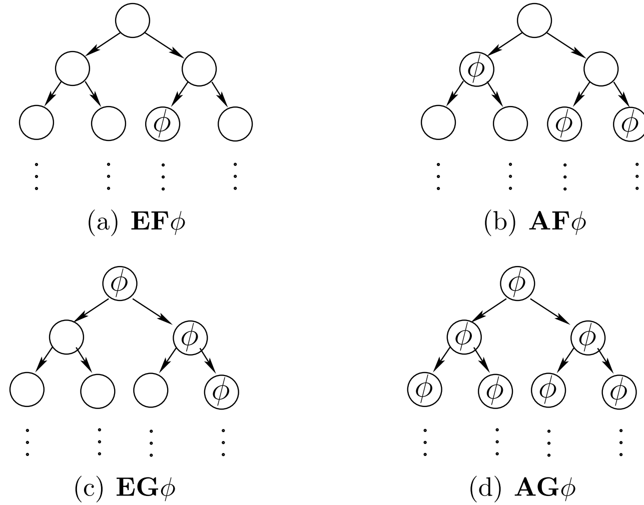


Figure 5: Basic CTL Operators

In LTL, Linear Temporal Logic, formulas have meanings on individual computation paths. LTL formulas are restricted to the form $\mathbf{A}f$ where f is a path formula over atomic propositions of the system. More formally:

- If $p \in AP$ then p is a path formula
- If f and g are path formulas, then $\neg f$, $f \vee g$, $f \wedge g$, $\mathbf{X}f$, $\mathbf{F}f$,$\mathbf{G}f$,$f\mathbf{U}g$, and $f\mathbf{R}g$ are path formulas.

Examples for LTL formulas:

- mutual exclusion: $\mathbf{G}\neg(critical_1 \wedge critical_2)$
- a request is always eventually acknowledged: $\mathbf{G}(req \rightarrow \mathbf{F}ack)$
- *try* holds as long as *critical* does not hold: $\mathbf{G}(try \rightarrow (critical\mathbf{R}try))$
- eventually it will be *allright* and it will stay so: $\mathbf{A}(\mathbf{FG}(allright))$

LTL and CTL have different expressive powers: CTL allows branching but does not allow combining temporal operators, whereas in LTL path formulas can be combined using logic operators (such as $\neg$, $\vee$), but branching cannot be expressed. They are both sublogics of

CTL*: there are formulas in CTL that cannot be expressed in LTL and vice versa, and there are formulas in CTL* that are not in either CTL or LTL. For example, the LTL formula $\mathbf{A}(\mathbf{FG}p)$ cannot be expressed in CTL, and the CTL formula $\mathbf{AG}(\mathbf{EF}p)$ has no equivalent in LTL. Obviously, the conjunction or disjunction of the two formulas exists in CTL* but does not exist in LTL or CTL.

Verification

For **verification**, the following methods are used:

Theorem proving The model $\mathcal{M}$ and specification Φ are expressed as a set of formulae, and $\mathcal{M} \models \Phi$ is checked through symbolic proof construction using inference rules.

Model Checking through exhaustive state space search Explicitly enumerate the state space and verify the specification using brute force.

Symbolic Model Checking Enumerates or iteratively discovers the state space using symbolic methods such as OBDDs (finite state systems) or polyhedral state set representation (linear hybrid systems). Finite-state tools support CTL and/or LTL, as there are no efficient algorithms for full CTL* implementation.

Theorem Proving

In theorem proving, the system and the specification are provided as a set of mathematical statements, and verification proceeds by iteratively proving theorems about the system based on the model formulae. The proof must show that the specification statements can be formally derived from the model formulae [23].

Automatic theorem provers are available, but in most cases an expert user has to provide guidance during the proof — the process is semi-automatic at best. The task of translating the system description into logical statements and axioms is also difficult to automate and is

generally done by a human logic expert. Because of the amount of both expertise and human interaction are required theorem provers are very expensive to use and success (termination) is not guaranteed. On the other hand, with the necessary expertise any kind of system model can be translated into logic statements, and the approach is not limited by the state space. Therefore, abstractions resulting in a loss of important detail can be avoided.

In our context (integrating formal verification techniques into the MDA framework) theorem proving is a less useful approach, because here models primarily reflect *designer* knowledge, and do not take the form of logical axioms. Automating the model translation process as much as possible is also an important goal of MDA and semi-automated proof systems requiring extensive external domain expertise (i.e. not from the design domain) do not really fit into the framework.

Model Checking for finite-state systems

Model checking is a method to algorithmically verify formal systems (systems specified in a mathematically precise formalism such as the modeling methodologies discussed above). Model checking results either verifying the given properties with certainty, or generating counterexamples by exploring the state space. For finite-state systems, the two main approaches are either full state space representation and exhaustive search, or symbolic state space representation and discovery. Both methods are fully automated, deterministic and guaranteed to terminate.

Exhaustive state space search

In the first, exhaustive case, the states are encoded as bit-vectors or enumerated through a recursive search. With this method, good estimates can be made on resource requirements of the verification process. A good example for this method can be found in the SPIN [29] modelchecker: Promela models can be translated into C programs which enumerate each state and verify the given properties.

This approach is severely limited by the *state space explosion problem*: adding variables or (parallel) composing additional automata to a system will increase the global state space exponentially. For example, adding a boolean variable will *double* the state space (each state will split into two). As we have seen earlier, composing two Kripke structures results in a KS with a state set which is the cross product of the two original sets. Composing several automata leads to huge state spaces quickly.

SPIN has yet another interesting feature to overcome this problem: It provides *bit-state hashing* which is a probabilistic state space representation technique. Instead of the full encoding of a state, only a hash value is stored in order to decrease memory and processing requirements. Of course, this method does not guarantee the full state space representation. SPIN provides a probabilistic indicator on how close the state space search is to a full one.

Symbolic state space representation

This approach uses symbolic methods like OBDDs to encode large state sets, and provides efficient verification methods which operate on sets of states. The *model checking problem* can be formalized for state sets of Kripke structures and temporal logic formulas the following way [23]:

Given a Kripke structure $M = (S, R, L)$ that represents a finite-state concurrent system and a temporal logic formula f expressing a specification, find the set of all states in S that satisfy f:

$$\{s \in S | M, s \models f\}.$$

The system satisfies the specification if all of its *initial states* are in the set.

CTL Model checking

Let $M = (S, R, L)$ be a Kripke structure, and f a CTL formula. First, we parse f hierarchically by nested CTL operators, so that the bottom of the parse tree will contain atomic propositions only. We assign these to states according to L.

Next we iterate through the parse tree until the root is reached: during the ith stage subformulas with $i-1$ nested CTL operators are processed. When a subformula is processed, it is added to the labeling of each state where it is true. Once the algorithm terminates, we will have the set of states labeled with f.

Since all CTL formula can be expressed in terms of $\neg$, $\vee$, **EX**, **EU**, **EG**, we need to provide the labeling algorithm with the following forms: $\neg f_1$, $f_1 \vee f_2$, $\mathbf{EX} f_1$, $\mathbf{E}[f_1 \mathbf{U} f_2]$, $\mathbf{EG} f_1$. f_1 and f_2 are either atomic, or have been processed in the previous stage.

Algorithm 1 $CheckEU(f_1, f_2)$: labeling states satisfying $\mathbf{E}[f_1 \mathbf{U} f_2]$

```
T := {s|f2 ∈ label(s)};
for all s ∈ T do
  label(s) := label(s) ∪ {E[f1 U f2]};
end for
while T ≠ ∅ do
  choose s ∈ T;
  T := T\{s};
  for all t such that R(t, s) do
    if E[f1 U f2] ∉ label(t) and f1 ∈ label(t) then
      label(t) := label(t) ∪ {E[f1 U f2]};
      T := T ∪ {t};
    end if
  end for
end while
```

Labeling states for $\neg f_1$, $f_1 \vee f_2$ or $\mathbf{EX} f_1$ is trivial: for **EX** we need to consider the successor states for f_1. To label states for $\mathbf{E}[f_1 \mathbf{U} f_2]$ first we find states where f_2 holds, then progress backwards using the converse of the transition relation and find all states labeled with f_1. Algorithm 1 shows the pseudocode for $CheckEU$.

Checking for $\mathbf{EG} f_1$ is slightly more complicated and requires the decomposition of the transition graph into *strongly connected components* where f_1 holds. The basic idea is that once we are "inside" such a subgraph, f_1 will globally hold, so we need to find those states which have paths into such subgraphs. The exact algorithm can be found in [23].

The entire CTL model checking procedure can be performed in $O(|S| + |R|)$.

CTL Model Checking with fairness

A Kripke structure extended with *fairness* constraints is given as $M = (S, R, L, F)$ where $F = \{P_1, \ldots, P_k\}$ is the set of fairness constraints.

The base of the model checking algorithm extended with fairness is a modified version of the *CheckEG* algorithm (briefly discussed in the previous section). Here, we decompose the transition graph again into *strongly connected components*, and mark them for fairness: We say that a strongly connected component C is *fair* with respect to F iff for each $P_i \in F$ there is a state $t_i \in (C \cap P_i)$, i.e. each fairness condition is true for at least one state in C. The procedure is then identical to *CheckEG*, except we will only consider fair strongly connected components. The complexity of the computation is $O(|S| + |R|) \cdot |F|$ since we have to decide which strongly connected components are fair.

In order to check other CTL formulas with fairness, we extend the Kripke structure with an additional atomic formula $fair$ which is true for a state iff there is a fair path starting from the state, i.e. $fair = \mathbf{EG}(true)$ for each state. We use the procedure discussed above to label each state with the new proposition. Then, we use the ordinary (non-fair) model-checking procedure on formulas updated with $fair$ according to the following rules:

1. $M, s \models_F p \Leftrightarrow M, s \models p \wedge fair$
2. $M, s \models_F \mathbf{EX} f_1 \Leftrightarrow M, s \models \mathbf{EX}(f_1 \wedge fair)$
3. $M, s \models_F \mathbf{E}[f_1 \mathbf{U} f_2] \Leftrightarrow M, s \models \mathbf{E}[f_1 \mathbf{U}(f_2 \wedge fair)]$

The complexity of the analysis is $O(|f| \cdot (|S| + |R|) \cdot |F|)$ if f has $|f|$ nested CTL expressions.

LTL Model Checking

In this section a brief and informal summary on the LTL checking algorithm will be given. As we have seen, LTL formulas are restricted of the form $\mathbf{A}f$ where f is a restricted path formula consisting of atomic propositions. Since $M, s \models \mathbf{A}f \Leftrightarrow M, s \models \neg\mathbf{E}\neg f$, it is sufficient to check for the formulas of the form $\mathbf{E}f$.

The algorithm is based on constructing a *tableau* T for the path formula f, where T is a Kripke structure and contains every path that satisfies f. Then, T and the system model M are composed. A state in M will satisfy $\mathbf{E}f$ iff it is the start of a path in the composition that satisfies f.

The tableau construction algorithm is exponential with the length of formula f, and the CTL model checking algorithms discussed earlier can be used to find the states satisfying f in the composition automaton.

A detailed description of the algorithm along with a method to represent the tableau using OBDDs can be found in [23].

Conclusions

Modeling

In this chapter, we have surveyed various modeling methods and their formal foundations.

As we can see, there are a multitude of formal, well-established modeling approaches available. Most of these methods are geared towards describing physical systems rather than design models: they are excellent tools for simulation or to study the system's behavior. Physical systems are inherently parallel and non-deterministic, and all the discussed modeling approaches handle this problem at different levels.

An important question is the represenation of *time*: in general, the more accurate is the time representation in the model, the more difficult it is to implement such models in a CBS.

In their pure form, most of these models lack one or more important features desirable for modeling engineering systems: modularity, reusability, hierarchical / logical grouping of components. Using syntactic extensions, these problems can be overcome, the question is whether or not this is worth the effort ? The prevalent direction (which is also pursued

in this thesis) seems to be using a more traditional, engineering-oriented DSML in for the design model, and to convert the model into a *analysis model* using one of the modeling approaches discussed here by matching the semantics of the design and verification models.

Specification

For modeling, we have seen many, conceptually different approaches to describe systems. For specification, we can conclude that there are only two different approaches being used: one using some kind of *temporal logic* from the CTL* family. The other approach (reachability statements on hybrid systems) is not discussed in this work. For the sake of completeness it needs to be mentioned, as it is a very important research topic today. Actually, these two approaches may even overlap if *transition systems* are being used.

For us, probably the most interesting question is how these specification statements can be formulated based on the design model. Fortunately, this does not seem to be an impossible task, since Kripke structures allow labeling their states with logical predicates, and in this manner it is possible to carry over references to the system's state.

In general, these specifications express *global* statements on the system. In a component-based system, extending a local statement into a global one or establishing the local effects of a global statement could be problematic.

For example composition is a crucial question: how specifications formulated for individual components can be carried over to the whole system, or how global requirements can be broken down to component levels. In connection to this, scalability and complexity issues are also important. Because of the limitations of the verification algorithms avaliable, the form of the specification can influence the feasibility of verification.

Verification and Model Checking

Finally, an overview of the most frequently used formal verification implementation algorithms has been given. These approaches can be categorized into two broad areas:

1. Finite-state model checking based on temporal logic
2. Hybrid system reachability analysis (mentioned but not discussed here)

For the former, there are efficient algorithms (at least for sub-classes of temporal logic languages). The algorithms themselves are efficient, but the system to be analyzed can grow very large (especially with parallel composition), therefore scalability is still a problem.

Good reachability algorithms for hybrid systems are still being extensively researched, and several important questions are still open (decidability [27] [10], convergence, approximation problems [13], [20] etc.)

CHAPTER III

BACKGROUND: MODEL TRANSFORMATIONS

Models and model transformations are the cornerstones of MDA/MDD's design and synthesis process, with the ultimate model transformation being the synthesis of the executable.

No less important is to ensure that the resulting final product will work correctly, in other words, verifying that it satisfies its specifications formulated during the design process. (We treat the specifications as parts of the design model.)

In this work, we are concentrating on the verification process. So far we have seen an overview of the tools available for verification, and the next step is to examine how these tools will fit into the MDA/MDD framework.

In order to use these tools, an *analysis model* needs to be created based on the design model. This is done through *model transformation*, and in the following sections, a discussion of model transformation methodologies relevant in our context will be given.

Unlike the area of formal verification, the theory of model transformations is much less covered by comprehensive literature. The multitude of books on modeling in (software) engineering and design are of limited use here, and the best resources are recent papers and standardization efforts by industry groups, most notably the Object Management Group's Model Driven Architecture (MDA) framework [1]. For the lack of any better (and more standard) notation, we are going to follow OMG's terminology and basic definitions, marking the differences as necessary.

Basic concepts

In the previous section, we defined a design model as a formal specification of the function, structure, and/or behavior of the system, expressed in a well-defined modeling language

(metamodel). In the following section, fundamental definitions for *transformations* of such models will be summarized.

Terminology

In a recent standardization effort [43], OMG structured the terminology for model-to-model transformations the following way:

Query A query is an expression evaluated over a model. The result is one or more instances of types defined in the model or in the query language.

The Object Constraint Language (OCL [57]) is the prominent query language in this area.

View A view is a model which is completely based on the source model. It is often read-only, or the changes to it result in direct changes to the base model. Views are typically used to hide irrelevant information, or to present a particular aspect of the system. A query is a restricted kind of view, and views are usually generated via model transformation.

Transformation A transformation generates a target model from a source model. Transformations can be *dependent* when the two models retain their coupling after the transformation, or *independent* when no further relationship is maintained. A transformation can be *unidirectional* when changes can be propagated only from the source towards the target, or *bidirectional* otherwise. In the latter case, source and target models are often referred to as the *left-hand model* and the *right-hand model*, respectively.

The reason for this structure is that these operations build upon each other (each can be considered a restricted version of the following one), and queries are often used as elemental building blocks of views and transformations.

The following terms are used in the definition of transformations:

Rule Rules are units in which the transformation is defined. A rule describes the transformation for a particular selection from the source model into the corresponding subset of the target model. A transformation is specified as a set of rules. Composition mechanisms for rules may be defined.

Declaration A declaration is a specification of a relation between elements in the LHS and RHS models. A declaration may specify uni- or bidirectional transformation, or may only serve as a selector for the associated implementation procedure.

Implementation An implementation is an imperative specification on how to generate the target model elements from the source model elements. Implementations are typically unidirectional, although bidirectional ones are also possible.

Match A match occurs when elements from the source model satisfy a declaration specification. A match may trigger the application of a rule.

Incremental transformation If individual changes in the source model lead to the execution of only the rules matching the modified elements, the transformation is said to be *incremental*.

The basic building blocks are *rules*. Generally, the following methods are used in their specification:

Declarative Using a relational or functional language to specify the relationships between elements in the source and target models in terms of functions or inference rules.

Imperative Using a traditional programming language based approach to describe an explicit algorithm manipulating the models.

Hybrid A combination of the above two — typically a declarative approach is used to select subsets of models and an imperative procedure specifies the relevant transformation.

Model Transformation Approaches

In the following classification, we roughly adhere to the categories outlined by Czarneczki and Helsen [22], who give a classification based on domain analysis.

Transformation Rules and Representations

A transformation is usually given as a collection of *rules*, as a pair of a left-hand side for the source model (*LHS*) and right-hand side (*RHS*) for target model. LHS and RHS can be represented as:

- *Variables:* can be associated with instances from the source or target models. They are also referred to as *meta-variables* to avoid confusion with variables being modeled.
- *Patterns:* Patterns are model fragments to be matched against the source or target models. They can be *textual* (string), *term* (like a logic expression) or *graph patterns* depending on model represenation.
- *Logic expressions:* computations and constraints on model elements. Non-executable logic can specify a relationship between models and executable logic can have a declarative or imperative form.

Both variables and patterns can be untyped, syntactically typed (associated with a meta-model element) or semantically typed (e.g. the result of the specifying expression is an integer).

Other characterizing aspects of transformation rules are:

- *Syntactic separation* between LHS and RHS. In a graph pattern, the two sides are syntactically separated, but in procedural rewriting code they are not.
- *Bidirectionality:* A rule (and a transformation as a whole) may be executable uni- or bidirecionally

- *Rule parametrization* for additional rule configuration
- *Intermediate structures:* Some approaches may allow or require the construction of intermediate structures during rule invocation.

Rule Scoping and Organization

Scoping allows the transformation to control where in the model rules can be applied. This is usually related to the organization of the ruleset. Rules can be grouped according to several criteria:

- *Modularity and reuse mechanisms:* rules may be grouped into packages and libraries, and inheritance and composition rules may be defined to aid organization and reuse.
- *Organizational structure:* rules may be organized by the structure of the source language (i.e. attribute grammars), by the target structure or by some other logical or practical separation

Rule scoping and organization may be influenced by *source and target relationship*, namely what kind of targets the rules support: existing, newly created, in-place (within the source), separated (source and target models can relate only though the transformation).

Rule Application Strategy and Schedule

Scheduling mechanisms determine the order in which the rules are applied. Four main categories are recognized:

- *Form:* implicitly determined by the form of rules (e.g. dependency). Alternatively, the user may be allowed to explicitly define scheduling (*external scheduling*).
- *Rule selection:* By built-in or user-specified rule selection (*match*) conditions. Interactive selection may also be supported.

- *Iteration mechanisms:* Rules may be organized to form recursion, looping or fixpoint iteration and this organization determines scheduling.

- *Phasing:* The transformation process may be organized into several phases achieving distinct goals. A frequent example is first create/manipulate target model hierarchy and then "fill it in" with attributes generated in the second phase.

Application strategy is needed if there is more than one match for a rule. It can be deterministic or non-deterministic, or even interactive. The target location for a rule is usually deterministic.

Properties of Transformations

There are a number of theoretical properties of model transformations worth examining. The transformation approach being used might guarantee some of them, or make them easy to check at a general level. With imperative, algorithmic approaches, establishing these can be a tedious, even impossible task on the level of the individual transformation specification.

The most frequently referred properties are:

Termination Does the translation always terminate? Are there cyclic dependencies, or can a fixpoint always be reached?

Confluence Does the order of transformation steps matter? Can the operations be performed in parallel? Formally, *confluence* means that a source graph can be rewritten in more than one way under a given transformation system yielding the same end result. As a consequence, in confuent systems the application order of rewriting operations is irrelevant. Confluence can be decided for the kind graph-rewriting systems this work is concerned with, as shown in [26].

Invertability and In-Place Transformations (e.g. updates on a model) Can the transformation be reversed (*undo*ne)? Can it be performed on the model in place?

Directionality Can it be applied both (many) ways without loss of information?

Composition Can multiple transformations be composed, and if so, what are the side effects?

Correctness and Consistency Can it be proved that the transformation results in a correct model? Can it be proved that all derived models preserve certain consistency conditions?

Complexity How expensive is it to perform the transformation on a general-purpose computer?

As mentioned earlier, the difficulty of verifying these properties depends on the transformation approach chosen. For example, relational and graph transformation approaches are usually naturally bidirectional, but on the other hand, without having information about the implementation, it is hard to estimate their complexity (computation requirement). Graph transformation-based model updates are usually invertible, but specifying an in-place transformation might require the notion of *state*, and not all approaches support this notion.

The Unified Modeling Language with OCL

The Unified Modeling Language [2], as part of OMG's Model Driven Architecture framework is the unification of a number of object-oriented notations and an emerging standard in modeling software design. The current UML standard is UML 1.5. The notations I am using in this paper are from UML 1.x, although they are quite general, so not much is expected to change.

Since OMG is the major standardization body in this area [43], much of their terminology and notation is widely used for the lack of anything better-defined and more universally accepted. Several research efforts, like MIC, have accepted parts of UML and use the elements of its notation sometimes in a wider role than for what it was originally intended.

The following chapter gives a short overview about the parts of UML notation relevant here, and about a particularly important UML component, the OCL language.

UML Class Diagrams

UML is essentially a collection of connected notations intended to describe the design of a software system. There are several different languages defined within the UML standard, and many of them uses visual diagrams.

The most important of those for us is the *class diagram*, which visualizes the *static structure* of the object collection. Again, what is important for us here is the visual notation chosen to express various relations (containment, generalization, association etc.) not the actual, strict UML meaning of those notations.

Class diagrams have the following elements:

Packages are structuring components used to group other related components into a single unit. Packages can be *nested*, and related by a *dependency* relationship.

Classes are representations of an entity or concept within the system being modeled. In UML, class defines the attributes, operations and behavior of the objects is represents. Relationships between classes are also visualized on the same diagram. In our concept, it is important as the *atomic* building block of a model. The visual representation is a rectangle, optionally horizontally divided into sections. The topmost section contains the name, the one below that has the attributes listed, and the rectangle on the bottom displays the operations of the class. The topmost rectangle may also optionally contain the containing package's name and the stereotype of the class.

UML defines two types of relationship between classes:

- *Generalization* or *inheritance* which defines a super- or sub-type relationship between two classes

- *Associations* which define an arbitrary relationship between any number of classes. *Adornments* can be added to the association to tighten the relationship's specification.

Figure 6 shows UML class and relationship representations. A solid line with an empty triangle arrowhead shows generalization (A generalizes B and C; F generalize E). Binary relationships are shown as solid lines (A,F), and N-Ary relationships use an empty diamond as a central junction for solid lines (B,C,D,E).

Associations : UML provides a number of *adornments* to add semantic meaning to associations defined between entities.

- *Rolename* on any or all ends to give a reference name to the object within the association
- *Cardinality* indicates the number of object that must or may form part of an association instance on the corresponding end. Further constraints like `ordered` or `unique` can be added to association ends if necessary
- *Navigability* assigns a direction to the association
- *Aggregation* indicates that one object is aggregate, and the others are parts of this aggregation. The exact semantics of aggregation are widely disputed in the community In aggregations, a black (filled) diamond is used to denote *compositional aggregation* or "ownership aggregation" vs. "by reference aggregation". The exact semantics of aggregation depends on the modeling language being used.

The OCL language

The Object Constraint Language [57] is a declarative language used in object-oriented software modeling, analysis and design. It is a subset of the industry standard Unified Modeling Language (UML) [2] that allows the specification of constraints and queries over

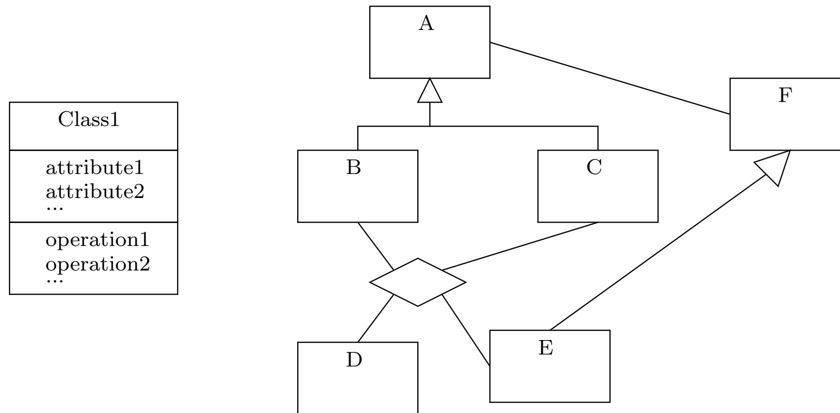


Figure 6: UML Class representation and relations

object models. These constraints are particularly useful, as they allow the creation of a highly specific set of rules to govern the aspects of an individual object. OCL has become quite successful, and it is extensively used within and outside of UML models and modeling environments.

The final version of the OCL 2.0 proposal was approved by OMG in November 2003 and — as of the time of writing this document — it is being finalized for standardization as part of the UML 2.0 specification. The official version can be found in the MGA standards & specifications repository [2]. Warmer and Kleppe wrote a good, comprehensive book on OCL [57], and introductory material can also be found in [7].

The language has a set-based semantics that is very similar to Object-Z [49], though the quantifiers and operations are represented by an OO programming language-like notation. The entire language is based on an ASCII textual representation, making it friendlier to software engineers than other formal methods of this kind. For example, the following two specifications are equivalent:

- *Object-Z*: $\forall c1, c2 \in SetofClasses \cdot c1 \neq c2 \Rightarrow c1.name \neq c2.name$
- *OCL*: `SetofClasses->forAll(c1,c2 | c1 <> c2 implies c1.name <> c2.name)`

Of course, there are deeper, semantic differences as well, but the above illustration is just an example showing that practical specification languages sometimes are quite close to their formal foundations.

Expressions and constraints

An OCL *expression* is an indication or specification of a value. A *constraint* is a restriction on one or more values of a (object-oriented) model or system.

In UML models, OCL expressions are used to:

- specify the initial value of an attribute or association end.
- specify the derivation rule for an attribute or association end.
- specify the body of an operation.
- indicate an instance in a dynamic diagram.
- indicate a condition in a dynamic diagram.
- indicate actual parameter values in a dynamic diagram.

OCL specifies four types of constraints:

invariants are constraints stating a condition that must always be met by all instances of the class, type, or interface. An invariant is described using an expression that evaluates to true if the invariant is met.

preconditions to operations are restrictions that must be true at the moment that the operation is going to be executed.

postconditions to operations are restrictions that must be true at the moment that the operation has just ended its execution.

guard are constraints that must be true before a state transition fires.

Context and language features

An OCL expression is defined for a given *context*, which is a model element, usually a class, interface, datatype or component. This is called a *Classifier* by the UML standard. The *contextual type* of an OCL expression is the type of its context or its container; OCL expressions are always evaluated for a single object which is an instance of the contextual type (the latter is called the *contextual instance*, which is not the same as the instance to which the expression belongs or for which it is defined). The keyword `self` refers to the contextual instance.

If an expression being evaluated can be resolved into multiple instances, a *collection* is returned. Constraints can be put on the size of the collection as well as the elements it contains, using *iterators*.

Models often contain attributes and associations which are *derived*, or determined from other model elements. To be complete, the model or the metamodel must not omit the way to derive such elements. OCL provides the `derive` construct to express this relation.

Attributes (or even associations) usually have initial values, and OCL specifies them using the *init* concept.

Class diagram models can specify *query operations*: these have no side effects (do not change the model's or any instances' state), and execution of a query operation results in a value usable (for example) in other OCL expressions or as statements of a stand-alone query language.

Evaluating a constraint does not change any values in the system, because OCL is totally side-effect free. A constraint merely states "this should be so", and if it is not, then the model is incorrect, but does not specify what to do about it. It is up to the user of the modeling tool to decide whether this is a fatal error or a minor mistake, and also what course to take in order to correct the model: this information is not expressed in OCL.

Conclusions

In this chapter, two important UML notations were introduced. As mentioned, the notations based on UML visual languages are widely used in the modeling and model translation community, as well as the OCL language.

The graph-based diagram language includes many high-level concepts (hierarchy and containment, interfaces, OO concepts etc.) to be a good vehicle for today's designs. The notations are intuitive, yet formal enough, and OMG as a standardizing body has the necessary gravitas to promote their usage. We can witness the increasing popularity of UML-based notations in the engineering community.

OCL, on the other hand is formal but less intuitive. It is widely used for the lack of a better standard, and it is quite good as a declarative language, but its steeper learning curve limits its general acceptance. In this respect it is similar to the logic-based specifications discussed in the previous section. Nevertheless, it fits well into our context where formality and strictness is more important than user-friendliness.

Graph Transformations

All modeling approaches discussed in this work use *graphs* to represent system structure in one way or another. For this reason, formulating the model transformation problem as a graph transformation task seems a natural choice. Graphs are well known, well understood mathematical concepts, and graph transformation has been a highly formal, and well researched area since the 1970's [12].

In particular, the methods with which we are concerned operate on typed, attributed, labeled graphs (or more precisely, *multigraphs*), like a UML class diagram. Generally speaking, graph transformation rules consist of a LHS graph pattern and a RHS graph pattern. The LHS is *matched* in the source model and replaced by the RHS pattern using particular embedding rules to integrate the different elements with the existing graph.

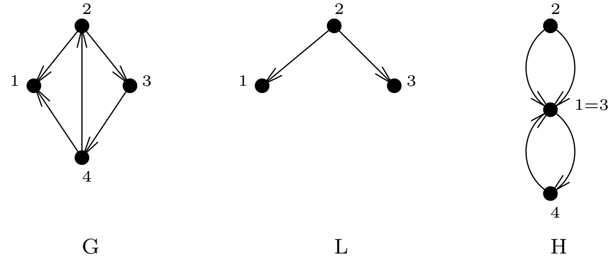


Figure 7: Graph Concepts: L is a subgraph of G and has an occurrence in H

In the following, a very general definition of graph transformation and related concepts will be given based on [12], followed by a case study featuring a graph transformation tool.

Graphs

For the purposes of the definition, we choose node- and edge-labeled, directed graphs:

Graph A (*labelled, directed*) *graph* $G = (NODES, EDGES, source, target, label)$ consists of a finite set $NODES$ of *nodes*, a finite set $EDGES$ of *edges*, two mappings *source* and *target* assigning a source and target node to each edge, and a mapping *label* assigning a labelling symbol from a given alphabet to each node and edge. An edge e goes from node $source(e)$ to node $target(e)$. If a graph contains multiple edges between the same node pair, then it is called a *multigraph*.

A graph L is a *subgraph* of G ($L \subseteq G$) if the node and edge sets of L are subsets of those in G and the source, target and label mappings coincide with the respective mappings in G.

L has an *occurrence* in G ($L \to G$) if there is a mapping *occ* which maps the nodes and edges of L to G, respectively, and preserves sources, targets and labelings (i.e. for each edge, the source in L coincides with the image of that source in G, and the same is true for the target and label mappings, respectively).

Figure 7 illustrates the above concepts.

There are further extensions to this concept: *undirected graphs* can be represented by running two edges between a pair of nodes in the opposite direction, *hypergraphs* allow edges

to have several source and target nodes, and *hierarchical graphs* are graphs where a subgraph can be collapsed to one node, and all nodes of the subgraph have edges to the nodes having edges to the collapsed node.

In *attributed graphs* nodes and edges can be annotated with attributes, and a particular case of these is *typed graphs*.

Transformations on graphs

Graph transformation is the process of (iteratively) applying a *transformation rule* to a graph. Each rule application transforms the graph by replacing a part of it: rules contain a left-hand side (LHS) L graph and a right-hand side (RHS) R graph, and the *occurrence* of L is replaced by R according to the following definition [12]:

Graph transformation rule $r = (L, R, K, glue, emb, appl)$ consists of the LHS graph L, RHS graph R, the $K \subseteq L$ *interface graph*, an occurrence *glue* of K in R relating the interface graph with the RHS, and an *embedding relation emb* relating the nodes of L to the nodes of R. *appl* is a set of *application conditions* governing the applicability of the rule.

Note that this and the following definitions are very general. In practice, transformation systems support only a restricted subset of transformations.

Application of rules An *application* of rule $r = (L, R, K, glue, appl)$ to graph G yields a resulting graph H in the following steps:

1. CHOOSE an occurrence of L in G
2. CHECK the application conditions *appl*
3. REMOVE the occurrence of L up to the occurrence of the interface graph K from G as well as the *dangling edges*, i.e. all edges adjacent to nodes being removed. This yields the *context graph* D of L which still contains an occurrence of K.

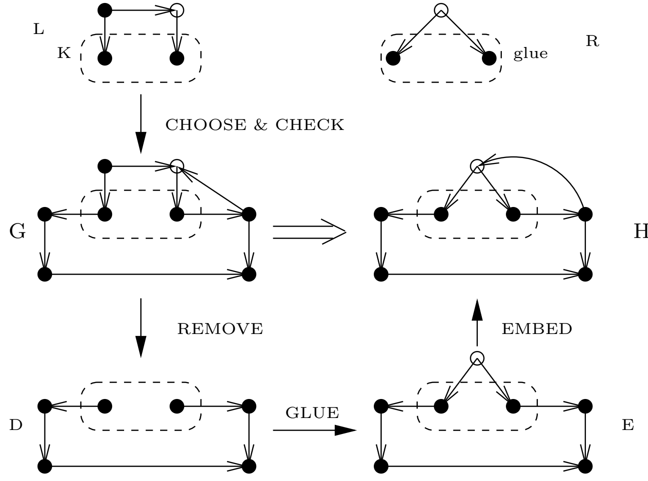


Figure 8: Illustration of a graph transform step $G \Rightarrow H$

4. GLUE the context graph D into the RHS graph R according to the occurrences of K in D and R: for every item in K identify the corresponding item in R and construct the disjoint union of D and R. This process yields to *gluing graph* E.

5. EMBED the RHS graph R into the context graph D by the *embedding relation emb*: insert the edges corresponding to the *dangling edges* removed in step 3. For each dangling edge incident with a node v in D and its image v' in G, and each node v'' in R a new edge (with the same label) between v' and v'' is established in E provided that (v', v'') belongs to *emb*.

The application of r to G yielding H is called a *direct derivation* from G to H and denoted by $G \Rightarrow_r H$ or $G \Rightarrow H$. Figure 8 illustrates the definition given above.

Properties of graph transformation

Amongst the general transformation properties mentioned previously, several have importance in the context of graph transformation. In the following list, these are detailed along with a few specific properties:

Locality The change induced by a transformation step is "local" to the occurrence subgraph: the changes can be limited and the bounds of the changes can be established by analyzing the rule specification.

Invertability *undo* operations can be performed by inverted rule applications. The following conditions guarantee invertability:

1. no dangling edges will arise during REMOVE (*contact condition*): if the image of a node in L contacts an edge not in the image of L, the node must be in K
2. the occurrence of L in G only identifies elements in the interface graph K (*identification condition*)
3. the morphism *glue* is injective
4. the *emb* embedding relation is empty
5. LHS application conditions are convertible to equivalent RHS conditions

Independence Two direct derivations $G \Rightarrow_{r_1} G_1$ and $G \Rightarrow_{r_2} G_2$ *commute* if there is a graph H such that $G_1 \Rightarrow_{r_2} H$ and $G_2 \Rightarrow_{r_1} H$.

Confluence A graph transformation is *confluent* if for each two derivations $G \Rightarrow^* G_1$ and $G \Rightarrow^* G_2$ there is a graph H such that $G_1 \Rightarrow^* H$ and $G_2 \Rightarrow^* H$. Confluence implies that every graph can be transformed into at most one irreducible graph.

Termination A graph transformation is *terminating* if infinite derivations $G_1 \Rightarrow G_2 \Rightarrow G_3 \Rightarrow \ldots$ are impossible. In general this question is undecidable [44], but there are a number of simple sufficient conditions for termination (e.g. each rule reduces the size of a graph).

Complexity Efficient matching of the LHS or checking the application conditions are central problems in graph transformation. Well-known techniques from constraint logic programming are used, but there is no efficient or generally accepted solution.

Graph Transformation Systems and Grammars

The simplest form of a *graph transformation system* is a P set of rules. The set P of rules together with an *initial graph* S and a set of T *terminal rules* forms a *graph grammar*. A sequence of direct derivations $G_0 \Rightarrow G_1 \Rightarrow \cdots \Rightarrow G_n$ is a *derivation* and G_n is said to be derived from G_0 by the rules of P. The set of graphs with symbols of T that can be derived from S by rules in P is the *language* defined by P, T, S.

Graph transformation systems are usually *non-deterministic* since multiple rules may match at multiple locations.

The Generic Modeling Environment (GME)

GME is the modeling framework for implementing the examples related to this research. In the following section, an introduction to its concepts and architecture will be given.

The Generic Modeling Environment [37] [6] is a configurable framework for creating domain-specific modeling and model transformation (synthesis) environments. Configuration is accomplished through UML *metamodels* specifying the *modeling paradigm* (modeling language) of the domain. The modeling paradigm contains all the syntactic and semantic information of the domain, and also contains basic presentation (visualization) information. The syntactic and semantic information entails the main domain concepts, their relationships and organization as well as model construction rules. The modeling paradigm defines the *family of models* that can be created using the domain(or paradigm)-specific modeling environment.

The metamodeling language is based on UML class diagrams with OCL constraints. GME automatically generates (configures) a domain-specific modeling environment (visual editor and API) based on the modeling paradigm. This environment is then used to build and process domain models that are stored in GME's model database format or exported into XML.

GME is open-source, and has a modular, extensible architecture based on Microsoft COM technology. GME is easily extensible; external components can be written in any language that supports COM (C++, Visual Basic, C#, Python, Java etc.). GME's advanced features include:

- Built-in constraint manager enforcing domain well-formedness rules in the visual model editor
- Metamodel composition and reuse facilities
- Model libraries within a family of models
- Sophisticated type inheritance within the framework
- "Decorator" API for custom model visualization

The GME Metamodeling Environment

GME relies on a set of abstract modeling concepts that are generic enough to describe a wide range of domains. These concepts include:

- containment (e.g. composition, aggregation, hierarchy)
- associations and interconnections
- multi-aspect (multi-view) modeling
- numerical/textual value-storing attributes

These concepts can be "instantiated" (customized) for each domain to express domain concepts directly. The key idea is the consistent application of the meta-level concepts through the modeling layers.

GME supports the four-layer metamodeling architecture advocated by the UML specification, as illustrated in Figure 9.

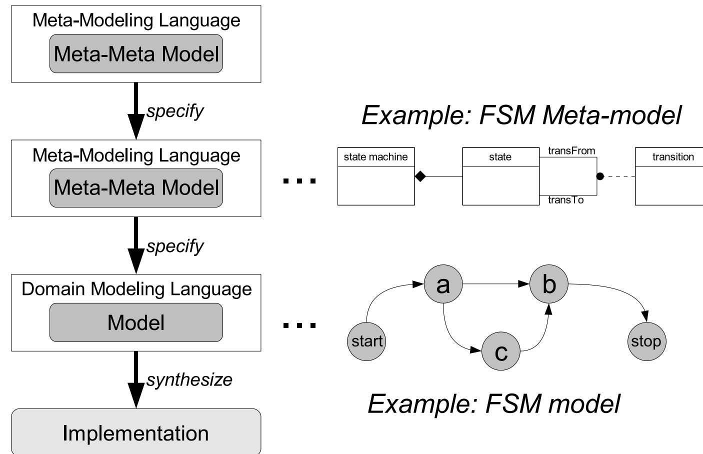


Figure 9: Four-layer modeling architecture with GME

The metamodel describes the family of FSM models with structure, consistency and visualization rules (omitted from the Figure). Then, the GME model API can be used to process (e.g. analyze, generate code based on it etc.) the models designed in the visual editor. The GME API supports several different kind of *add-ons* (software components working on the in-memory image of the model being edited). Such add-ons include *decorators* to aid visulatization, and most importantly *model interpreters*. Interpreters are paradigm-specific programs to process models (e.g. code generators).

The GReAT Graph Rewriting and Transformation System

Graph Rewriting and Transformation (GReAT) [30] [25] is an UML-based approach for model transformations. It enables the specification of explicitly sequenced graph rewrite transformations which are represented by visual UML diagrams.

GReAT is implemented within the GME [37] modeling framework, and provides seamless integration with any DSML implemented with GME. GReAT is realized on top of the Universal Data Model (UDM [14]) toolkit, which is a reflective, meta-programmable software package for developing modeling tools.

The GReAT package contains various GME interpreters to perform its functions (e.g importing GME meta-models to build class libraries to be used in the transformations, a GReAT interpreter, a GUI debugger and a C++ code generator for transformations).

GReAT aims to solve a number of shortcomings found with other graph/model transformation implementations:

1. *Specification of different graph domains:* by importing well-defined UML metamodels from GME. In GReAT, a UML metamodel (with its OCL constraints) serves as a "graph grammar" describing all valid configurations.

2. *n-to-m independent transformations across different domains:* by building a unified class library from the imported metamodels.

3. *cross-references between various domains:* by automatically specifying a temporary "inter-domain" that contains the information about cross-links, and showing them in the visual transformation model editor

4. *efficient control flow:* by providing several high-level control structures to organize rules

5. *aid programmer productivity:* by providing means to organize the rules into hierarchical transformation libraries; a visual editor and debugger; supporting the OO concepts of the GME toolset

The GReAT language consists of 2 major parts. In the following sections, these will be introduced:

1. Graph Transformation language

2. High-Level Control flow language

Graph Transformation Language

In GReAT, a transformation is built from elementary rewriting rules. Each rule contains a pattern graph. Each element in the pattern graph explicitly refers to the *unified metamodel*, which is generated from the source and target metamodels and the crosslink definitions. Rules violating the unified metamodel are not allowed. This, and removing termporary elements such as crosslinks at the end of the transformation ensures that the generated output model conforms the output metamodel as well.

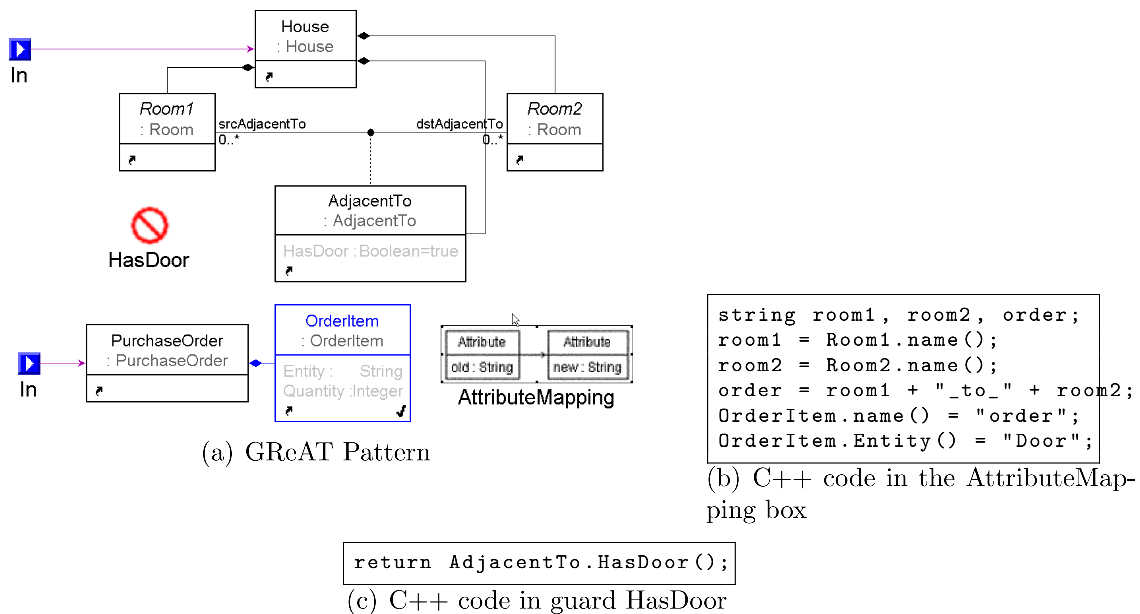
The transformation itself is specified as a set of partially ordered transformation rules. A transformation rule contains a pattern graph with vertices and edges. These pattern objects refer to specific types from the unified metamodel. When working with (source and target) model instances, the GReAT engine maps the patterns formulated on the unified metamodel onto the souce and target models. Thus, the transformation writer is presented an unified view of the source and target models, making the specification of patterns much simpler. Each pattern object has an attribute called *Action* specifying the object's role in the pattern. *Action* can have three different values:

1. *Bind:* Match object(s) in the graph.

2. *Delete:* Match object(s) in the graph, then remove the matched objects designated with the *delete* action.

3. *CreateNew:* These objects are created anew, provided that the rest of the pattern matched sucessfully.

A rule is executed by first matching every object with *bind* or *delete* actions. If the pattern matching was successful, then each object with the *delete* flag is removed, and then the objects marked with *CreateNew* are created. GReAT avoids the possible conflicts stemming from the *delete* operation by restricting the number of matches: an object is deleted only if it is matched exactly once (it cannot be deleted while a subsequent, not yet processed match holds a binding for it).



(a) GReAT Pattern

(b) C++ code in the AttributeMapping box

(c) C++ code in guard HasDoor

Figure 10: Example GReAT rule

Rules may also contain procedural (C++) code blocks using the GME API as *Guard* conditions or *AttributeMapping* blocks. Guards are boolean-valued functions on the values of pattern elements' attributes. All guards have to evaluate *true* in order for a pattern to match. AttributeMapping blocks can be used to set the attribute values of newly created objects, and/or modify attribute values of matched (*bound)* objects.

Realization of the graph transform language

GReAT is used to transform models conforming to one metamodel into different model(s), where the metamodel for the target(s) are also given. Therefore, it is important to allow only patterns (in both directions) which conform to their respective metamodels. This is achieved by importing the source and target metamodels into GReAT (into the transformation model to be precise), and having the user specify the metamodel for any temporary elements needed for the transformation.

Figure 10 shows a typical GReAT rule. The pattern matches a *House* object with two adjacent *Rooms*, and adds a corresponding new *OrderItem* to an *OrderList*. Rectangles with a small arrow in the lower left corner refer to metamodel elements from the UML

class diagram. GReAT operation is indicated in the lower right corner of the rectangle. A small checkmark indicates *create*, a small 'x' stands for *delete.* Elements with no designated operation are for *matching* (or *bound* in GReAT terminology). The input (a.k.a. LHS) side of the patterns in the set of elements to match (no designation in the lower right corner). The output (RHS) of the pattern is the set of elements marked with a "change" operation (create or delete).

Association role names are written next to the edges, and the small rectangles with the triangle inside represent *ports.* Ports are used to pass object references from rule to rule. They have two roles in GReAT:

1. to provide *initial binding* for rules in order to speed up computation of the matching set. The rule gets references from one or more previous pattern's matches (*House* and *OrderItem*). This way pattern repetition can be avoided and pattern matching can be sped up significantly.

2. to provide for implicit rule sequencing and decomposition of complex rules into simpler parts.

The code setting model attributes via the *attribute mapping* box illustrates the C++ API provided by GReAT. Matching objects are instantiated automatically and can be referenced by their pattern name. Similarly, the rule has a *guard* box, containing C++ code. The guard has to evaluate (return) `true` for the pattern to match.

Rules operate on *packets*, which are $(port, bound_data_object)$ pairs, and there is extensive (implicit and explicit) support for the routing of these packets to streamline processing. This enables the nesting of rules and the building of high-level control structures, as discussed in the next section:

Control flow language

GReAT supports the following control flow constructs:

Sequencing Rules are chained by their in/out ports and pass *packets* to enable the firing of each other. For startup, initial binding is provided for the *root folder* of the models being processed. A rule sequence is *terminated* if the last rule has no output ports or generates no output packets (matches).

Hierarchy Rules can be grouped into *rule blocks* which have the same input/output port interface as individual rules, and different strategies are available to define internal packet propagation (and subsequently rule activation) within rule blocks: a simple *Block* will push each incoming packet through its first internal rule. A *ForBlock* will process the first incoming packet and keep executing its internal rules until a (ForBlock-level) output is generated, at which time the ForBlock begins processing the next input packet.

Recursion A high-level rule (rule block) can call itself: a *reference* to any block (including itself) can be placed inside and packets routed into its input ports.

Conditional branching Rules with special semantics are defined for branching: a *Test* block usually has a few inputs, and several output ports. Inside, it contains *Case* rules, which can perform matching only. The packets matched by a *Case* rule are forwarded to selected output port(s) of the *Test* block. Therefore, a *Test* block will usually emit output only on a few of its output ports, so only a few of the connected rule blocks will fire.

Non-determinism When a rule is connected to more than one follow-up rule or there is a test with multiple successful cases, the execution becomes non-deterministic. The execution engine chooses a path arbitrarily, and executes it completely before a next path is chosen and executed. All execution paths are eventually executed, but the order is arbitrary.

CHAPTER IV

IMPLICIT PLATFORM MODELING: CASE STUDY

This chapter defines and explains *platform modeling*. After introducing the concepts, a detailed, end-to-end case study is presented. The case study illustrates *implicit platform modeling* and also motivates the research described in chapter V.

In the case study, an embedded system design language specifies components and interactions. SMOLES [53] is a high-level DSML, implementing a variant of the dataflow MoC.

SMOLES systems are realized using the DFK [51] dataflow engine, which is a "third-party" software library written in C++. DFK can be configured into either *single-* or *multi-thread* mode: in single-thread mode parallel execution is emulated by a scheduler evaluating dataflow trigger conditions.

The case study demonstrates the construction of an analysis model using *model transformation.* A Timed Automata (TA) analysis model is built, corresponding to a SMOLES system, as realized over the DFK platform in single-thread mode.

The analysis model captures key properties and behavior of the DFK engine. (actor scheduler, internal resource constraints etc.). The analysis model is generated in the UPPAAL [34] model checker tool's concrete syntax.

Thus, the verification of the resulting analysis model can answer questions regarding the properties of the end system as implemented over a particular platform, DFK.

This case study was prepared using the GME framework (for modeling the languages involved), and GReAT was used to specify and execute the SMOLES → UPPAAL model transformation. The results of the case study were published in [53].

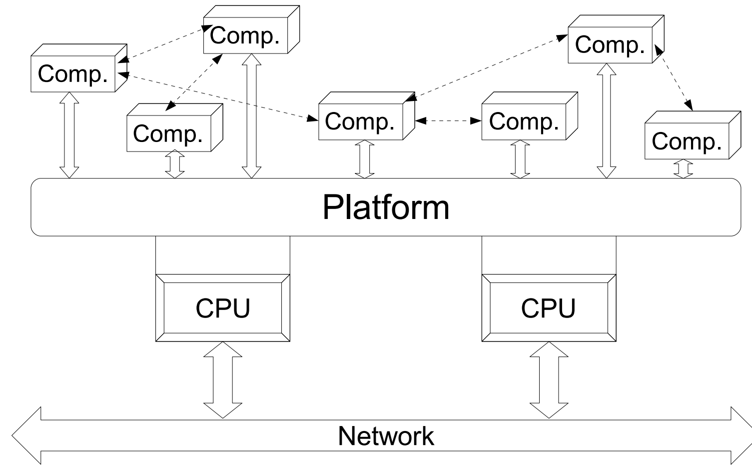


Figure 11: Component-based system implemented over a platform

The Platform concept

Design models are *abstractions* of the system, as they describe complex systems and abstraction is the human way to conquer complexity. In component-based systems, design is constructed from components that have well-defined behavior and interfaces for interaction with each other. Components interact with each other through precisely defined interaction patterns, called *Models Computation (MoC)*. In traditional software systems, the main (sometimes the only) method of component interaction is the procedure call, with the usual call-return semantics. For many systems this is not sufficient, and complex interactions patterns are needed. The hardware typically gives direct support for the call-return interactions via the subroutine call/return instructions, but very rarely gives direct support for more complex concurrency patterns. These concurrency patterns are implemented using some underlying computational framework or infrastructure that is a very good example for the *platform* concept. As mentioned in Chapter I, in the context of this work *platform* is defined as the

> *Collection of hardware, software or middleware services and resources the system is implemented over.*

The platform provides an abstraction layer above the raw hardware (which may include multiple processors with communication, etc.), and defines the interaction patterns among

components. We make the assumption here that all component interactions happen via the platform, and no other component interactions are allowed. Thus, component based systems are constructed as shown on Figure 11. In the figure, broad arrows indicate the physical interactions (between the component and the platform), while thin dashed arrows indicate the logical interactions (between components). Obviously, the latter are implemented by the former. Note that this platform-oriented view is compliance with the principles of *platform-based design*, as defined by Sangiovanni-Vincentelli and Martin in [47]:

> *In general, a platform is an abstraction that covers several possible lower-level refinements.*

In the embedded design process we often need to perform analysis on the design. Such analysis answers questions about real-time properties, resource limitations and so on. Obviously, the precise (and ultimately satisfying) answers to these questions could be given only after analyzing the code of the entire system, but this is technically not feasible. Instead, designers typically analyze the model of the system and draw conclusions about the final system based on the models. This approach accepts the fact that models are abstractions (and somewhere it needs to validated that the abstractions are valid), and proof obtained is valid only if the models are valid.

Note that (at least) two kinds of models are needed: models for the components and a model for the platform. Note that the component models are not necessarily the same as the models designers have created. The designer can work on a higher level, and lower-level models could be necessary for analysis. In terms of the language of the MDA: while the designer could work with Platform-Independent Models (PIM), for analysis one needs Platform-Specific Models (PSM) for the components.

For the platform, one needs a model that unambiguously captures the platform's behavior and describes how the components interact with each other via the platform. Note that the overall system is a composition: $C_1 \| C_2 \| \ldots \| C_n \| P$ (that is the components are

composed with the platform). The objective of the analysis is to compute the properties of the composed system consisting of the components and the platform (models).

Note that the platform model is abstract (in the general sense), but it must be made concrete to the configuration of specific applications. The abstract platform model captures how components interact in general, without referring to any specific component configurations. However, we are always interested in concrete, specific systems, where components are "wired" in a specific way. Therefore the platform model must always be concretized for the application one wants to analyze.

To summarize, analysis of component-based embedded system designs necessitates that platforms are modeled (in addition to components). The MoC defines the rules how the components interact with each other, but the platform assigns concrete (observable) behavior to this interaction, as every operation of the system is manifested through the platform.

In summary, we can say that

> *For the analysis of specific designs it is necessary to have the concrete model of the platform, which includes the models for specific component interactions.*

The requirement for concretizing the platform models can be solved using a transformational approach, as illustrated in the following case study.

Design-time model analysis

During the design process, we often need to perform analysis on the design. This analysis can be used to answer important questions about the system: Does it meet all required deadlines ? Does it deadlock ? What is the maximum latency between the arrival of an event and generating a response to that ?

Some of these questions can be answered by performing analysis on the DSML PIM model, such as finding deadlocks in a dataflow network. Most of these questions can be answered though only knowing key properties about the implementation platform. For deadline and

latency analysis, we need to know about inherent platform delays, such as token propagation delays. It is also necessary to know how the platform scheduler works.

Thus, the analysis needs to be performed on the PSM of the design. Analysis methods capable of answering the above questions, such as model checkers work with mathematically precise abstractions, such as *Timed Automata* [11]. The input languages accepted by model checkers (and simulators) are intended to capture the *behavior* of physical artifacts. Therefore, they are much different from the design languages intended to describe *structure*. As a consequence, the structures described by the design need to be modeled in terms of analysis languages, capturing their physical behavior (semantics).

This modeling is accomplished by a *model transformation*: the design model is mapped onto an analysis model, capturing the intended behavior of the implementation. In other words, it can be said that this transformation assigns a certain semantics to the design. This semantics is described in terms of the analysis language, and it is an abstraction of the physical semantics (behavior) of the implementation. Thus, the correctness of this abstraction is crucial to the validity of the analysis results obtained by verifying this analysis model.

The SMOLES design and synthesis environment in GME

The following section introduces the SMOLES modeling language and the DFK dataflow platform.

The SMOLES Language

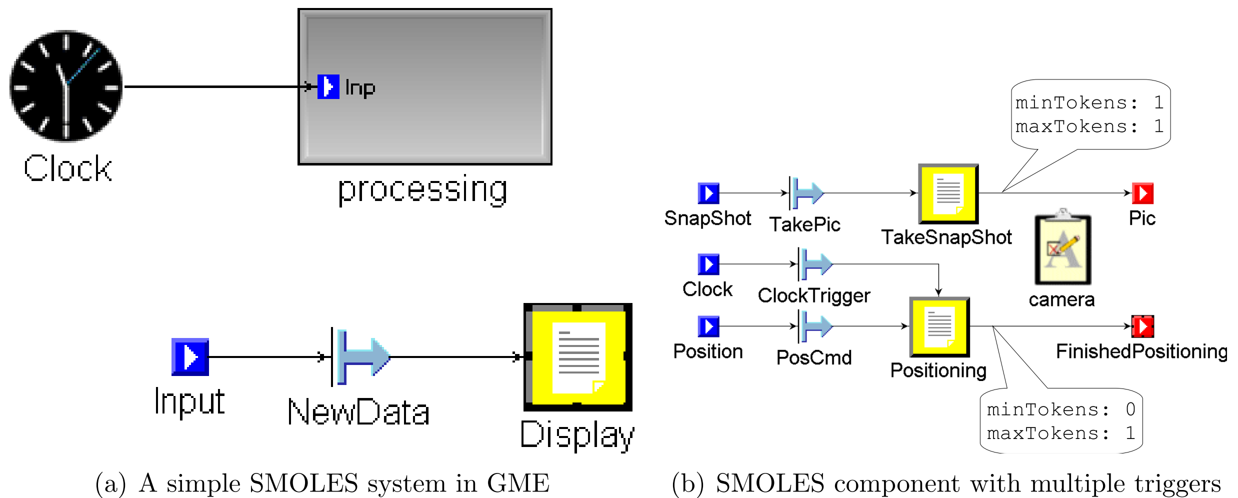
The Simple Modeling Language for Embedded Systems is used for the construction of small, component-based embedded systems. The formal definition of SMOLES, along with a formal definition of its execution semantics, is given in [53]. SMOLES has two main structural concepts: *components* and *assemblies*. Components are functional objects that perform computations, while assemblies act as hierarchical containers that encapsulate components and describe the dataflow between them.

The major concepts of the language are as follows.

Components represent concurrently executing objects and contain the following ingredients:

- *Input* and *output ports*, which are used to pass *data tokens* between components.
- *Attributes*, which are data members of the components.
- *Methods*, which are operations the component implements.
- *Triggers*, which describe how arriving data will trigger methods.

Within *components*, input ports feed into triggers that activate a single method. The triggers specify *firing conditions* for methods: If multiple ports are fed into a single trigger, then ALL of the ports have to have data available to activate the trigger. If multiple triggers are connected to a method, then the method executes if ANY of these triggers becomes active. When the condition is satisfied, the activated method executes. Methods can also be connected to output ports, on which they can send data tokens to downstream components. The number of tokens produced by a single method execution could be specified as a range of integers. (Figure 12(b) shows a SMOLES component with multiple trigger conditions. Token emission is also indicated.) One of the methods can be marked as initial, which will be executed when the component is instantiated for the first time. The methods' worst- and best-case execution times are also modeled as *WCET*, *BCET* attributes, respectively. Components also contain *attributes*, which can be implemented as instance data members. Methods access the attributes as instance variables of their owner (the component object). Input and output ports are realized as member variables of the component object, accessible through an implementation-specific API.



(a) A simple SMOLES system in GME (b) SMOLES component with multiple triggers

Figure 12: Sample SMOLES systems in GME

Assemblies contain components, other assemblies, and connections representing dataflows between components. Like components, assemblies can have their own input and output ports. Assemblies may also contain three additional atomic elements:

- *Timers* that produce data tokens with a fixed frequency.
- *Queue* objects, which are multi-writer multi-reader data structures and ensure that each incoming token is passed on through each outgoing connection.
- *Queue references* which provide access to data flows in foreign assemblies.

Within Assemblies, a port can be connected to precisely one other port. If data produced by one port should be sent to multiple other ports, or if multiple data streams should be merged into a single port a Queue has to be inserted. Component input ports in an assembly can also be connected to Timers. If there is a need for accessing a non-local data flows in a specific assembly, one can place a queue reference object into the assembly that acts as a pointer to the remotely declared queue object.

Figure 12(a) shows a very simple SMOLES system as modeled in GME. It consists of a single assembly, `System`, (on the left). The assembly has a Timer, `Clock` which is connected to component `processing`'s `Input` port via a dataflow link (straight arrow).

The internal structure of the component is shown on the right. There is one Method, `Dispay`, triggered by a trigger condition `NewData`, connected to the Input port. In other words, the arrival of a data token into `Input` will trigger `Display` to execute.

The visual model editor's focus is on `Display`, (light frame with four dots), thus the editor shows `Display`'s attributes at the bottom: It has a *WCET* of 5000ms and *BCET* of 100ms, and it is not an initial method.

If executed, `Clock` will generate a data token every time its period (a model parameter) expires. This will trigger the execution of `Display`. Thus the above example describes a very simple time-triggered system.

The DFK platform

The DataFlow Kernel (DFK, [51]) is a runtime environment and software library for the DataFlow model of execution. DFK allows creating and executing parallel and distributed data-driven dataflow networks. The first version of DFK was implemented in C++, leveraging on advanced object-oriented concepts. DFK compiles and runs on Microsoft Windows and UNIX.

A lightweight version of DFK was also implemented for a family of Hitachi industrial microcontrollers, running the μC [32] operating system in a severely resource-constrained environment [56]. DFK was also ported to the restricted Java environment offered by RCX bricks [36], which are the "brains" of LEGO Mindstorms robots, also using a microcontroller as CPU [45].

DFK supports *asynchronous* dataflows with a high level of dynamism. Dataflow connections and actors can be created, registered and removed from the network runtime.

DFK provides abstract base classes for user-provided functionality (*actors*, or *nodes* in DFK). The execution of the dataflow network is coordinated by the `Kernel` class. The user parameterizes and instantiates the Kernel, then instantiates the actor sub-classes and registers them with the Kernel. The dataflow connections are established through Kernel

API calls, and finally the execution begins by invoking the Kernel's `run()` method. DFK also provides a base template class for *data tokens*, with operators similar to those of typed arrays. Dataflow links are implemented as finite-capacity FIFO buffers between Output and Input ports (see below).

Nodes implement the dataflow actor concept. Nodes contain *Input* and *Output* ports (dataflow endpoints). Input and Output port classes provide an API for the user for data token read and insertion.

Each node has a pair of *(InputTrigger, OutputTrigger)* conditions associated. Trigger conditions can have the following values:

- `TM_ALL` all ports are *ready*
- `TM_ANY` at least one port is *ready*
- `TM_NONE` always true (the trigger always evaluates true)

An Input port is *ready* if it has data token(s) available for reading. Output ports are *ready* if the associated dataflow link is not full (i.e. a token.can be inserted).

A *node* is ready to *fire* if both trigger conditions (Input,Output) evaluate true. When a node *fires*, the node classes `run()` method is executed. `run()` is an abstract method of the actor base class, i.e. the user has to provide an implementation when subclassing.

DFK can be configured for *single-threaded* and *multi-threaded* execution. In single-threaded mode, the `Kernel` takes care of actor scheduling by evaluating nodes' trigger conditions and serializing calls to their `run()` methods accordingly. In multi-threaded mode each node has its own OS thread. Input and Output read and write calls block if tokens cannot be read / written. Thus there's no need for explicit actor scheduling. (In single-thread mode, read / write calls to ports not ready return with an error).

Implementing SMOLES over DFK

SMOLES is a high-level language with its semantics defined in abstract terms. When realizing SMOLES systems, this semantics needs to be implemented over particular run-time platforms. In Model-Based Development, this realization takes the form of *program synthesis*. During synthesis, source code corresponding to the actual design is generated. The user-specified functionality is also included as source code or object code. The generated code leverages on further software libraries and OS (and *platform*) API.

DFK is a software library providing the implemention of a dataflow MoC variant. Using DFK as the implementation platform offers two advantages:

1. Reduced development effort, since a number of concepts required for SMOLES is available in DFK (e.g. data token, dataflows, actors for SMOLES components etc.)

2. Portability: if the user-specified functionality in the design is not OS-specific, the generated code will compile and run on any OS DFK has support for.

It is important to note that SMOLES models specify functionality as user-supplied source / object code for component methods. SMOLES defines no explicit API for token passing / manipulation. Thus, implementation over DFK is possible only if the method code uses the DFK API to access data tokens. Had SMOLES have an API defined, it would have to be implemented using DFK services.

Some of the functionality DFK offers is not required for SMOLES, such as dynamic dataflow creation / deletion or the ability to connect multiple dataflows to a port. (SMOLES systems are static and the language allows one-to-one port connections only).

Some concepts required by SMOLES are not available directly in DFK. For these concepts, either *glue code* needs to be developed, or the design model modified (preserving the end system's properties).

Problems to be addressed are as follows:

Assembly hierarchy SMOLES supports the hierarchy of assemblies (and components within). The DFK network is a flat network of nodes. The problem is solved by mapping the SMOLES design onto an equivalent flat network of Components, Timers and Queues.

Queues Code for a DFK node implementing the Queue functionality needs to be developed. This is very simple: read from any port, copy the token to all output ports. In the actual implementation, a Queue is first transformed into a special component, with the necessary number of input and output ports. It has a single method, `copy` method, triggered by any of the input ports. and emitting a token to each of the outputs on invocation.

Timers In single-thread mode, implementing a node with Timer functionality is challenging. The node cannot *sleep*, since this would block the execution of any other nodes ready to run. There are two possible solutions:

1. Set the timer nodes' input trigger to TM_NONE, thus the node fires whenever possible, and during firing, check the expiration of the timer. This solution is correct, however terribly inefficient.
2. Having the possibility to modify the DFK source code, a special timer node with Kernel support could be created. The expiration of the period is checked inside the Kernel's dataflow scheduler and a token is inserted periodically. This solution is efficient, but requires modification of the DFK. In the case study, this second choice was modeled: Timers have explicit Kernel support.

In multi-threaded mode this problem is not a problem, since Timer nodes can *sleep* and block their thread.

Method trigger conditions SMOLES supports multiple Methods within a component, with a rich input trigger condition language. This is mapped onto DFK as follows. Node input triggers are set to TM_ANY. The node's `run()` method contains dispatch

code synthetized by the code generator. It is composed of `if` instructions with conditions built of `Inport.ready()` queries matching the SMOLES trigger condition. SMOLES component Methods are mapped onto node methods, and these are invoked when the corresponding `if` condition evaluates true.

Using platforms with abstractions higher than OS level can be quite helpful in reducing development effort. Yet, there is still a gap between the platform abstraction and the abstraction of the DSML. A platform is a *generalization*, intended to be used for a family of DSMLs. While implementing the DSML over a particular platform, the implementor has to be mindful of these differences in abstraction, i.e. that the available platform concepts might not match directly the DSML's requirements.

Modeling and synthesis environment

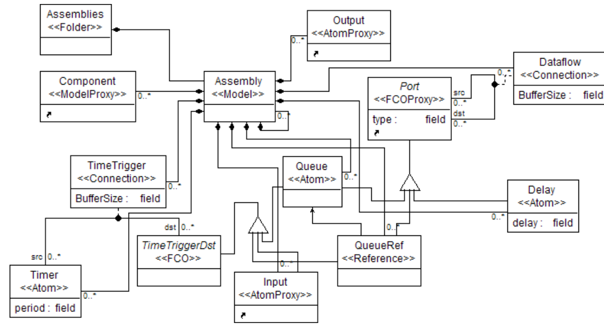
For the SMOLES language, a visual modeling and program synthesis environment was specified in GME. SMOLES was *metamodeled* in *MetaGME*, the UML variant GME uses for metamodels.

Metamodeling captures the key domain concepts and their relations, and maps them onto MetaGME language concepts, such as model objects, attributes, inheritance, containment etc. Visualization rules (icons, colors) are assigned. The metamodel is annotated with OCL *constraints* and cardinality rules enforcing well-formedness rules, such as "an output port must be connected to exactly one input port" or "each component must have a unique name".

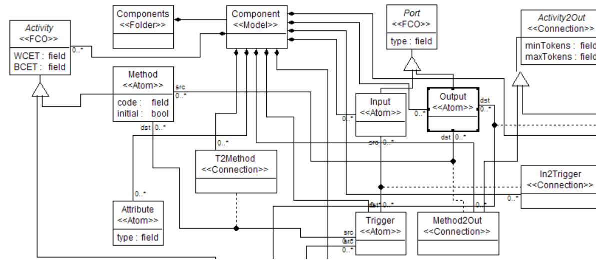
Figure 13 shows a portion of the SMOLES metamodel in the GME model editor.

Using the metamodel, GME synthesizes a visual modeling environment. This consists of a model editor (shown in Figure 12(a)), and metamodel-specific APIs for writing *model interpreters*, programs to process models.

Modeling SMOLES was straightforward: as shown in Figure 13, the main concepts of the language and their relations were captured, and mapped onto MetaGME entities.



(a) GME metamodel for Assemblies



(b) Part of GME metamodel for Components

Figure 13: Parts of the SMOLES metamodel in GME

Models (container objects) represent Assemblies and Components. *Atoms* (atomic objects) model Methods, Attributes, Ports, Triggers, Timers, Queues etc. A *Reference* was used to model QueueRef. Port-to-Port typed *Connections* represent dataflow links, as well as Input→Trigger and Trigger→Trigger or Method directed connections.

For the SMOLES *paradigm*, a model interpreter (code generator) was written to produce *implementations* for SMOLES systems. Implementations consist of source code (using the DFK platform API), with compiler and linker rules. The user-provided component method implementations are also composed into the resulting project. After compiling and linking, the user obtains a software implementation of the SMOLES design, ready to be executed.

Mapping SMOLES models onto UPPAAL Timed Automata

In the case study, the UPPAAL model checker is used for PSM analysis. UPPAAL implements model checking over *Timed Automata* [11].

The TA model is another abstraction of the system. Here, the assumed (real-time) properties of the components and the desired (real-time) properties of the system are captured. Note that we assume that the timing properties of components are known (and captured in the models), the system composition is known, and the analysis will be used to determine if the desired properties hold for the system. Since this work focuses on building the analysis model, questions regarding requirements modeling are only briefly discussed at the end of this chapter. Still, mapping requirements onto model checker queries is a very important part of the validation effort.

In SMOLES , timing properties of component methods are abstracted into $BCET$ and $WCET$ attributes. During the TA modeling of the DFK platform, the following significant abstractions were made:

- The data content of data-flow tokens is not considered, and only the number of tokens is represented. Therefore, transmission delays depending on token size are not modeled. The models demonstrated do not model token passing delays, although they could be trivially extended to do so.

- Each processing step (method invocation) delay is represented by a $[BCET, WCET]$ interval. Data-dependent execution delay (a realistic, but hard to model consideration) is not captured.

- A generalized data token production/consumption scheme is used. Methods are assumed to consume data tokens available on their Input ports when they are started, and produce output data tokens when they finish. The number of data tokens produced is represented by a $[min, max]$ interval.

The transformation applied to the model of the application results in a network of concurrent, strictly synchronized timed automata. There is one automaton for each component.

In addition, there is one automaton for the platform model that coordinates component interactions (the *Kernel*). Global variable declarations facilitate component-kernel interaction and synchronize the network of TA.

The component TA models are prepared by customizing a template or "skeleton" TA, as discussed below. Component models are reusable for different kernel models, as they contain no direct references to parts of the Kernel TA. This is due to the component – Kernel TA interface taking advantage of certain UPPAAL features (also discussed below in detail).

The Kernel is also generated by customizing a "skeleton", but it is unique for each system as it contains direct references to the components of the actual system. The runtime semantics including the behaviors for process scheduling, resource handling, concurrency and communication etc. are encoded in the Kernel TA. The Kernel TA is constructed similarly to the component models: the same quantities (e.g. time) are considered, and expressed in a similar manner. Modeling certain properties (delays, the runtime system's own resource requirements) becomes straightforward, since we use the same apparatus to express those as we used in the component modeling. This way, the Kernel becomes a "super-component", lending itself to the same verification techniques as those applied to the component model.

The kernel and component templates and construction rules (platform semantics) are encoded in the translation algorithm, i.e. they are *implicit*. For each platform to be modeled, a different translation algorithm has to be devised. The general scheme for constructing TA for platform components is as follows.

- The translation algorithm starts from a TA "skeleton" containing default states (e.g. `Start`, `Idle`, etc.).
- Then component and platform-specific states are added to the skeleton, for example, to represent each method invocation.

- Finally, state transitions are generated, implementing the fine details of the platform. The exact formulation of transition guards, synchronizers and reset functions is responsible for establishing the platform-specific behavior for the resulting network of timed automata.

Modeling component-kernel interaction

Construction of the analysis model TA begins with the declaration of system-wide constants and variables. These will be composed later into state transition guards in member TA. Component–Kernel interaction is also facilitated through these global entities.

For example, the following listing shows the global declarations for the TA modeling the simple system shown in Figure 12(a).

```
1  // Kernel parameters
2  const nProcs 1;         // number of processes in the system
3  const IdleTick 10;      // Kernel idle sleep
4  const DefChanSize 1;    // default dataflow buffer size
5
6  int [-1,nProcs-1] running:=-1;  // active process
7  int [0,nProcs] nInitializedProcs:=0; //number of procs. past init
8
9  // Timer 'Clock'
10 const ClockPeriod 10;   // Clock period
11 clock ClockClk;
12 // Node 'processing'
13 const processingPID 0;  // PID for the processing process
14 chan processingRun;     // synchronizing context switches
15
16 // dataflows
17 int [0,DefChanSize] ClockOut_processingInput:=0;
```

The declarations are constructed as follows.

2 In the UPPAAL model, a *Process* (concurrent TA) corresponds to each DFK node (plus one to the Kernel). Thus in the example system, `nProcs` is one (for the single node `processing`).

3 `IdleTick` is a kernel parameter, capturing a property of the single-threaded scheduler: the Kernel will sleep for 10 time units if no actor is available to run.

4 The next constant sets the default dataflow capacity (`DefChanSize`) to 1.

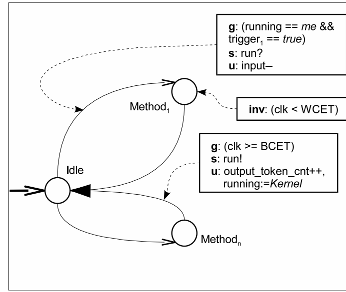
6 Integer variable `running` contains the PID of the process (node) currently executing on the CPU. This variable has an important role in implementing the component–kernel interface for the the single-thread scheduler model. Modeling of context switching will be discussed in detail below.

7 `nInitializedProcs` counts the number of nodes (processes) which have finished initialization (see the section about the Kernel TA for further discussion).

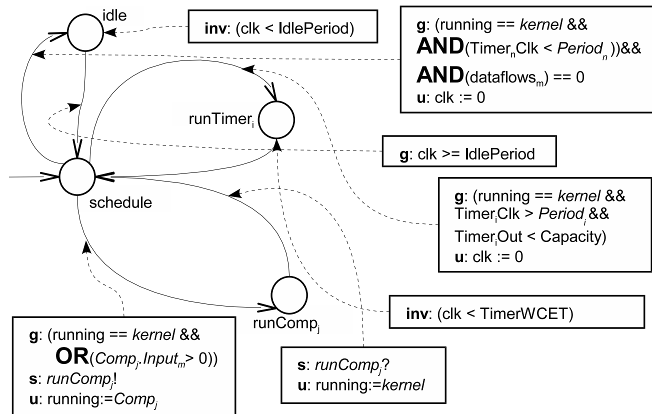
10-11 For each Timer, a `clock` variable is constructed, along with a `const` carrying its period.

13-14 Each SMOLES component (DFK node) is modeled by an UPPAAL TA (process), discussed in detail below. The scheduler model maintains a PID for each
(e.g. `const processingPID` for component `processing`). An UPPAAL *synchronization channel* is used to synchronize each component TA with the Kernel's scheduler automaton during context switches.

17 The last section of global declarations models dataflows as shared integer variables. In SMOLES / DFK, components interact through dataflows. Thus, both the source and destination components (as well as the Kernel) have to access the variable representing the dataflow. As mentioned earlier, dataflows are modeled as (bounded) integers describing the *number* of data tokens on the particular dataflow link. For each dataflow there is a corresponding a global integer in the UPPAAL model. The naming convention is: `SrcComponent.name + SrcOutputPort.name + '_' +`
`DstComponent.name + DstIntputPort.name` The range of the integer is the capacity of the dataflow link's buffer.



(a) TA Template for Components



(b) TA Template for Kernels

Figure 14: TA Templates for platform modeling

Component TA models

Component models are created by customizing a "skeleton" TA model describing the basic behavior. Figure 14(a) illustrates the concept (component initialization omitted). In UPPAAL , each transition is annotated with three attributes:

guard (g:) boolean function on local and global variables and constants. The guard has to be satisfied for the transition is to take place.

synchronizer (s:) the transition can be synchronized with transitions in concurrent TA through UPPAAL *channels*.

update (u:) atomic variable updates performed when the transition takes place.

Also, UPPAAL offers a feature called *local variable renaming*. When an UPPAAL process is *instantiated* (from a TA definition called *template*), a list of local variables gets assigned to globals. In the case study, component models have the following parameter list:

`Component`$_i$`(me, run, Input`$_0 \ldots$ `Input`$_n$`, Output`$_0 \ldots$`Output`$_m$`)`

`me` refers to the global `const` holding the component's PID. The second parameter is the kernel-component synchronization channel (called `run` locally, `Component`$_i$`.name + "Run"` globally). The rest of the parameter list maps the Input and Output ports within the component to global integers representing incoming and outgoing dataflows, respectively.

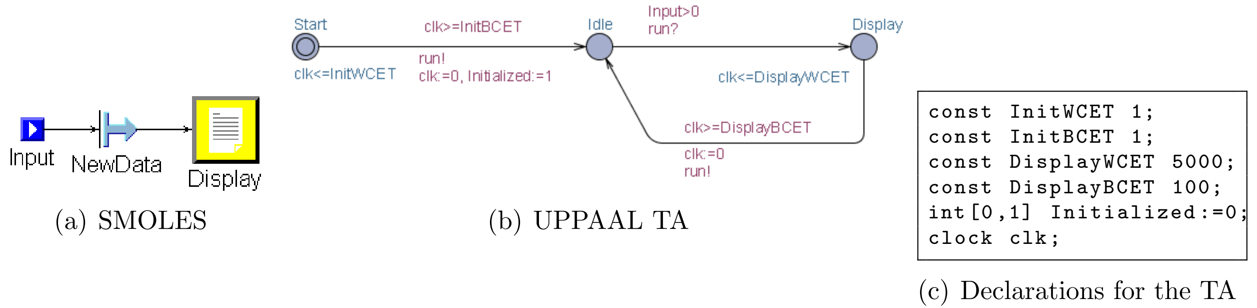
For example, the TA modeling component `processing` has the following *parameter list*: `(me, run, Input)`, and is instantiated as follows:

`processing:=Processing(processingPID, processingRun, ClockOut_processingInput);`

Thus, the condition `(running == me)` evaluates true in the local context of the component TA which is currently scheduled to run. (The scheduler sets variable `running` to the PID of the active component). The expression evaluates false in any other component's context, as `me` refers to their respective PID constant. Similarly, `Input--` means 'read a data token from dataflow `ClockOut_processingInput`' in the context of the `processing` TA instance `ClockOut_processingInput` is connected to `Input` port within `processing`, thus when the component process is instantiated, parameter `Input` gets assigned to global variable `ClockOut_processingInput`.

Using the `run` synchronization channel and the `running` variable in a component model is redundant. They are shown for the sake of completeness, since not all model checkers support both ways. If the model checker supports the rendezvous primitive (`chan` in UPPAAL) it should be preferred for synchronization (smaller global state space for the TA). In this case, the modeler still might keep and update the `running` variable, as it comes handy in formulating verification queries and analyzing simulator output.

The component automaton has a state representing the invocation of each SMOLES method. When no method is being invoked, the TA is in state `Idle`, as shown in Fig. 14(a).



(a) SMOLES

(b) UPPAAL TA

(c) Declarations for the TA

Figure 15: SMOLES model and Timed Automata for component *Processing*

The TA transitions to state $Method_i$ if the node is released by the scheduler (`running == me`), and $Method_i$'s trigger condition is satisfied. In SMOLES, trigger conditions are composed of Input ports, Triggers and *Triggers* connections connected to a Method (as seen in Fig. 13(b)). These elements visually represent a boolean condition on Input ports using *OR* and *AND* operators. This boolean condition is composed into the guard of transition $Idle \rightarrow Method_i$. In the guard, a port's "readyness" (i.e. whether there is a data token available on the port) is represented by the expression $Input > 0$, where *Input* is the local reference to the integer variable modeling the dataflow connected to the port. In the example (component `processing`), the trigger expression is as simple as `Input > 0`. (`Input` refers to `ClockOut_processingInput`). OR trigger conditions are modeled by multiple $Idle \rightarrow Method_i$ transitions. AND conditions are mapped into conjunctive guard expressions. In Fig. 14(a), updates "`input--`" and "`output_token_cnt++`" illustrate token consumption and emission. In the generated TA, a method is assumed to read one token from the port(s) triggering it. (i.e. the same port(s) present in the guard condition of the particular transition). Thus, an one-decrement update expression is generated for each. Token emission updates are generated based on the method→ Output port connections.

Figure 15 shows the component model in SMOLES and the corresponding UPPAAL TA. The component TA has two private variables: `clock clk`, and `integer[0..1] Initialized`.

In this model, synchronization with the scheduler model is done through `chan run` and `running` is not used.

The lifecycle of the component is as follows:

1. starts in state `Start`

2. goes to state `Idle` if `clk>=InitBCET`, synchronized with the corresponding transition in the scheduler model (`run!`). If this does not happen while `clk<=InitWCET`, an UPPAAL invariant, the model checker signals *deadlock*. On the transition, `clk` is reset and the `Initialized` flag is set.

3. from `Idle`, the automaton transitions to `Display`, if the SMOLES trigger assigned to Display is satisfied (as simple as `Input>0`, there's a data token on `Input`). On the transition, `clk` is reset and the token is removed from the buffer (`Input--`).

4. from `Display`, the automaton returns to `Idle` no sooner than `DisplayBCET`, synchronized with a transition in the scheduler model (`run!`).State `Display` also has an UPPAAL invariant `clk<=DisplayWCET`: the TA deadlocks unless the transition happens on time.

Analysis model for the Kernel

Figure 14(b) shows the template for DFK Kernel models (parts implementing component initialization not shown). This template models the single-threaded scheduler with the Timers implemented in-kernel. The central state is *schedule* It is implemented as an *committed* state in UPPAAL, i.e. the TA must immediately take a transition from here.

All outgoing transitions from *schedule* are guarded by condition `running == kernel`, i.e. the scheduler executes concurrently with no component. In the following text, $Out(Timer_i)$ stands for the dataflow originating in $Timer_i$ and $In(Comp_j.Input_k)$ is the dataflow terminating in $Input_k$ of $Component_j$.

From *schedule*, three kinds of transitions are possible:

Fire Timer $schedule \rightarrow runTimer_i$ when $Timer_i$ has expired:

$Timer_iClk > Timer_iPeriod$ and $Out(Timer_i)$ is available ($Out(Timer_i) < Capacity_i$)

Upon returning to *schedule*, $Timer_iClk$ is reset and the variable representing $Out(Timer_i)$ is incremented.

Execute Component $schedule \rightarrow runComponent_j$ when the node implementing $Comp_j$ is triggered:

$\bigvee_{k=0..m} Comp_j.Input_k > 0$ where m is the number of Inputs in $Comp_j$ (i.e. the TM_ANY DFK input trigger)

The transition is synchronized with $Comp_j$'s TA via $\texttt{runComp}_j$ and `running` is updated.

Idle $schedule \rightarrow Idle$ if none of the above conditions is satisfied. In this case the scheduler will *idle* (wait) for $IdlePeriod$ time units and re-evaluates the guards afterwards.

Note two important properties of the scheduler model:

a) If multiple transitions are enabled, the scheduler chooses one non-determistically. There is no *priority* in this model. The analysis model accurately captures this very important property. A SMOLES designer might have a (false) intuition that since Timers are implemented in kernel, they have priority over component executions.

b) The guard in *runComp* transition describes how much the platform scheduler "knows" about the components' internals. Here, DFK's TM_ANY node trigger is modeled. The platform scheduler does not know about the internal trigger condition implemented by the dispatch code in the node's `run()` method. The accurate modeling of such details of the DSML implementation over the platform is crucial for the validity of the analysis model.

   Having chosen a "better" platform or a "better" technique for the DSML→Platform mapping (i.e. the platform and DSML abstraction levels being closer to each other) might result in a more efficient implementation. For example, if $Method_m$ in $Component_j$ has trigger $Input_1 \wedge Input_2$, a data token on *either* $Input_1$ or $Input_2$ will trigger the execution of $\texttt{Component}_j\texttt{.run()}$ continuously, hogging the CPU. Even worse, if the method

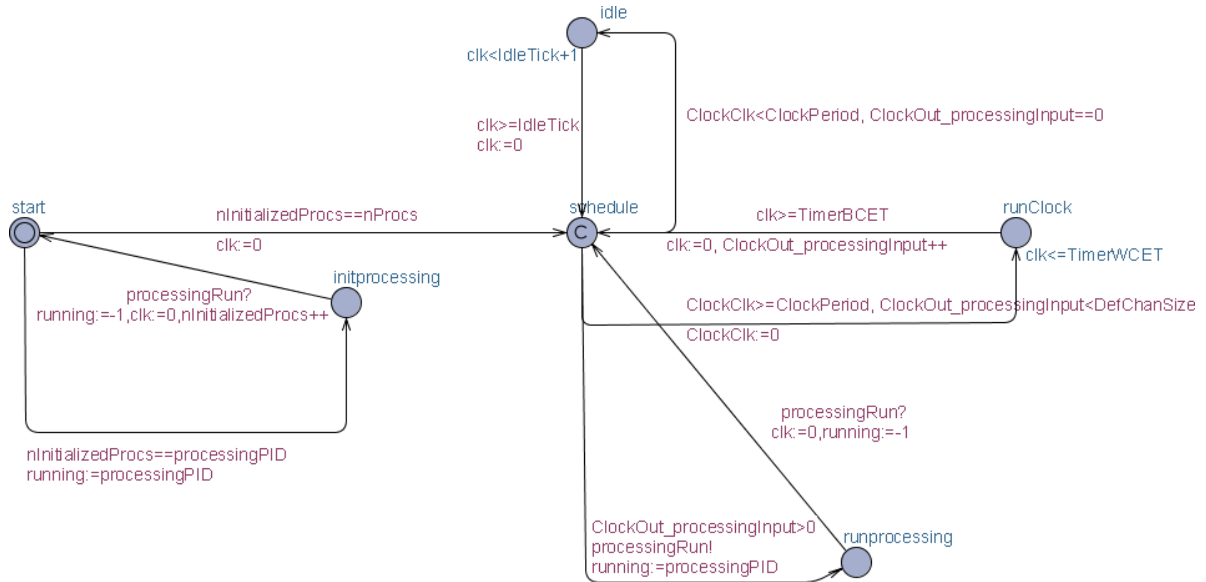


Figure 16: Kernel TA model for the system in Fig 12(a)

dispatch structure is not formulated carefully and reading from an Input with no token is attempted, the component execution might fail.

It is the very purpose of this work to provide means to identify such issues. The abstraction gap between DSML intentions and platform capabilities might be the source of many hard to identify problems. Platform modeling enables the formal study of this abstraction gap.

Figure 16 shows the TA generated for the Kernel, designed to work with the `processing` component TA in Fig. 15. In this example, upon finishing a component method execution the the Kernel TA updates `running` by resetting it to -1 (Kernel PID). In the generalized scheme shown in Fig. 14(b) this is done by the component TA. These two transitions are synchronized, thus the difference is irrelevant.

Implementing the transformation with GReAT

The following sections discuss the model transformation mapping SMOLES models onto UPPAAL TA analysis models. GReAT, the model transformation tool operates on GME

models, hence the output it produces is also a GME model. Thus, the first, step is extending the modeling environment with the UPPAAL metamodel.

Metamodeling UPPAAL

Modeling the UPPAAL language in GME consists of two steps:

a) Capture the concepts of the language and their relations (the UPPAAL realization of the TA concept).

b) Provide a GME export tool (*interpreter*) to save models in UPPAAL's native XML format (textual concrete syntax).

Both steps are routine tasks and each takes about a day or two's work for an experienced GME modeler. Figure 17 shows the metamodel, divided onto three *paradigm sheets* to avoid visual clutter. The following conceps are captured and modeled:

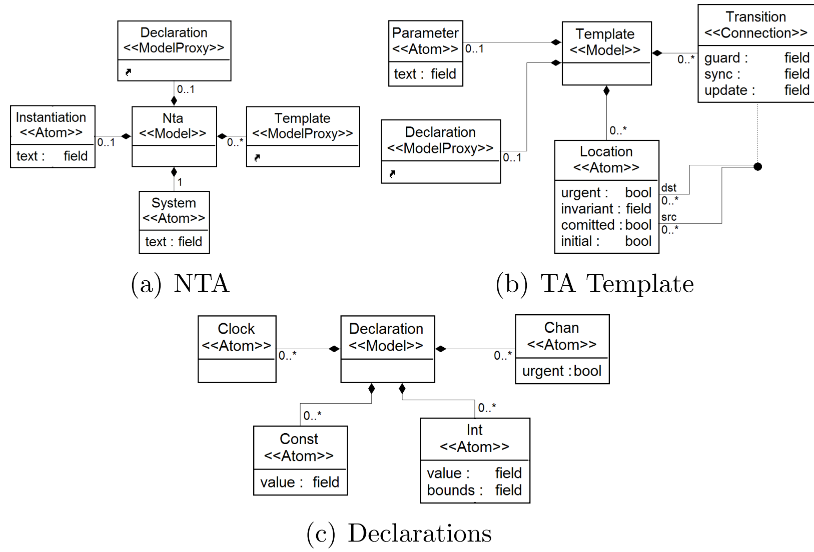
NTA *Network of Timed Automata* the top-level UPPAAL object, containing *declarations*, *TA templates*, their *instantiations* and the list of active TA in this particular *System*.

Declaration (both system-wide and local to TA templates) contains *Clock*, *Const*, *Int* and *Chan* variable declarations.

TA Template describes a TA with its *Locations* (states with invariants), *Transitions* between them with *guard*, *synchronizer* and *update* attributes. The template also contains local *declarations* and the instantiation *Parameter* list. *Parameters* are macro-like names, which get expanded into global variable names upon the template's instantiation.

Instantiations create TA *instances* based on templates by assigning global symbols to elements of the template's parameter list.

System contains the NTA's name and the list of TA instances active in the NTA



(a) NTA (b) TA Template

(c) Declarations

Figure 17: The UPPAAL metamodel in GME

The model interpreter traverses an UPPAAL model and prints it into XML format recognized by the model checker. Since the structure of the GME model matches exactly the native XML structure, this conversion is trivial.

Transformation setup and phases

GReAT is a model transformation tool, integrated with the GME framework. It operates on GME models, and internally uses the GME API to read the source model(s) and read-/write the target(s). As explained in Chapterr̃efp:Background, a GReAT transformation is specified as follows:

a) Transformation rules are specified in terms of source and destination metamodel entities. Thus, the first step is importing the source (SMOLES) and destination (UPPAAL) GME metamodels.

b) Define *crosslinks*, additional elements for the “unified” metamodel.

c) Using the visual language defined by the first two steps, specify the transformation rules and organize them into control flow structures.

d) *Configuring* the transformation by designating input and output files, adjusting options and designating the *startup* rule. In this case, the input is the model of a SMOLES system, and the output is an UPPAAL model.

This transformation reads a SMOLES model. During the transformation, temporary modifications are made to the *memory copy* of the source model. The source's permanent storage (model file) is not modified. A new, empty model (using the UPPAAL paradigm) is created at the beginning, modified and saved when the transformation finishes. Subsequently, an auxiliary *model interpreter* is used to export the resulting UPPAAL model into UPPAAL's native XML file format.

The actual SMOLES → UPPAAL transformation consists of three major phases:

1. Establishing the context by locating the top-level assembly in the model and creating the corresponding (yet empty) UPPAAL *NTA*.

2. The next step corresponds to a phase in the SMOLES → DFK code generator. The system is translated into an equivalent SMOLES system without assembly hierarchy. As DFK supports no hierarchy, the components within sub-assemblies need to be "projected" onto a "flat" component network.

3. Finally, this "flat" SMOLES system is translated into the analysis model. A TA is constructed for each component, and the Kernel model is built.

Figure 18 shows the top-level view of the SMOLES → UPPAAL transformation. The first rule to be executed is `TopLevelNTA`. As shown in the top left, it receives the two (input and output) model's top-level objects as inputs. For the very first rule, context is specified by the transformation configuration. Top-level objects in GME models are always *Root Folders*, and even empty models contain one.

The rule (expanded in left center) matches the two Root Folders, then matches all *Assemblies* folders within $SMOLESRootFolder$, finally matches all assemblies within (`Top`). Well-formed input models contain only one.

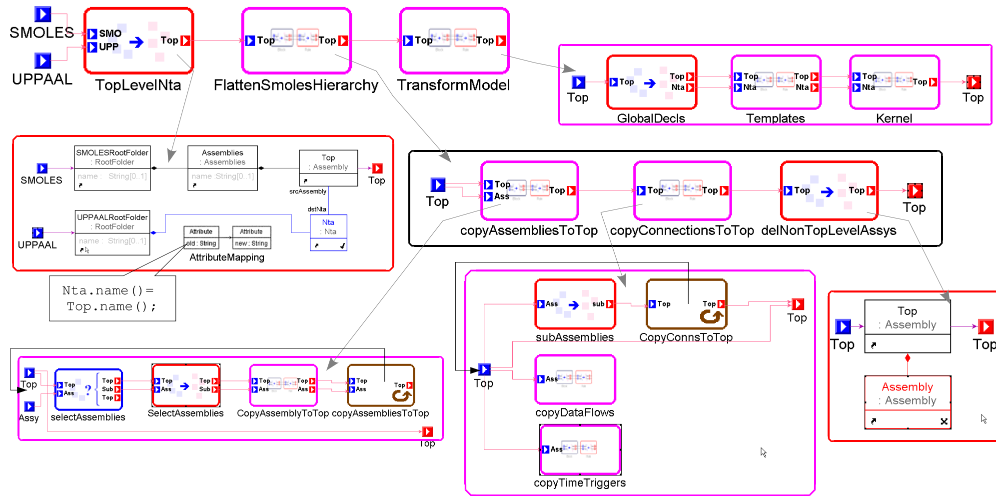


Figure 18: Top-level blocks of the SMOLES → UPPAAL transformation

For `Top` an *NTA* object is created into *UPPAALRootFolder*. (NTA - Network of Timed Automata, the top-level container of an UPPAAL system). *Create* operation is indicated by the checkmark in the bottom right of the NTA object. In addition, a *srcAssembly* → *dstNta crosslink* is created. Using the crosslink, it becomes easier to specify context for subsequent rules: only `Top` is passed along via ports, and the corresponding *NTA* can be found by matching the crosslink.

The rule also features an *AttributeMapping* box: these contain C++ code fragments (using the GME API) to manipulate model attributes. The code is executed for each match of the input pattern. In this case, the name of the newly created *NTA* will be set to match the corresponding assembly's name. Finally, the rule passes along SMOLES object `Top` for subsequent rules.

Flattening the assembly hierarchy

The next rule-block (`FlattenSMOLESHierarchy`) performs the second step listed above. It maps the SMOLES system onto an equivalent one containing components, atomic elements (e.g. Timers, Queues) and dataflows only.

The steps of this "projection" of the SMOLES system onto a "flat" one is shown on the right in the expanded rule-block (in Fig. 18). First, in rule-block `copyAssembliesToTop`, the assembly hierarchy is traversed recursively, and the contents of each assembly is "projected" into the top-level assembly (`Top`). Crosslinks are used to link the objects in hierarchy and their corresponding "images".

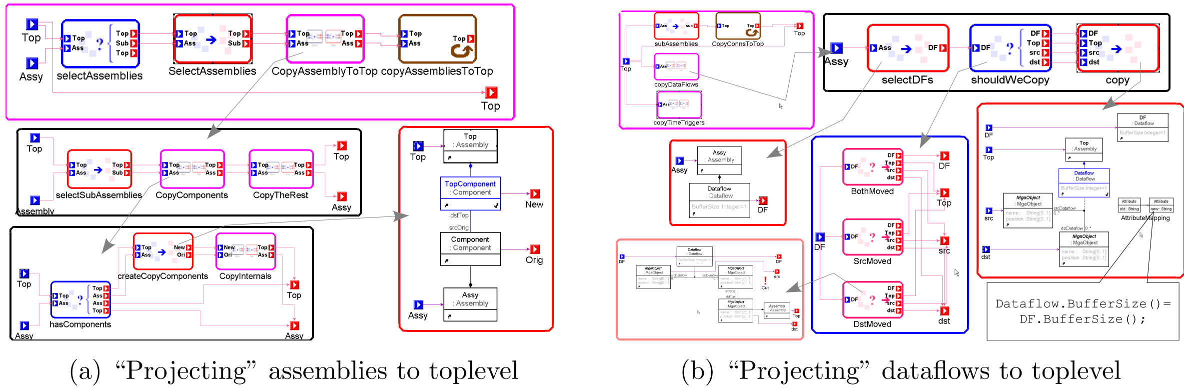
Then, the next rule block (`copyConnectionsToTop`) traverses the hierarchy again. It matches dataflow connections within non-toplevel assemblies and "projects" them into `Top`. Corresponding connection endpoints in `Top` are identified by the crosslinks created previously.

Finally, all sub-assemblies (with their connections and contained objects) are deleted from `Top` (rule `delNonTopLevelAssys`). The rule is shown in the lower right section of Fig. 18. It receives its context (`Top`) via the input port, and matches (and deletes) all contained assemblies. The *delete* operation is denoted by the small 'x' in the lower right corner of the icon. The transformation configures GReAT to work with a memory copy of the SMOLES input model, so the original model is not modified.

Rule blocks `copyAssembliesToTop` and `copyConnectionsToTop` are further expanded at the bottom of Fig. 18. Their context consists of 2 assemblies One is `Top`, the other one is used for traversal. At the start of the recursive rule block, both points to `Top`.

In `copyAssembliesToTop` (on the left) a *select* block checks whether *Assy* contains sub-assemblies. If not, the rest of the rule block is ignored. If yes, they are matched and "projected" to the top (`CopyAssemblyToTop`), and the rule-block contains a *reference* to itself (rightmost block), implementing recursion. Rule block icons with a circular arrow in the lower right corner indicate references (to other rule blocks). In this example, all references point "back" to the rule block containing them. This is how a recursive rule block can be specified in GReAT.

Rule block `copyConnectionsToTop` works similarly: "projects" all dataflows and time triggers (dataflows originating in Timers), and passes all sub-assemblies to itself recursively.



(a) "Projecting" assemblies to toplevel (b) "Projecting" dataflows to toplevel

Figure 19: Rule blocks flattening the SMOLES hierarchy

Figure IV shows the recursive rule blocks "projecting" sub-assemblies and their dataflow connections into `Top`. Again, the rule block contains a self-reference (`CopyConnsToTop`) for recursion. After matching sub-assemblies, the operation is decomposed into "copying" components and the "rest" (atomic objects such as Timers, Queues etc.). `CopyComponents` first checks whether the sub-assembly has components at all.If it has, the "image" component is created in `createCopyComponents`, and its contents are copied in `CopyInternals`. For each element copied onto toplevel, a $srcOrig \rightarrow dstTop$ crosslink is created, as seen in `createCopyComponents` (expanded rule on the right in Fig. 19(a)).

Figure 19(b) shows how the dataflow connections are projected into `Top`. Dataflows and TimeTriggers (dataflows originating in Timers) are matched in the "current" assembly. The dataflow needs to be copied if at least one of its endpoints had been projected into `Top`. This decision is implemented by the *Test* block `shouldWeCopy`, containing three test *cases*. One of the test cases is expanded in the lower left. Its context is the *Dataflow* object matched earlier. It matches the connection's two endpoints, with the destination having a $srcOrig \rightarrow dstTop$ crosslink (i.e. it was projected in `copyAssemblyToTop`, seen in Fig. 19(a)). If any of the three test cases matches, the dataflow connection is copied into `Top`. Rule `copy` implementing this is exploded on the right: it creates a new Dataflow connection instance in `Top`, with the two endpoints matched by one of the Test cases. In a Test structure, only one of the cases matches. This pattern will propagate its context (the

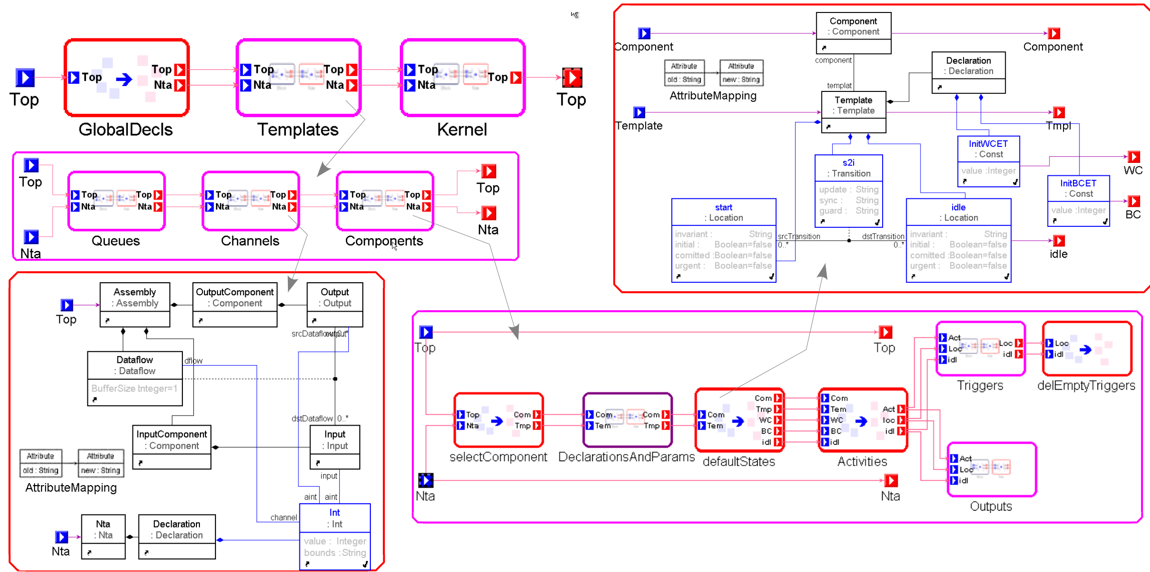


Figure 20: Rule blocks creating UPPAAL TA

matched objects) through GReAT ports to the next rule (`copy`). Traditionally, output ports (propagate the context of matched objects) is drawn on the right side in GReAT. Input ports *bind* matches of earlies rules to pattern elements. These ports are usually drawn on the left. Output → Input connections between rules and rule-blocks are also illustrated in Fig. 19(a)).

Creating the TA templates

Figure 20 shows the overview of the third phase, mapping the "flat" SMOLES network onto UPPAAL TA. First, `GlobalDecls` creates global variable and constant declarations for the UPPAAL model, such as `nProcs` and `running`. As explained earlier, the Component–Kernel interface relies on these. Sub-sections of the UPPAAL NTA are also created here (e.g. yet empty *System* and *Instantiation* declarations).

Next, `Templates` is responsible for the followings:

Queues Create a specific TA for each Queue. Queues allow multiple incoming and outgoing dataflow connections, and copy each incoming token onto each outgoing dataflow. The mapping consists of two steps. First, modeling the operation of the SMOLES → DFK

code generator, the Queue is mapped onto a SMOLES component with the necessary number of Input and Output ports, and a "library" method `copyToken`. Arrivals on each Input trigger the method, and in response it emits a token on each of the Outputs. An UPPAAL TA modeling the component is created according to the rules described earlier by Fig. 14(a).

Channels For each SMOLES Dataflow an UPPAAL global integer variable is created, as illustrated at the bottom in Fig. 20. All Dataflows are matched within the top-level assembly `Top`, along with their endpoint ports and components. Then, an *Int* variable is created in the UPPAAL TA. The AttributeMapping box names the variable according to the naming convention described earlier. (`SrcComponent.name + SrcOutputPort.name +'_'+ DstComponent.name + DstIntputPort.name`)

Components creates a TA *template* (definition) for each component, as seen on the bottom right.

`DeclarationsAndParams` creates local variable and *const* declarations (e.g. `clock clk`). `DefaultStates` creates the TA "skeleton" by adding `Start` and `Idle` states. Then, `Activities` generates states modeling method invocations, and rule blocks `Triggers` maps trigger conditions onto transition guards as follows:

1. An `Idle`→ $Method_i$ transition is created for each $Trigger_j \rightarrow Method_i$ connection.
2. For each $Input_k \rightarrow Trigger_j$, $Input_k > 0$ is ANDed to this transition's guard.

This implements the "ANY incoming Triggers may trigger a $Method_i$, and ALL Inputs connected to a Trigger have to have token(s) in order for the Trigger evaluate true" semantics of SMOLES .

Rule block `Outputs` models token emission by appending `Output_port++` update statements to $Method_i \rightarrow Idle$ transitions, derived from the SMOLES model.

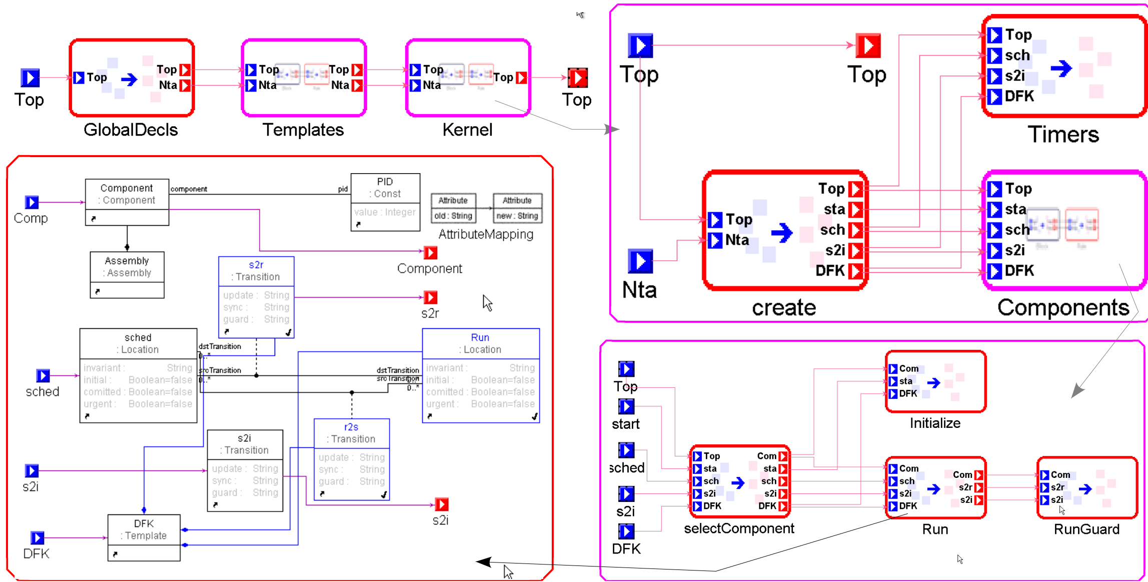


Figure 21: Creating the Kernel TA

Figure 21 shows the construction of the Kernel TA. First, the TA itself is created (`create`). Then, for each $Timer_i$ a $RunTimer_i$ state is added with the appropriate transitions in `Timers`. (In single-threaded DFK, Timers are implemented in kernel). For each $Component_c$, a state $RunComponent_c$ is added as well, as shown in the bottom and left of the figure. The guard conditions for $RunComponent_c$ are composed in a separate rule. This rule (`RunGuards`) matches all $Inputs_j$s and composes $\bigvee Inputs_j > 0$ for transition $schedule \rightarrow RunComponent_c$ as described earlier in the section about the component TA "skeleton", and shown in Fig. 14.

Verification via model checking Timed Automata

Having the system model converted to a TA network enables using a model checker to verify properties of the system. First and foremost, TA can be used to verify timing (and real-time) properties. Since the verifications possible depend both on the platform model and the model checker's capabilities, no general recipe can be given for formulating the verification queries.

In general, this is certainly an area of further research, namely how to express those properties and requirements in the high-level modeling apparatus, and how to propagate these expressions through the different abstractions of the system. The focus of this research is capturing end system properties due to platform characteristics, which is a hard enough problem in itself. Thus, this work provides only a few generic examples to show how to formulate verification queries, without providing a general recipe.

Model checkers typically verify Boolean queries only. The designer is often interested in numerical answer. One typical example is: "What is the fastest timer-rate the system can safely operate at ?" Model checkers can usually verify questions like: "Is the system safe with a clock of 5 Hz ?" so deducing the actual highest clock-rate is not straightforward. *Parametic analysis* ([28]) is an interesting attempt to solve such problems, although very limited in scope. Alternatively, the designer might iterate through a sequence of verification queries to approximate the answer.

In the following examples,.typical properties will be highlighted designers might be interested in. Basic ideas for expressing and verifying these properties are presented. These ideas could be used in the context of models similar to the ones used in the case study.

Checking for latency A typical verification question: "Is the delay between invocations of a certain method larger than n time units ?" To answer this question, we have to extend the generated TA trivially by adding a dedicated clock (`myclock`) and reset it at each invocation. Then, a model checker query can be formulated such as (in CTL) "$E\Diamond(myClock > 5)$", meaning does it ever happen that the clock value exceeds 5 ?

Checking for resource usage and conflicts In the case study, each activity (method) was annotated by its WCET and BCET. Similarly, they could be annotated by resource requirements other than CPU time. The Kernel TA could be extended with accounting for that resource through a shared variable across the TA network. Dynamic memory allocation is a good example: each component TA increments/decrements the size of

the dynamic memory pool upon entering/leaving states corresponding to method invocation. With the model checker, the designer is able to verify the system's maximum memory consumption.

Bounding the number of tokens on dataflow links The case study uses a dataflow oriented runtime platform. Many system properties (such as deadlock) can be verified by formulating queries on the number of tokens per buffers. For example, if the number of tokens ever exceeds one on a link coming from a periodic clock, it means that the destination component was not able to process the clock token on time and the system missed a clock tick.

Checking schedulability Although the definition of schedulability is general, non-schedulability can manifest itself on different platforms and systems different ways. Fortunately the question (since it is Boolean) usually can be formulated within the model checker. In the context of TA, it can be usually formulated as a deadline-violation statement. In systems such as the ones presented by the case study, the maximum number of tokens on certain dataflows carries information on schedulability. If this number ever gets greater than a significantly large constant, it can be concluded that the system produces more tokens than it consumes. Thus, it will eventually it deadlock.

Conclusions

In this chapter, derivation of a platform-level analysis model for a high-level DSML was presented. The analysis model captures the behavior of the system described by the DSML, as implemented over a particular platform. The demonstrated technique enables accurate modeling of platform properties / limitations influencing key properties of the end system.

Using the platform concept narrows abstraction gaps during the implementation and helps separating concerns. Platforms define a set of well-defined services and resources for the DSML implementor. Platforms also facilitate *reuse* and accelerate development by

implementing commonly required services over a wide range of hardware / OS / middleware. By choosing the appropriate platfrom abstraction, the designer can leverage on the availablity of competing platform implementations.

This work focuses on using platform models for analysis. For analysis models, designers choose the appropriate level of abstraction, which might cross technology boundaries for better accuracy / focus. Designers are able to incorporate key features into the analysis model, such as the way multithreading / multitasking is emulated on uni-CPU systems. Modeling resource allocation schemes and limitations imposed by the OS also becomes straightforward using this approach.

It is also important to emphasize that this approach *automates* the construction of analysis models. Thus, the outlined approach enables the automated, design-time verification of computer-based systems *as implemented over given platforms*. By choosing the appropriate level of abstraction, platform modelers excercise control over the complexity of the resulting analysis model.

This chapter demonstrated *implicit* platform modeling: the analysis model of the platform was embedded in the transformation specification. For the implementation, a general-purpose graph transformation language was used. The result was a large, hard to manage monolithic transformation, spanning from the DSML to the analysis level, containing the platform information implicitly.

There are several ways to make this transformation more feasible.

1. Breaking up the DSML→Analysis transformation into a DSML→Platform→Analysis chain. This way, the first transformation becomes reusable for different platform mappings.

2. Using a *explicit platform models*, which can be "plugged into" (i.e. input to) the transformation, instead of being a part of it.

3. Instead of a general-purpose model transformation language, specify the mappings in a more abstract formalism (platform modeling language), thus making them simpler.

The next chapter explores these possibilities and introduces a language for *explicit platform modeling*.

CHAPTER V

EXPLICIT PLATFORM MODELING

The DSML→Platform→Analysis transformation chain

The previous chapter presented the idea of platform modeling. The platform concept was introduced, and the benefits of having a PSM analysis model argued for. A case study was also presented to show how this PSM analysis model can be generated, using model transformation on the design model. In the demonstrated approach, the platform knowledge was implicitly encoded into the transformation. If the design has to be evaluated over different platforms, a transformation needs to be devised for each. In other words, the previous chapter defined a transformation $\mathcal{T}_{P_i}$ for a specific platform P_i:

$$A_{(P_i,j)} = \mathcal{T}_{P_i}(\langle PIM \rangle_j)$$

$\langle PIM \rangle_j$ is a specific design, P_i is a given platform, and $A_{(P_i,j)}$ is the corresponding, platform-specific analysis model (i.e. it models the design as implemented over platform P_i). In order to verify the design as implemented over a different platform P_k, a different $\mathcal{T}_{P_k}$ needs to be specified. As we had seen in the previous chapter, such transformations are rather complex. Verifying $\mathcal{T}_{P_i}$ is difficult, as it would be for $\mathcal{T}_{P_k}$.

One of the reasons behind the complexity of $\mathcal{T}_{P_i}$ is that it implicitly contains the "implementation transformation" $\mathcal{P}_i$:

$$\langle P_i SM \rangle_j = \mathcal{P}_i(\langle PIM \rangle_j)$$

This can be observed on case study as well: the SMOLES → UPPAAL transformation had two, well separated phase. In the first one, the SMOLES model was implicitly mapped to a DFK-level model. The second phase mapped this model onto UPPAAL TA templates.

Thus, formally separating $\mathcal{T}_{P_i}$ into "platform" and "analysis" parts could actually be quite useful:

$$A_{(P_i,j)} = \mathcal{T}_{P_i}(\langle PIM \rangle_j) = \mathcal{A}_{P_i}(\mathcal{P}_i(\langle PIM \rangle_j)), \text{ where}$$

$$(\langle P_i SM \rangle_j = \mathcal{P}_i(\langle PIM \rangle_j),$$

$$A_{(P_i,j)} = \mathcal{A}_{P_i}(\langle P_i SM \rangle_j)$$

$\mathcal{P}_i(\langle PIM \rangle_j)$ is the "implementation" transformation, generating $\langle P_i SM \rangle_j$ (a PSM specific to P_i) for $\langle PIM \rangle_j$ as implemented over P_i. $\mathcal{A}_{P_i}$ is the analysis mapping: it assigns semantics to platform-level elements of $\langle P_i SM \rangle_j$ by mapping them into the analysis domain, such as constructing a corresponding Timed Automata.

This approach relies on the fact that $\langle P_i SM \rangle_j$ *refines* $\langle PIM \rangle_j$ and no information is lost by the translation.

The DSML → Platform mapping

Decomposing $\mathcal{T}_{P_i}$ into $\mathcal{P}_i$ and $\mathcal{A}_{P_i}$ offers sevaral advantages. First, $\mathcal{P}_i$ might be already available, since in the context of MDA, $\mathcal{P}_i$ is actually the model transformation used to implement the system!

A good analogy for this transformation is a compiler. Similar to a C++ compiler mapping object-oriented C++ code onto CPU-specific machine code instructions, $\mathcal{P}_i$ maps a PIM onto a $\langle P_i SM \rangle$, a platform-specific refinement of the design model. The difference in the level of abstraction (C++ objects vs. assembly instructions) illustrates the typical DSML–Platform abstraction gap. Just like a C++ compiler, this transformation is often rather complex and hard to verify.

Using this approach has one more crucial advantage. Using the very same transformation in both the synthesis (implementation) toolchain and in the analysis chain further guarantees

the validity of the analysis model. This way, the analysis model can be used to analyze (debug) even the DSML → Platform "compiler" transformation. Compilation errors will be reflected in the analysis model, and can can be traced back.

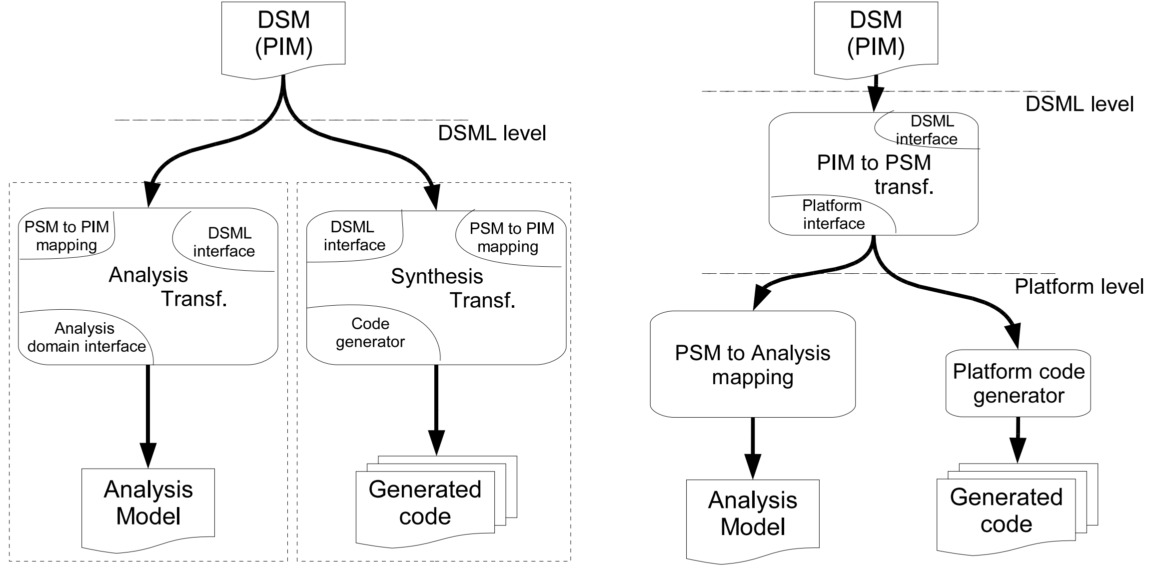
For example, the SMOLES → DFK transformation mentioned in the case study was implemented in code, as a GME *model interpreter*. The interpreter generated C++ source code, implementing the SMOLES system using the DFK API. In a different project [9], SMOLES was used to program LEGO Mindstorm robots. For this project, DFK was ported to the Java-based `lejos` operating system running on the robots.

Thus, the code-generating interpreter was rewritten to produce Java code. The basic structure / class hierarchy of the generated code remained the same, but the differences between the C++ and Java versions still required careful re-engineering of the interpreter, going through the develop-test-debug-fix cycle again.

Having had a common *DFK platform* layer in the modeling framework would have eliminated this problem. In this case, a SMOLES → DFK model transformation produces a platform-level model of the system. "Printing" program code (C++ or Java) based on a DFK-level model (whose structure follows the structure of the code to be generated) is a much easier task than doing it so based on the SMOLES system, defined at a different level of abstraction. Figure 22 compares the two approaches. Rounded rectangles represent model transformations, and their size in the figure is indicative to the particular transformation's complexity. As we can see, having an explicit platform layer (and a corresponding PIM→PSM transformation) has two advantages:

1. The analysis model is generated from the same PSM as the implementation (code). Thus, its relevance to the implementation is further ensured.

2. The code generation step becomes much simpler, as the structure of the platfrom-level model usually mirrors the structrure of the code.



(a) Model synthesis with no platform abstraction (b) Model synthesis with explicit platform abstraction

Figure 22: Advantages of having an explicit platform abstraction in MIC

The Platform→Analysis mapping

As we had seen earier, in many projects using MDA, the $\mathcal{P}_i$ "compiler" transformation is ready and available, and it delivers the $\langle P_iSM\rangle_j = \mathcal{P}_i(\langle PIM\rangle_j)$ mapping.

Next, the analysis transformation $\mathcal{A}_{P_i}$ needs to be provided. In general, we seek a transformation $\mathcal{T}$ such that

$$A_{(P_i,j)} = \mathcal{T}(\langle PIM\rangle_j, \mathcal{P}_i, \mathcal{A}_{P_i})$$

where $\mathcal{A}_{P_i}$ is the $P_iSM \rightarrow \mathit{Analysis}$ transformation and $A_{(P_i,j)}$ is the analysis model (of design j over platform P_i).

Let us make two observations here: First, generally the analysis $\mathcal{A}_{P_i}$ transformation is less complex than $\mathcal{P}_i$, since the abstraction gap it needs to bridge is less wide. Basically, $\mathcal{A}_{P_i}$ has two tasks:

1. Map each platform-level entity onto an analysis language structure (such as the DFK dataflow → UPPAAL variable mapping, or DFK node →"component" UPPAAL TA template mapping)

2. Define composition rules for the analysis language structures produced in the previous step. Composition will "configure" the set of analysis-language structures according to the "wiring" (component interconnections) of the specific DSM. In the case study, the corresponding step is the instantiation of the component TA templates using global variables to facilitate interaction.

$\mathcal{P}_i$ is specified using a general-purpose model transformation formalism, since it might be very complex. Capturing the analysis mapping can be done using a more specific formalism. This formalism should provide native support for the above patterns (e.g. instantiating "skeleton" analysis language structures, extending and composing them). By having a platform modeling language at higher level of abstraction (than a general-purpose GT language) the platform modeler's task is made easier. Additionally, using a stricter, special-purpose mapping formalism helps studying (and proving) properties of the transformation.

Thus, instead of transformation $\mathcal{T}$ let us seek $\mathcal{T}'$:

$$\mathcal{A}_{P_i} = \mathcal{T}'(\langle PIM \rangle_j, \mathcal{P}_i, \mathcal{M}_{P_i})$$

where $\mathcal{M}_{P_i}$is referred as the *explicit platform model*, and $\mathcal{T}'$ produces the analysis transformation $\mathcal{A}_{P_i}$. Technically, $\mathcal{T}'$ maps the platform model $\mathcal{M}_{P_i}$ onto the general-purpose model transformation formalism used for $\mathcal{P}_i$ and integrates the two.

Transformation $\mathcal{T}'$ has to be specified (and verified) only once. This is quite important, as the validity of analysis results obtained by $A_{(P_i,j)}$ depends on the correctness of $\mathcal{T}'$.

For $\mathcal{T}_{P_k}$, both the validity of the encoded platform model *and* the correctness of the DSML→Analysis domain mapping have to be verified. An explicit platform model M_{P_i} provides a clean separation for these concerns, and ultimately reduces the effort involved. This approach enables the specification of explicit platform models (M_{P_i}) separately. Thus, platform experts can compose platform model libraries, which in turn enable designers to evaluate their designs over different platforms in an automated manner.

This chapter introduces *explicit platform modeling*, and describes the generic transformation $\mathcal{T}'$, and introduces a framework for building explicit platform models.

Platform metamodeling for synthesis

The first step towards providing both $\mathcal{P}_i$ and $\mathcal{M}_{P_i}$ in the design→synthesis chain is to metamodel the platform within the modeling framework. Metamodels define the "modeling language", thus enable the construction and use of platform-specific models within the framework. Metamodeling captures the concepts of a domain (the platform in this case) and their relations, and assigns abstract and concrete syntax to the model elements.

Establishing the appropriate level of abstraction becomes very important here, as the designer / modeler faces a tradeoff:

a) Either the metamodel captures the platform concepts very precisely, working exactly at the level of the platform provider's abstraction. This results in a "generic" platform metamodel. This metamodel can be used for different DSMLs, if they are to be implemented over this platform.

   In this case, writing the code generator / system synthesis tool becomes trivial.The DSML → Platform transformation has to bridge a wider abstraction gap thus it becomes more difficult.

b) Alternatively, the metamodel may reflect the particular needs of this particular DSML → Platform transformation, and omit certain platform concepts or model them at an abstraction level closer to the DSML. Thus, the platform metamodel becomes specific for this plarticular DSML → Platform transformation.

   In this case the DSML → Platform transformation becomes simpler, and the code generator has to map these concepts down to platform level.

This problem arises because for implementing a certain DSML, the platform services are used in a certain configuration (e.g. in the SMOLES → DFK implementation, a component's

`run()` method always contains a dispatch structure matching the SMOLES trigger). Either this particular structure / configuration is modeled at the level of platform primitives and repeated for every system, or the platform metamodel implicitly assumes the presence of this structure, and does not model it in depth.

For example, a C→Assembly compiler developer may choose to work with a "generic" assembly metamodel, capturing each individual instruction. Alternatively, knowing that certain C language entities (such as function entry / exit) map onto certain well-defined assembly instruction sequences, one may choose to model these concepts explicitly in the assembly metamodel. In the first case ("generic" assembly metamodel), the C→Assembly compiler has to generate these sequences, thus it becomes more complex. In the second case, the compiler is simpler, and the assembly model "export" tool expands these elements into instruction sequences. Also, in this case the assembly metamodel might become"C-specific". The decision which case to take is made after considering the advantages of having a simpler "compiler" transformation over being generic. It might also be possible to achieve both objectives at the cost of considerable extra work (e.g. providing a macro assembly language).

Let us examine this tradeoff in detail for the case study: In SMOLES components, elaborate trigger conditions select one method for invocation. In DFK, the node triggers are much simpler (TM_NONE, TM_ANY, TM_ALL), and there is only one method (`run()`) that gets invoked. The implementation bridges this gap by using TM_ANY and synthetizing dispatch code structure into `run()`, invoking the appropriate method. This is one possible implementation of the SMOLES trigger condition over the DFK platform.

If the DFK metamodeler chooses the first option and prepares a DFK metamodel exactly matching DFK concepts, he / she models the three simple DFK triggers. All DFK models produced through the SMOLES → DFK transformation use TM_ANY , and the other two will never be used. Additional means have to be provided to model the "internals" of method `run()`, in order to model the dispatch structure, as it is a very important part of the system

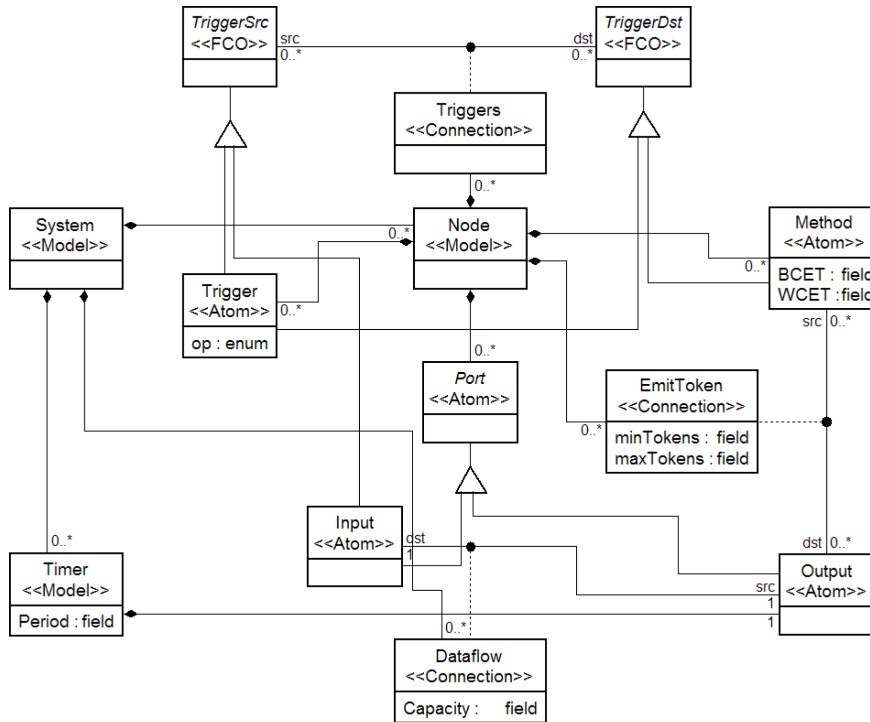
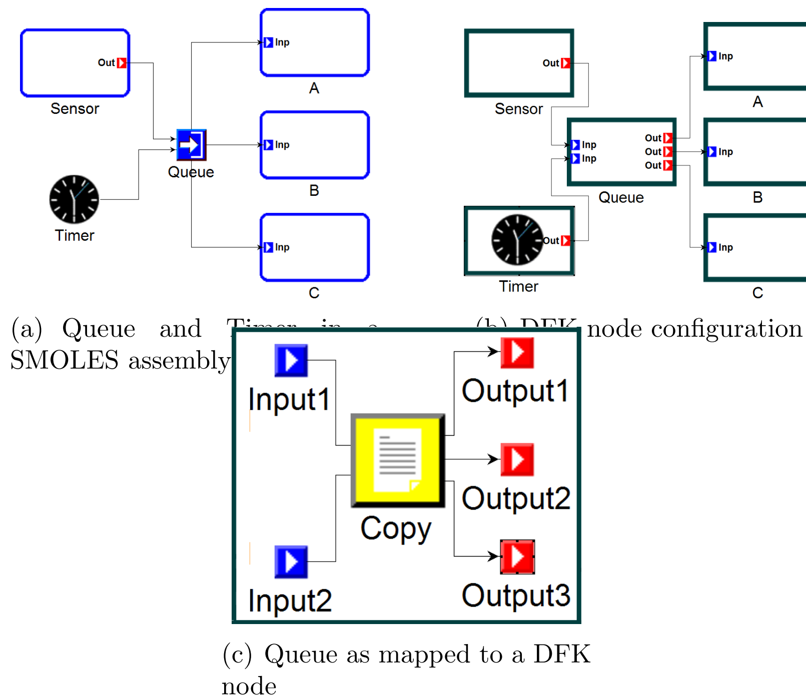


Figure 23: The DFK metamodel in the case study

under construction. For a general DFK metamodel, this means modeling the internal `if` instructions at some level of abstraction. For ultimate precision, the C++ language has to be modeled.

Figure 23 shows the DFK metamodel prepared for the case study. This metamodel was prepared to fit for the SMOLES → DFK translation, rather than being a generic DFK model. One more reason for this decision was the need to model a special *Timer* node. This is not a native concept of DFK but important for implementing SMOLES . In order to implement an efficient Timer, it is necessary to modify the DFK kernel anyway. A generic DFK metamodel does not describe this property.

As shown in the figure, this metamodel uses trigger conditions similar to those used in the SMOLES metamodel. The `run()` method is not modeled explicitly. Using this metamodel, the model interpreter generating the implementation source code has to process the trigger conditions, and synthesize the corresponding dispatch structure for `run()`. Also, as mentioned earlier, this metamodel cannot describe every DFK system, since it captures



(a) Queue and Timer in a SMOLES assembly

(b) DFK node configuration

(c) Queue as mapped to a DFK node

Figure 24: Illustrations for the SMOLES → DFK transformation: Queue and Timer

one particular configuration. For example, this metamodel cannot capture negative trigger conditions, such as "fire $Method_1$ if there is data on $Input_1$ but no data on $Input_2$". The native DFK API allows constructing such conditions.

Example: The SMOLES → DFK transformation

Using the "specific" DFK model, mapping SMOLES onto DFK is an easy task, consisting of the following steps:

1. Flattening the assembly hierarchy. This problem has been solved in the previous chapter, and the rule block can be re-used here.

2. Mapping Timers and Queues. As seen in Fig. 24, both Timers and Queues are mapped onto DFK nodes. Timers are simply mapped into DFK timer nodes, and no internal structure is modeled (apart from the presence of an Output dataflow port). For Queues, a "regular" DFK node is created, with the necessary number of Inputs and Outputs,

and a library method `Copy` within. This method is triggered by token on any of the input ports, and it will emit one token on each Output in response.

3. Mapping components. This step is rather simple: SMOLES Inputs, Outputs and Methods map onto DFK Inputs, Outputs and Methods, respectively. Trigger mapping is also straightforward, as shown in Fig. 25. In SMOLES , a Trigger evaluates true if *all* connected Inputs have a token present, and a Method gets triggered if *any* of the connected Triggers evaluates true. Thus, a SMOLES Trigger is mapped onto a DFK AND Trigger, with the respective Input→Trigger *Triggers* connections. For each SMOLES Method, a DFK OR Trigger→Method sequence is created, and incoming trigger connections attached to the OR trigger.

   The GReAT rules shown in Fig. 25(c) implement this mapping:

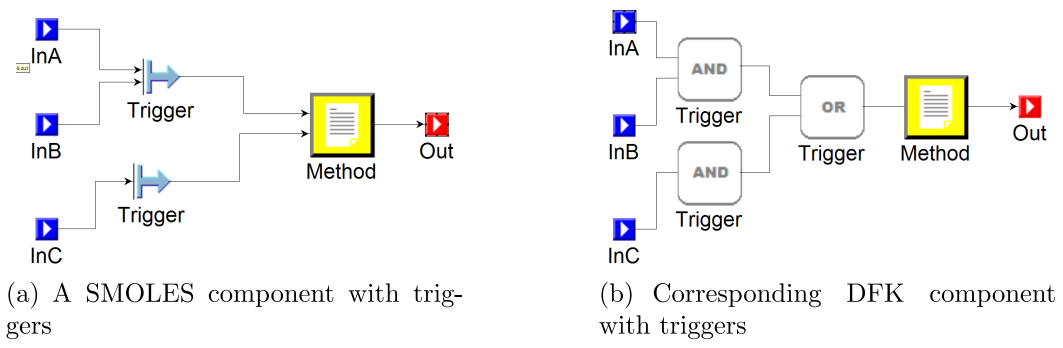
   a) First, for each SMOLES Method within a Component, a DFK Method (with the attached OR triggger) is created into the corresponding Node. The two Methods are associated using a SMOLES → DFK *crosslink*.

   b) Then, for each SMOLES Trigger→Method connections a DFK AND trigger is created and connected to the OR trigger attached to the corresponding DFK Method. The corresponding DFK Method is found by following the crosslink.

   c) Finally, for each SMOLES Input→Trigger connections, the corresponding DFK Input is created, and connected to the appropriate DFK Trigger.

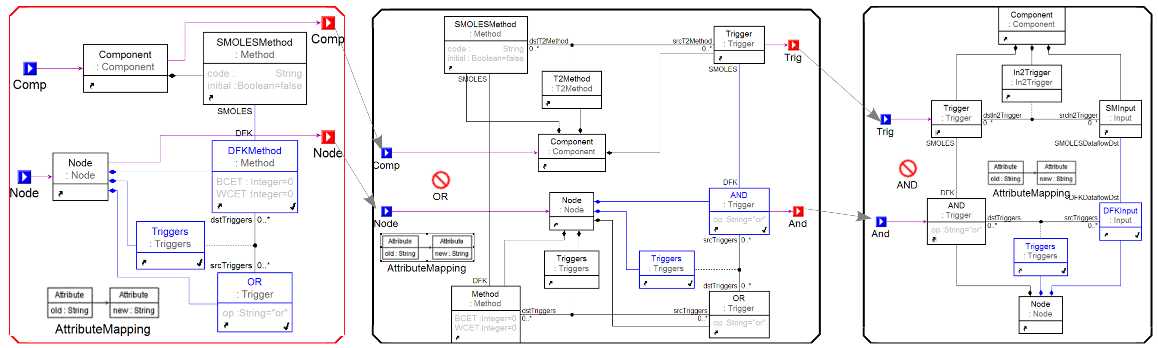
As explained earlier, this mapping is easy because the language chosen to model DFK is very specific for the SMOLES → DFK implementation. In this case, it is the task of the code generator to produce blocks of code for entities not explicitly represented in the PSM. Such entities include nodes' `run()` method and the code within implementing method triggers.

During the course of this research, PIM→PSM transformations have also been prepared for case studies where the platform metamodel reflects the native structure of the platform.



(a) A SMOLES component with triggers

(b) Corresponding DFK component with triggers



(c) GReAT rules implementing the mapping

Figure 25: SMOLES → DFK trigger mapping

In these cases, the transformation implementing the DSML→Platform mapping may become rather complex, as illustrated by the papers published about such transformations: [52], [55].

The Platform $\rightarrow$ Analysis transformation

This chapter introduces explicit platform modeling, and an important part of this effort is to provide a formalism for capturing the Platform $\rightarrow$ Analysis mapping $\mathcal{M}_{P_i}$ in transformation $\mathcal{T}'$:

$$\mathcal{A}_{P_i} = \mathcal{T}'(\langle PIM \rangle_j, \mathcal{P}_i, \mathcal{M}_{P_i})$$

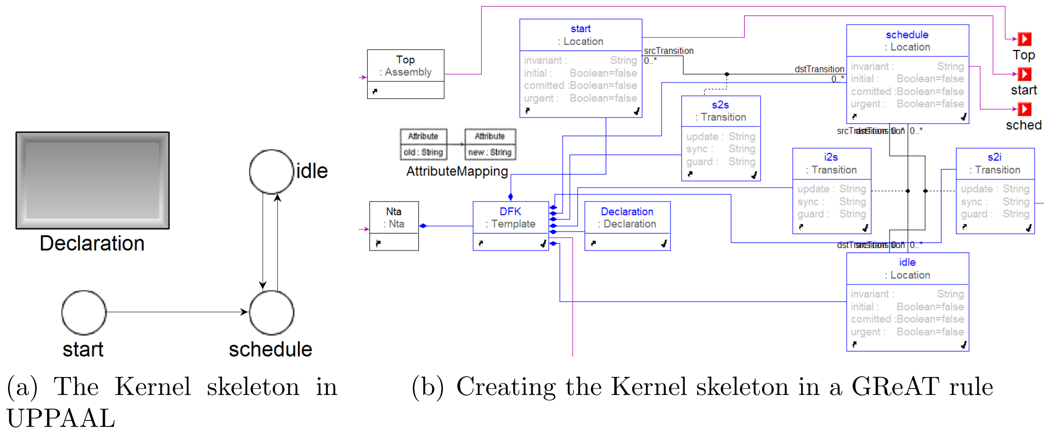
$\mathcal{M}_{P_i}$ captures the information necessary to map $\langle PIM \rangle_j$ onto $\mathcal{A}_{P_i}$ (the analysis model specific to platform P_i).

A platform model specifies a mapping from platform-level primitives (e.g. instruction sequences or platform-level component descriptions) to analysis model structures (such as timed automata fragments), which model the behavior of the design implemented over the particular platform. According to the above formalism, platform models define the second transformation in the DSML$\rightarrow$Platform$\rightarrow$Analysis chain.

As we have seen earlier, the DSML $\rightarrow$ Platform transformation (the "compiler") might be very complex, as shown in the case study published in [52] and [55]. Due to this complexity, this transformation is specified using a general-purpose model transformation language, such as GReAT.

However, the Platform $\rightarrow$ Analysis transformation is typically simpler. As demonstrated in the previous chapter, this transformation takes a "template" analysis model for the kernel or for the components, and "expands" it with design-specific details (e.g. additional states and transitions). For this simpler transformation a more specific, more abstract formalism — resulting in a simpler, higher-order transformation language — could be used. Using a more specific formalism offers the following advantages:

1. Higher level of abstraction makes the resulting platform models simpler and smaller.
2. Operations or structures common in this area can be captured and supported explicitly. This leads to a simpler, more concise language.



(a) The Kernel skeleton in UPPAAL

(b) Creating the Kernel skeleton in a GReAT rule

Figure 26: Creating an analysis model fragment in GReAT

For example, GReAT supports different control structures for the explicit specification of rule sequencing. This enables the use of GReAT in many different graph transformation scenarios. The price we pay is the sophisticated (and in many situations awkward) way bound objects are passed between rules. If matched objects from a rule are being used by more than one subsequent rule, the same set of objects have to be propagated to all "downstream" rules, regardless of which objects are actually used. This leads to either overly complex rule interfaces (and rules with many unused pattern elements), or the proliferation of "filter" rules whose only purpose is to streamline the interfaces of other rules.

Furthermore, creating / attaching complex analysis model fragments ("skeletons") is difficult in GReAT, because in GReAT one always works at the metamodel level.

Figure 26 illustrates this problem: Fig. 26(a) shows an UPPAAL model fragment (the skeleton for the Kernel automaton), and Fig. 26(b) shows the GReAT rule creating this structure.

In GReAT, each (new) object has to be captured at the metamodel level, along its parent and containment relation. For specifying a simple connection, 8 objects have to be present (parent, source and destination, connection object, connector symbol and 3 associations). Looking at the rule in Fig. 26(b), it is hard to visualize the resulting structure. Unfortunately, in GReAT it is very difficult to include model fragments *at the modeling (i.e. not the metamodel) level*, such as the structure shown in Fig. 26(a).

Finally, in a platform modeling scenario a the high-level structure of the transformation is fixed:

1. Create the Kernel "skeleton", forming the backbone of the analysis model
2. Add and customize (instantiate) component skeletons
3. Integrate the resulting model structures

This structure can be made implicit in the platform modeling language, thus making both the language definition, both the models specified in the language simpler. Of course, it is still possible to map (compile) this higher-level language onto a general-purpose graph transformation language (such as GReAT). Thus, the resulting "low-level" transformation can be integrated seamlessly with the PSM→PIM transformation within the DSML→Platform→Analysis chain.

The following section introduces PML, the Platform Modeling Language, developed to address the above issues.

The Platform Modeling Language (PML)

PML is a simple, declarative formalism to specify the *Platform→Analysis model* mapping outlined above. The language supports the building of analysis (target) models by inserting, customizing and composing target model fragments (called "skeletons") into the model being built. These skeletons model platform-level components. PML uses graph patterns to identify, locate and insert these skeletons in the source and target models. Furthermore, PML includes a simple declarative graph transformation ("mappings") language for the composition and customization of these template structures.

The pattern language tries to keep (and leverage on) the most useful features of GReAT, such as constructing the pattern language over the metamodels and the UML-inspired visual language. At the same time, PML tries to be as "clean" and simple as possible. In patterns, LHS and RHS are syntactically separated, the execution semantics is much simpler (no control flow structures), as well as the (implicit) rule sequencing. Complex (LHS) patterns can be broken up into simpler parts, and the passing of the pattern context among these parts is simpler and easier to use than that of GReAT.

Unlike in GReAT (e.g. as shown in Fig. 26(b)), the output model fragments corresponding to platform components can be given at the model level, as shown in Fig. 26(a). Thus, these partial models can be created and edited using the visual modeling environment created for the analysis language. This makes the platform modeling process much more user-friendly and less error-prone.

The output is still generated as a GME model, conforming to the analysis language metamodel. Using this approach, the resulting analysis model can generated in a tool-independent format, and simple "export" tools could be provided within the modeling framework to generate tool-specific concrete syntaxes. For example, the resulting "generic" timed automata could be exported into text files conforming to the input format of different tools (UPPAAL [34], HyTech [28], IF [16] etc.).

A PML model consists of 4 basic parts, corresponding to the three tasks outlined above, plus one to establish the pattern language:

1. The UML metamodels for the source (platform) and target (analysis) models. PML provides an import tool to include GME metamodels into platform models. Additionally, PML models may contain optional crosslinks definitions, to be used later during the mapping. Unlike in GReAT, it is not mandatory to define crosslinks in advance: they can be introduced "on the fly" in the mapping rules.

2. The construction of the output model starts with the instantiation of the Kernel (given as a partial analysis model).

3. Next, *components* are identified in the source (platform) model via pattern matching. A target model fragment is associated with each component in the output model, and customized (instantiated) based on the source component. The destination within the target model for the copy operation is also designated by a pattern. Crosslinks are automatically generated to link the source component with the corresponding target model structure copied.

4. Finally, a simple GT language (*mappings*) is provided to further customize and extend the target model.

Appendix A details the PML implementation (metamodel and framework) created in GME.

The following sections introduce the above 4 basic parts of PML. Appendix B provides detailed examples for the concepts below from the DFK → UPPAAL PML model.

Metamodels, Crosslinks and the Kernel skeleton

Metamodels

Similar to GReAT, PML also assumes that the target model (the platform-level model) is specified as a UML object diagram, conforming to a *platform metamodel* M_P. Similarly,

the resulting analysis model A_i is an object diagram as well, conforming to the *analysis metamodel* M_A. The visual pattern language is established over these metamodels.

Thus, the PML model has to contain the UML metamodels of the source and target metamodels, just like with a GReAT transformation. The PML environment provides an import tool (`ImportMeta`) which performs this task for GME metamodels.

Crosslinks

Crosslinks are used (and defined) the same way as in GReAT, in a separate folder. Forward declaration of crosslinks is necessary only if either endpoint is an abstract class. Crosslinks between non-abstract classes can be used "on the fly" in mappings.

Kernel skeleton

Each platform model contains a filename ($KernelSkeleton$) designating a target model file, containing the skeleton of the Kernel model. Building of the target model starts by copying this model into the (empty) analysis model, and all subsequent operations commence from this model.

Component Skeletons

The concept of a *component* is central to PML. Components are a related group of elements in the source model, identifiable by a pattern. The pattern designates a single element to represent the component. This element is typically a container in the modeling language, such as the $Node$ element in DFK. Components are mapped to a corresponding structure in the target model. For example, in the UPPAAL model a TA ($Template$) models each $Node$, along with a few associated global variables and declarations (e.g. the PID of the component process).

This concept is modeled through *ComponentModel* structures in PML. ComponentModels contain:

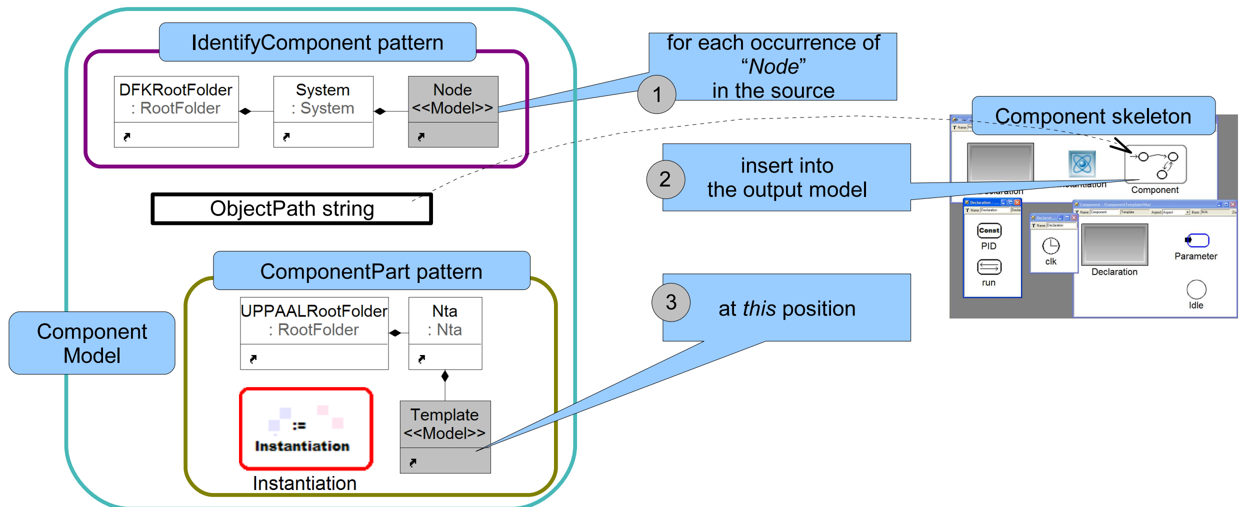


Figure 27: Visualization of the ComponentModel concept

- The name of an analysis (e.g. UPPAAL) model file, containing the skeleton
- A pattern to locate the component within the source model (*IdentifyComponent*)
- Definitions of one or more target model fragments, to which the component is mapped in the analysis model.

The idea of a ComponentModel is visualized in Figure 27.

Identifying Components

Components in the source (platform) model are identified by an *IdentifyComponent* pattern. This pattern defines "what is a component" in terms of source metamodel elements. The pattern also designates a "representative" element for the component. This is done via a specific pattern element, *LocatorElement*. The transformation creates associations between this representative element and the corresponding analysis fragments. Thus, the source component for each analysis fragment can be traced back at a later stage.

Component Parts

A single component might have corresponding elements / structures at multiple locations in the analysis model (e.g. the TA template and the PID global constant). In order to facilitate this, ComponentModels may contain one or more *ComponentParts*. ComponentParts define the following:

- The location of the fragment within the component skeleton model (*ObjectPath*)
- The destination of the fragment in the analysis model being built.

ObjectPath is given as a string attribute (see Appendix B for details). The destination within the target model is defined by a pattern, contained by the ComponentPart model. The pattern is formulated over target model elements, and designates a single element (*Locator*). During the mapping process, this element is created within the target model, and the skeleton is copied into it. The ComponentPart pattern might also contain a procedural code fragment (*Instantiation* box). This code sets the attributes of the inserted fragments using the GME API. The code can access both the source and target model elements.

Using this approach, the "skeletons" forming the backbone of the analysis model can be comfortably edited and reviewed using the modeling framework.

Mappings

In addition to the skeleton instantiation language, PML also features a simple graph transformation language (*Mappings*). The main purpose of this language is to facilitate fine-grained customization of the analysis model.. The mappings blocks are evaluated *after* the component skeleton instantiations had taken place.

It is important to emphasize that although the mappings language is auxialary to skeleton instantiation in PML, it is still a capable graph transformation formalism. All the mappings performed through component skeleton instantiations could be performed using only the

mappings language, at the expense of sufficiently complex mappings rules, as it will be demonstrated by the following examples.

The language uses rewriting rules based on graph patterns. The patterns are formulated similarly to those in GReAT. The main design intention was to come up with a simpler (easier to learn and understand) language. For somebody with a background in GReAT, the differences can be summarized as follows. In PML,

- LHS and RHS of graph patterns are syntactically separated into *filter conditions* and *actions*.
- Rule context propagation is simpler, subsequent rules can simply reference to elements matched earlier (*MatchReference*).
- There are no (explicit) control flow structures.
- The rule execution semantics is much simpler.
- There is no *delete* operation.

In the following sections, the basic concepts of PML Mappings will be introduced through a simple example. The example shows mapping rules from DFK (the metamodel in Fig. 23 is used) to UPPAAL (Fig. 17). Appendix B contains additional, more sophisticated examples.

A PML mapping identifies M_P patterns through *filter conditions* (LHS), and maps them onto M_A structures as specified by *action patterns* (RHS). Both the actions and conditions are expressed as graph patterns over M_P and M_A elements. Filter patterns can be annotated by C++ code snippets as boolean-valued *guard* functions over pattern elements' attribute values.

A filter condition is *satisfied* if the pattern matches a sub-graph of the input model and the optional guard expression evaluates true. Filter conditions are assumed to be side-effect free.

With the associated filter(s) satisfied, an action could get *executed*, creating the elements specified in the *action pattern* and setting attribute values as specified in optional *SetAttribute* C++ code fragments. (The precise execution semantics is discussed later in this chapter.)

Filters and actions contain two kinds of pattern elements:

a) *MetaClasses* are metamodel elements to be matched (in filters) or created (in actions).

b) *MatchReferences* refer to elements matched by previous filters (more about filter hierarchy in the next section).

Similar to MetaClasses, associations between elements in filters are to be matched, and to be created in actions. The dual role of MetaClasses and associations reflects the LHS nature of filters and RHS nature of actions.

In patterns, elements may refer to instances from both M_P and M_A. For all patterns, matching always starts from the two top-level objects (RootFolders from both models). In other words, in a pattern any element (except RootFolders) has to be associated with an other element.

A *mapping block* has one filter condition, and zero or more actions.

The next section explains context (bound object) passing between patterns and pattern hierachy in PML.

PML block hierarchy

Mapping blocks can be organized into hierarchy to simplify filter patterns. Figure 28 illustrates this (filters are named f_i, actions a_j and blocks b_k):

The topmost filter f_1 (in block b_1) matches the DFK *System*. Sub-block b_2's filter f_2 refers to the *System* matched using a *MatchReference* object, and matches all *Nodes* within the system, along with their associated UPPAAL TA Template. ($SrcComponent \rightarrow DstComponent$ associations are made by component skeleton instantiations).

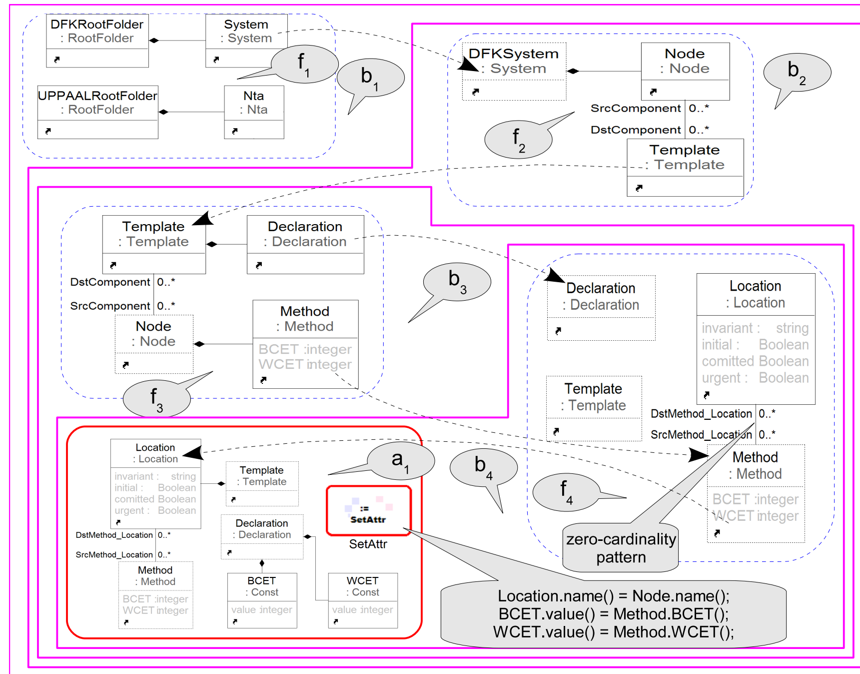


Figure 28: PML Block hierarchy to map Nodes and Methods

Inside b_2, sub-block b_3's filter (f_3) matches the TA template's *Declarations* block and all *Methods* within. f_4 matches all such Methods with no TA Location associated, and action a_1 creates the Location and maps the Method's BCET and WCET values onto constant declarations within the TA template.

Filter conditions in sub-blocks may refer (indicated by dashed frame) to elements matched by the immediate parent block's filter (*MatchReferences*). This enables breaking up filter patterns into simpler ones by eliminating the necessity to always derive each elements from RootFolder(s). (Some – not all – MatchReferences are visualized by dashed arrows in Fig. 28).

Note that the hierarchy does not imply scheduling for *action execution*, it is only to aid logical organization. Action execution order is discussed in the following section.

Global Filter Condition

For each action a_i, a *Global Filter Condition*(GFC_i) can be constructed by composing associated filter conditions along the block hierarchy:

Definition Let f_i be the filter of the block containing a_i. Let f_{i-1} be the filter of the immediate parent block (i.e. one level up in the hierarchy). $Comp(f_i, f_{i-1})$ is a filter obtained by joining f_i and f_{i-1} together as follows:

1. the *glue graph*, $Glue(f_i, f_{i-1})$ is the set of elements of f_{i-1} referenced in f_i (dotted frame in the diagrams) along with the associations internal to the set. Start the composition with the glue graph, $c_0 = Glue(f_i, f_{i-1})$.

2. construct $c_1 = Attach(c_0, f_i \setminus Glue(f_i, f_{i-1}))$: add the rest of f_i and attach the associations between f_i and the glue graph.

3. $c_2 = Attach(c_1, f_{i-1} \setminus Glue(f_i, f_{i-1}))$: add the rest of f_{i-1} and attach the dangling associations between c_2 and the rest of f_{i-1}.

4. $Comp(f_i, f_{i-1}) = c_2$.

Then, let us define the composition of filters along the block hierarchy as:

$$Comp(f_i, f_{i-1}, \ldots, f_0) = Comp(Comp(Comp(f_i, f_{i-1}), f_{i-2}), \ldots), f_0)$$

and the global filter condition for action a_i as

$$GFC_i = Comp(f_i, f_{i-1}, \ldots, f_0)$$

where f_i is the filter associated with action a_i.

For example, $GFC_1 = Comp(f_4, f_3, f_2, f_1) = Comp(Comp(Comp(f_4, f_3), f_2), f_1)$ in Figure 28 for action a_1.

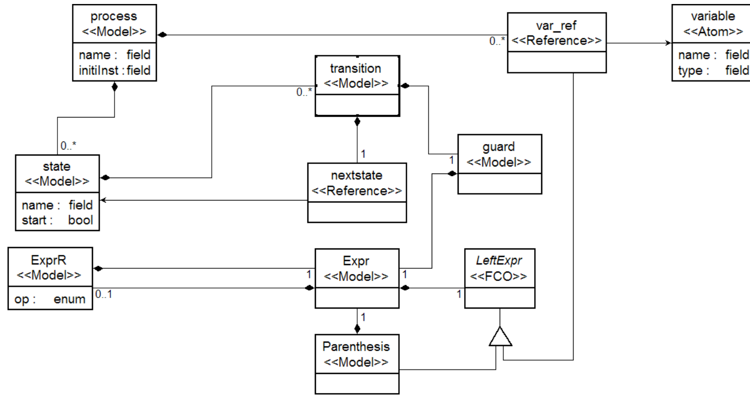


Figure 29: The simplified IF metamodel

Execution semantics for actions

The execution semantics for actions is shown on Algorithm 2:

Algorithm 2 Action execution semantics

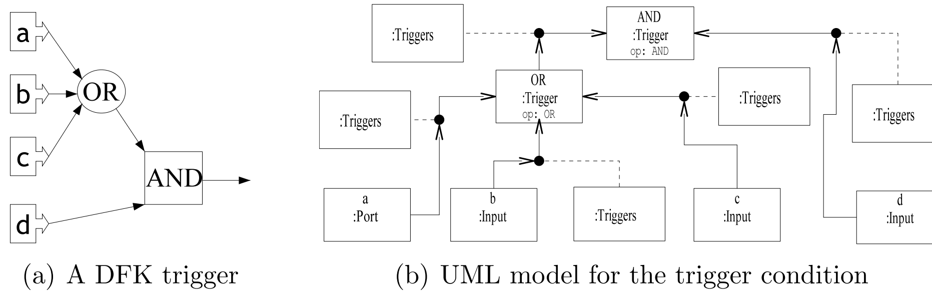
---

while $\exists i$ such that (GFC_i) is *true* **do**
  Execute $action_i$
end while

If there are multiple actions eligible for execution, one is selected non-deterministically. There is no explicit ordering. Implicit ordering can be established by referring to elements to be created by "previous" actions.

A detailed example: mapping triggers for IF

In this section, a detailed, non-trivial example will be discussed to illustrate the PML mappings language. The example shows mapping rules from DFK (the metamodel in Fig. 23 is used) to a simplified model of the IF analysis language. IF [16], just like UPPAAL has its own Timed Automata implementation. The simplified IF metamodel is shown in Figure 29 (many details omitted for simplicity). IF systems consist of automata (*processes*) with *states*, *transitions*, and *variables*. For *transition guards*, a simple expression language featuring binary AND, OR operators and variable references (semantics: $var \neq 0$) is modeled. The



(a) A DFK trigger (b) UML model for the trigger condition

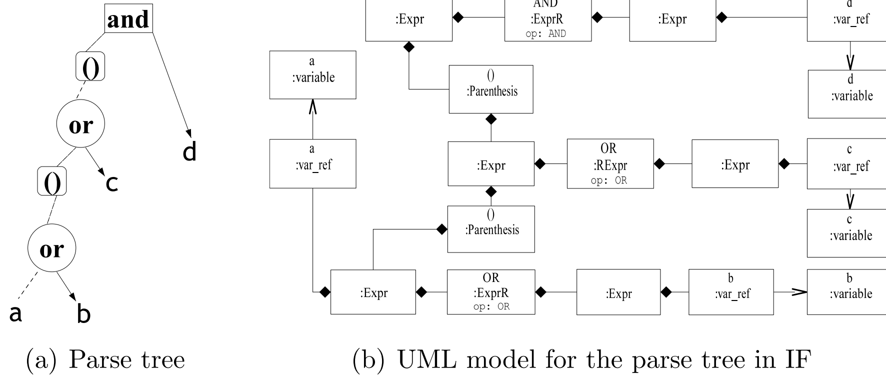
Figure 30: Trigger condition $(a \vee b \vee c) \wedge d$ in DFK

expression language models a simple parsing tree: expressions always have a "left" side ($Expr$), which might have a "right" side ($ExprR$), or contain a sub-expression in parenthesis. The trigger mapping example discussed later explains the expression model in detail. In the IF metamodel, operations on dataflows, variable updates and complex expressions are omitted for simplicity.

Mapping DFK *trigger conditions* onto IF *guard expressions* is a good example to illustrate the capabilities of the PML mappings language. It is especially educative for readers with background in GReAT, as it emphasizes the differences between the GReAT and PML approaches. (In GReAT, a recursive rule block would have been used.) In the following section, this transformation, which builds an expression parse tree based a DFK trigger structure will be discussed in detail.

Figure 30 shows a DFK trigger condition ($\{a, b, c, d\}$ are Ports) and its UML model, using the metamodel from Figure 23. Such a condition may trigger a Method or be a part of a larger trigger condition. Figure 31 shows the corresponding expression parse tree and the UML diagram, according to the IF metamodel in Figure 29, where $\{a, b, c, d\}$ are variables. Note the difference between the expressions: since in IF, the logic operators are *binary*, the parse tree is 3 levels "deep" (not counting the `()` operators), whereas in DFK it has only 2 levels.

The PML mapping is shown in Figures 33-35. It implements Algorithm 3, and can be summarized as follows:



(a) Parse tree (b) UML model for the parse tree in IF

Figure 31: Trigger expression $((a \vee b) \vee c) \wedge d$ in IF

Algorithm 3 Mapping trigger conditions

1: find a Triggers→Expr pair (T_i, E_i)
2: **if** $Src(T_i) = P_i$ is a *Port* **then**
3: create the expression into E_i meaning "data available on P_i" (Fig. 33).
4: **else if** $Src(T_i) = T_s$ (it is a Trigger itself) **then**
5: **if** the "left" branch is empty for E_i **then**
6: create a (yet empty) $E_{s[ht!]}$ into the left branch, associate (T_s, E_s) (Fig. 34).
7: **else if** the "left" branch is not empty for E_i **then**
8: create $RExpr_{op}(E_s)$ with *op* from $Src(E_i)$ (AND or OR) (Fig. 35).
9: **end if**
10: **end if**

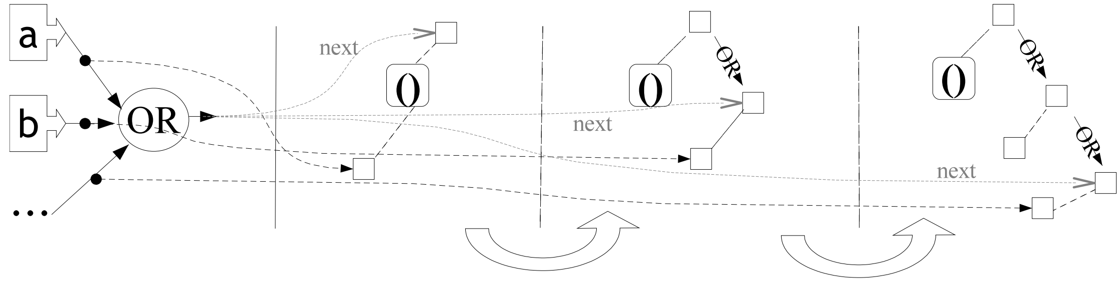


Figure 32: Progress of the trigger mapping process

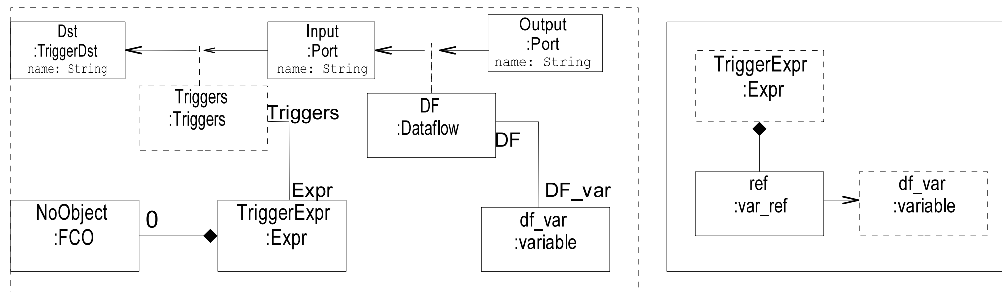


Figure 33: Block mapping a Port trigger source (steps 2-3))

First, find the top-level *Triggers* connection (i.e. the one connected to a Method) and create and assign a *Expr* to it through a Triggers→Expr *crosslink* (this preliminary step is not shown). Then, for each Triggers connection, create a "left" branch in the parse tree and associate it with its source Trigger's *first* argument. For each subsequent argument (incoming Triggers), create a "right" branch, and add a level to the right side of the parse tree. The progress of the algorithm is illustrated in Figure 32.

Figure 33 shows the PML block implementing steps 2-3: the filter pattern describes the input port with a Dataflow connection and the associated IF variable, and the corresponding *empty* Expr. The action creates a *var_ref*.

Figure 34 shows the block corresponding to steps 5-6: $Src(T_i)$ is a Trigger (`Src`), T_i has an empty Expr associated, and `SrcTrigger` has not been visited yet. In the action, a parenthesized Expr is created in the left branch of `TriggerExpr` and associated with T_s. Also, it is designated as the new "rightmost" node (`Next`) for T_i.

Figure 35 illustrates steps 7-8.

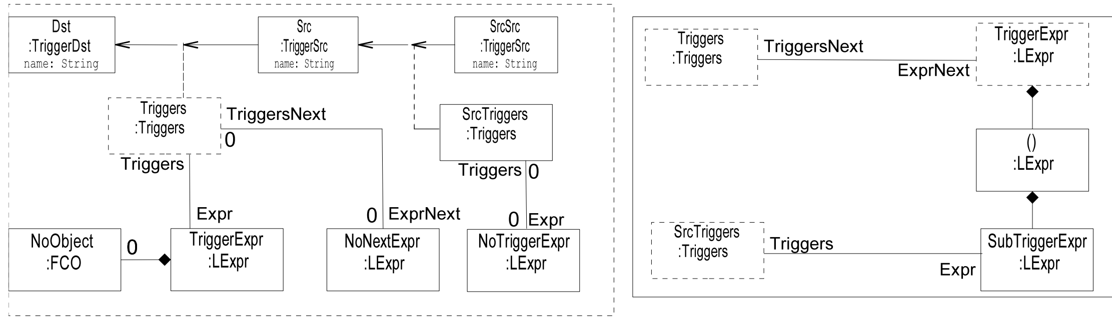


Figure 34: Block mapping an upstream trigger onto an Expr (steps 5-6)

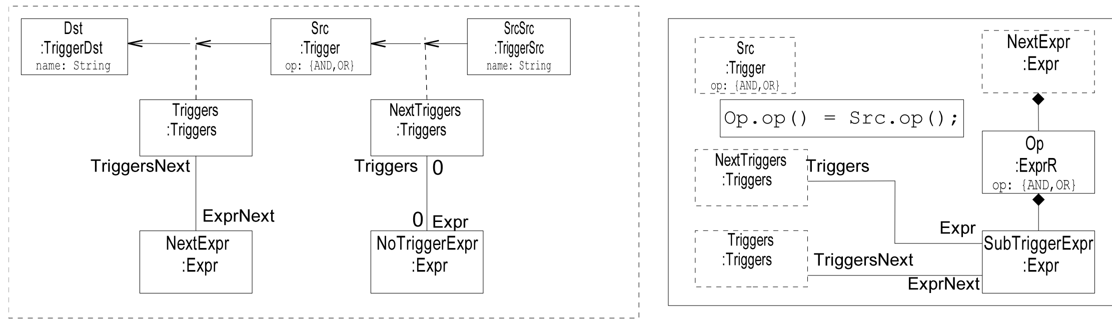


Figure 35: Mapping a subsequent trigger argument onto an ExprR (steps 7-8)

Comparing the DFK → UPPAAL mapping in PML and GReAT

The following table gives a rough complexity comparison of the DFK → UPPAAL mapping in PML and in GReAT. The columns give the counts of GME modeling elements in the models containing the mapping specifications.

In this example, the difference is small because the example itself is small. In general, the more complex the component skeletons are, the more compact the PML model is (versus a pure GReAT solution).

As for mapping blocks, the same mapping problem can usually be solved easier in GReAT. The reason for this is the availability of sophisticated control flow constructs. PML mappings as a language is much simpler and less efficient in terms of compactness.

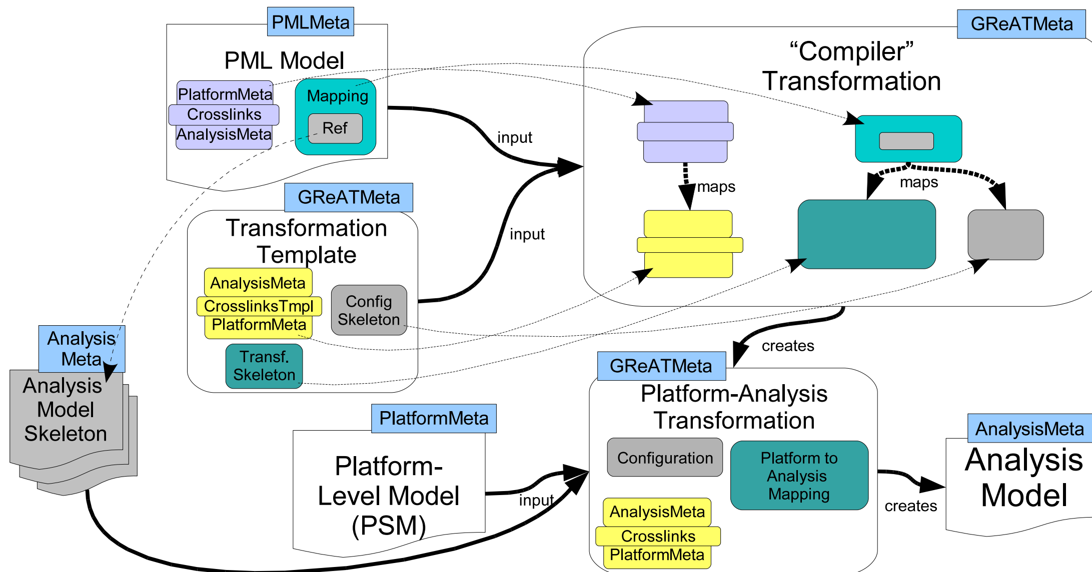


Figure 36: High-level overview of the PML → GReAT transformation

Transformation	Atoms	Models	Connections	Total
Hand-optimized GReAT	162	43	936	1142
PML	77	59	642	778
PML compiled into GReAT	567	86	2760	3413

Implementing PML over GReAT

PML shares several key concepts with GReAT , such as pattern matching on metamodel-conformant UML model instances. Hence, implementing PML over GReAT is a straightforward idea. The implementation is done by mapping (translating) a PML specification into an equivalent GReAT transformation — by a "compiler" GReAT transformation.

Figure 36 provides a high-level overview: Each component in the figure is a model, and the corresponding metamodel's name is written above the model in a rectangle. The "compiler" takes a PML model and a GReAT Platform → Analysis transformation template, and extends it with rules generated based on the PML model.

The PML model contains the UML metamodels (using an encoding suitable for PML) of the source and target models (platform, analysis), and optional crosslinks. Most importantly,

it also contains the Platform $\rightarrow$ Analysis mapping definition, with references to external analysis model fragments (skeletons), which are used in Kernel and Component mappings.

The "compiler" transformation (implemented also in GReAT) takes the PML model $\mathcal{M}_i$ and a GReAT template. This template contains the appropriate metamodels (imported into GReAT format), and skeletal configuration and transformation specifications (rule-blocks). The "compiler" works by extending these configuration and rule-block skeletons to implement the mapping given in the PML model. This creates a GReAT transformation, $\mathcal{A}_{(P_i,j)}$, which is the Platform $\rightarrow$ Analysis transformation at the bottom of the Figure 36. This transformation inputs the platform-level model $\langle P_i SM \rangle_j$ (and the analysis model skeletons, as specified in $\mathcal{M}_i$), and creates $A_{(P_i,j)}$, the analysis model.

In this approach, the GReAT transformation template has to be provided by the user for each (platform,analysis) pairs. The template's structure is fixed, and contains about 20 elements plus the metamodels. Building the template (or modifying an existing one) takes a few minutes by hand. Since the structure is fixed, a GReAT transformation could also be developed to modify an existing template for a new (platform,analysis) pair.

The rest of this section will discuss these (the "compiler" and its output, $\mathcal{A}_{(P_i,j)}$) transformations.

The PML $\rightarrow$ GReAT "compiler" transformation has the following major phases:

1. Mapping metamodels and crosslinks. Ultimately, the transformation maps PML patterns onto GReAT patterns. Both PML and GReAT models import the appropriate metamodels in order to establish their pattern languages. The GReAT transformation template already contains the metamodels, and the "compiler" associates the metamodel elements in the PML models with these. These associations will be used later when the patterns are mapped onto GReAT ones.

2. Extending the template transformation's configuration so that the appropriate component skeletons are read and input by the main transformation block. The attributes of the output model are also set so that it reads the kernel skeleton and copies it into the

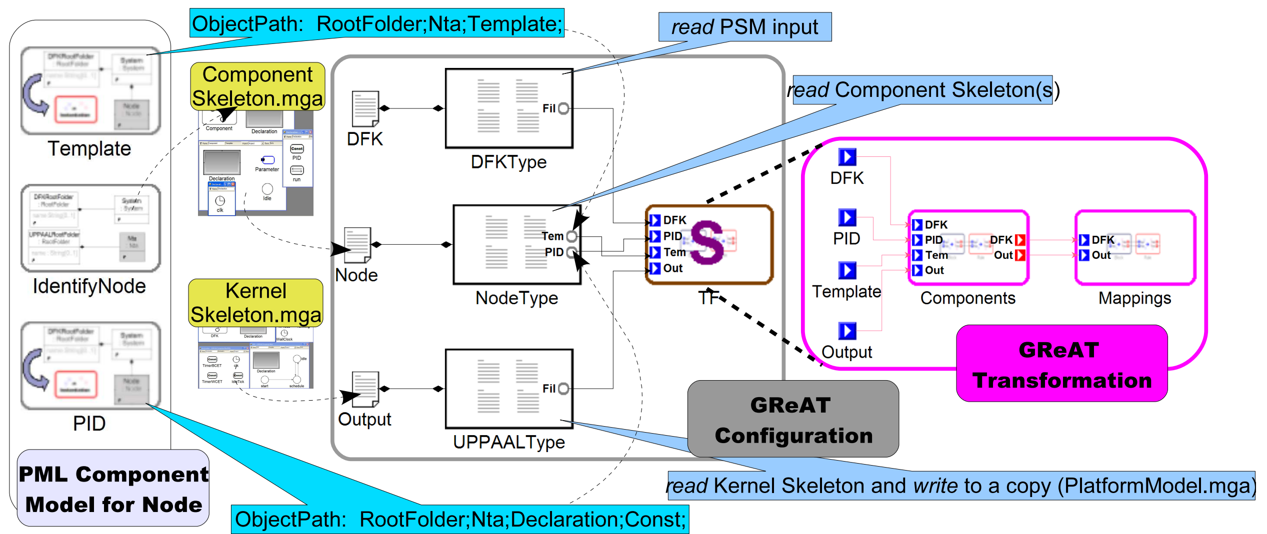


Figure 37: Configuration of a generated GReAT transformation

output model first. This way, the basic structure of the analysis model.is established at the earliest stage.

3. The template rule block `Components` (on the right side of Fig. 36, within block "GReAT Transformation") is extended to implement the mapping of components captured in the PML model.

4. The template rule block `Mappings` (block next to is `Components` extended to implement the mappings defined in the PML model.

Extending the transformation configuration

The template GReAT transformation contains a default configuration. During PML → GReAT mapping, this has to be concretized and extended according to the actual PML model.

This involves two steps:

1. Add the kernel skeleton model to the output file specification (so that it gets copied into the target model)

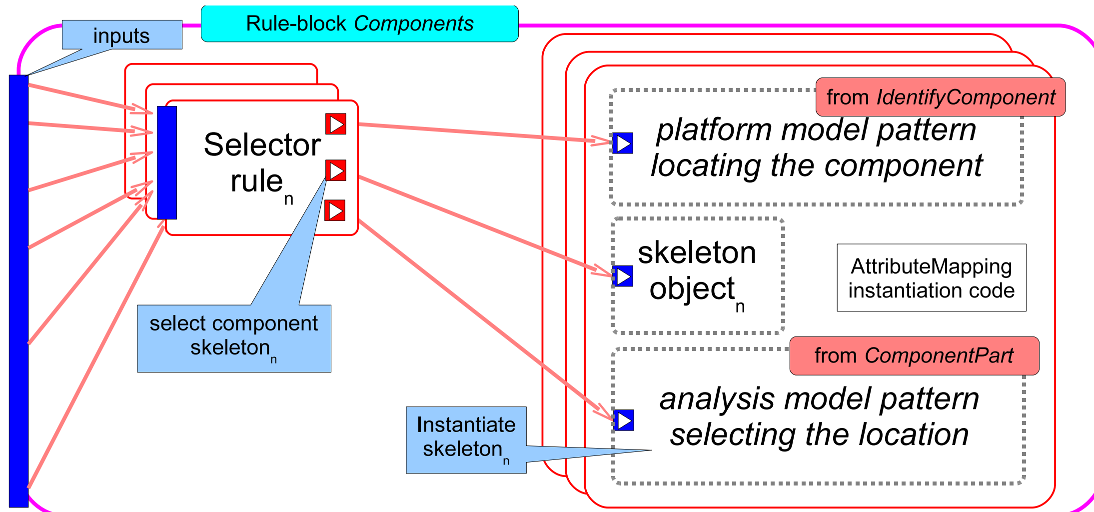


Figure 38: Schematics of the rules implementing component skeleton instantiation

2. Input the component skeleton models into the transformation.

The first step is straightforward, it involves rewriting the corresponding filename attribute. The second step is more involved, and consists of the following:

a) For each ComponentMode, create a *(File,FileType)* pair, pointing to the skeleton model file

b) For each ComponentPart, create a *FileObject* into the corresponding FileType object.

Figure 37 shows the configuration for a platform model with one ComponentModel and two ComponentParts. The GReAT configuration can be seen in the center. The figure also shows how the models defined by the configuration are interfaced with the transformation (on the right side).

Generating rules for component skeleton instantiation

After extending the configuration, the next phase creates rules to instantate component skeletons. This happens by extending rule-block `Components` (shown on the right in Fig. 37).

The configuration model passes all objects representing analysis model fragments (*ComponentParts*) into this rule-block. For each of them, a pair of rules is created. The first rule selects the template object and both (source and target) RootFolders.

The second rule performs the instantiation. It is the composition of:

a) The *ComponentIdentifier* pattern, matching the component in the platform-level model.

b) The template object from the componet skeleton model

c) The *ComponentPart* locator pattern, matching its location within the analysis model

d) AttributeMapping code performing the instantiation.

The instantiation takes place if both – a) and c) – patterns matched. The AttributeMapping code performs a "deep copy" of the template object into the location identified by pattern c). (Schematics shown in Fig. 38).

The "deep copy" operation is currently not supported by GReAT. Fortunately, the operation is available in the underlying library (UDM [14]) used to implement GReAT. Thus, a C++ code fragment using the `CopyUdmHierarchy` call of the UDM API is generated into the AttributeMapping code performing the deep copy. Additionally, this code box also contains the *Instantiation* code supplied with the ComponentPart in the PML model.

Appendix C contains two detailed examples from the DFK → UPPAAL mapping. The examples show and explain the generated rules corresponding to component skeleton definitions in the PML model.

Implementing PML Mappings in GReAT

The final task of the compiler is the translation of the *Mappings* block into GReAT. This involves two steps:

1. Mapping of PML filter conditions and actions into GReAT rules

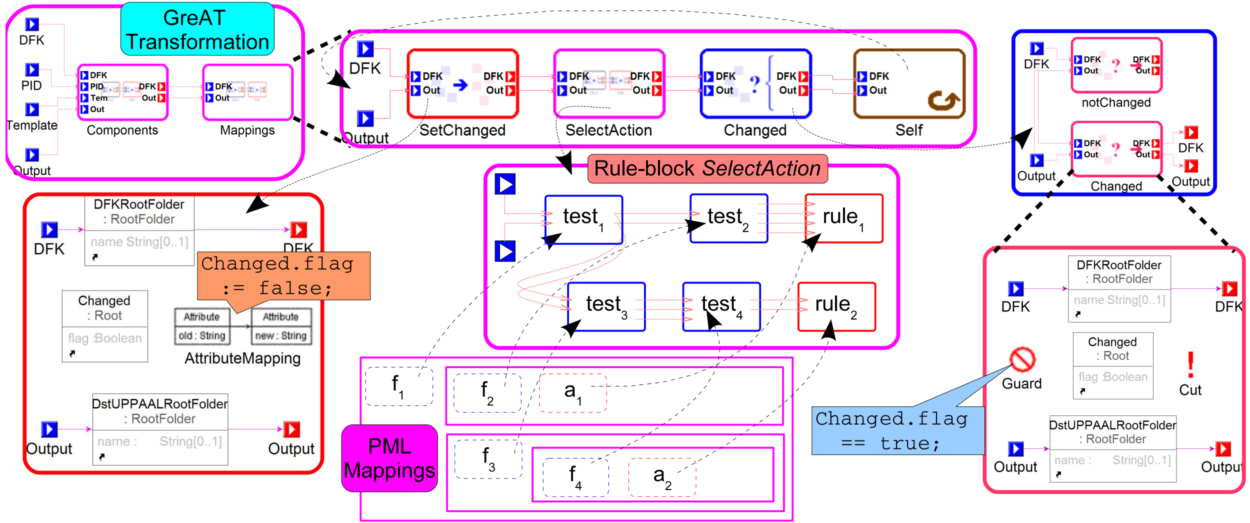


Figure 39: Schematics of PML Mappings implementation

2. Constructing a control flow structure to implement PML Mappings semantics (as shown in Algorithm 2): select $action_i$ whose filter conditions are satisfied, execute it and repeat this as long as possible.

The schematics of the GReAT rule-block implementing this is shown in Fig. 39. The iterative PML action execution is realized using a recursive rule-block (`SelectAction`, in the center).

Mapping filter and action patterns is straightforward. Each filter is mapped into a GReAT Test/Case (pattern matching with no side effect). Context propagation paths for MatchReferences betweeen GReAT patterns are generated by the compiler. Each action is represented by a rule. In rules, each pattern object is either bound by the previous filter pattern, or has its *Action* attribute set to *CreateNew*.

The GReAT transformation has a *global object* (`Changed`, defined in the Crosslinks folder), with a boolean flag. At the beginning, this flag is set to *false* by rule `SetChanged`.

`SelectAction` evaluates the *Global Filter Condition* (GFC) for each action by evaluating corresponding GReAT *Test/Case* chains. In each chain, the last element is a GReAT rule, corresponding to an action. If all the preceeding test cases (corresponding to filter conditions)

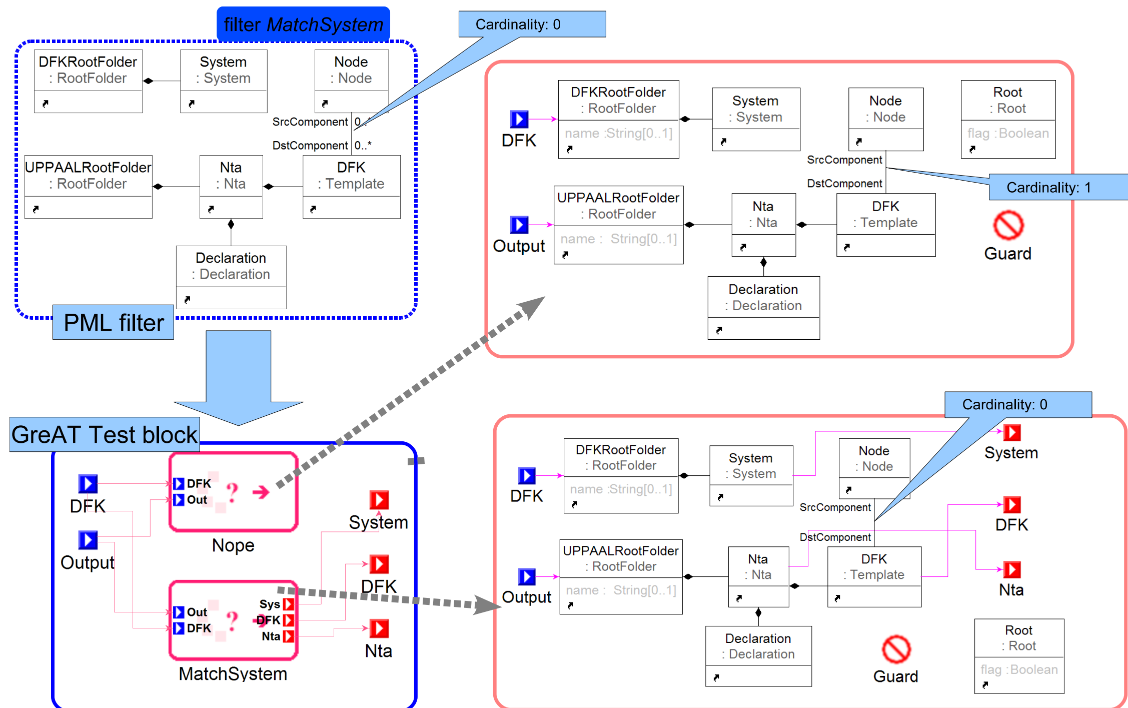


Figure 40: Mapping a zero-cardinality pattern

matched, the rule executes, carrying out the action. Whenever an action executes, `Changed` flag is set to *true*.

Each rule (both filter and action rules) in `SelectAction` has a guard condition returning `Changed.flag`. Thus, if one action has already executed, no other pattern (filter or action) will match, and a `SelectAction` cycle finishes.

Next, as shown in Fig. 39 in test block `Changed`, if `Changed.flag` was true (if any action rules had executed), rule-block `SelectAction` is invoked again using a GReAT self-reference (`Self`). If `Changed.flag` is false, no rule could execute in this cycle, so the transformation terminates.

Appendix C provides more detailed examples for actual PML → GReAT mappings.

Zero-cardinality patterns

PML also supports *zero-cardinality patterns*: patterns which express the *lack of* something. Patterns with zero cardinality match if the pattern object *is not* present. Figure 40

shows the mapping of a pattern containing an association with zero-cardinality. The filter (on the top left) matches UPPAAL Templates in the Nta with *no* associated DFK Node through $(SrcComponent, DstComponent)$.

The filter is mapped onto the Test block shown below. It contains two *Case* patterns, one with the zero-cardinality associations' (GReAT) cardinality set to 1, and one with the original zero cardinality. The GReAT pattern matcher evaluates this block as follows: If the model does contain the association in question, the top pattern (`Nope`) evaluates true, and the bottom one is ignored. Since the top pattern has no output binding(s), the Test block itself generates no matches for subsequent blocks. Thus, the evaluation of the GFC (represented by this Test block chain) fails. On the other hand, if the given association is not present, the above pattern fails to match (as it has the association with nonzero cardinality). Then, the pattern matcher evaluates the bottom Case (`MatchSystem`) which succeeds. The output bindings pass the matches down to the next block, and the evaluation of the GFC chain continues.

The compiler transformation

The schematics of the PML→GReAT compiler transformation was shown on the right side of Fig. 36. The transformation takes a PML model and a skeleton GReAT transformation, and proceeds by extending the skeleton according to the PML model.

The skeleton transformation contains the following:

1. The source and target metamodels imported using the appropriate GReAT tools.

2. A Configuration model, with references to the input (platform-level) and output (analysis) model files.

3. An empty `Components` block and a `Mappings` rule-block with an empty `SelectAction` loop. (Review Figure 37).

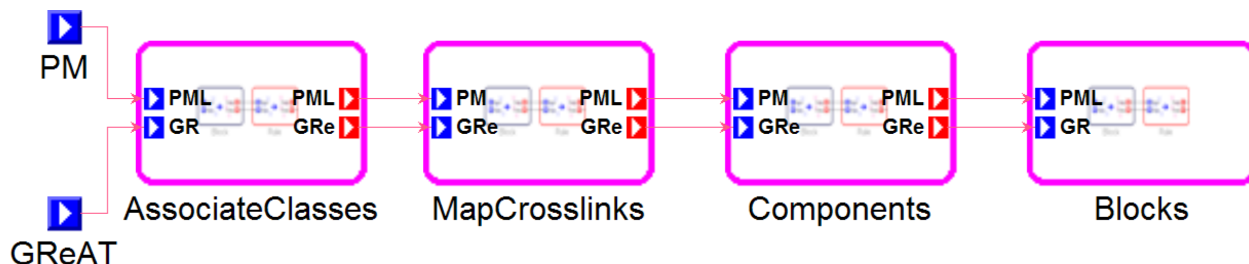


Figure 41: Major phases of the compiler transformation

The compiler itself is specified in as a GReAT transformation, and its high-level schema can be seen in Fig. 41. The transformation has four major phases:

1. **Associate metamodel elements** The source PML and the target GReAT models both contain the (platform, analysis) metamodel pairs. In order to construct GReAT patterns corresponding to PML patterns later on, the compiler needs to know the corresponding GReAT metamodel element for each PML object. In both languages, pattern elements reference to metamodel elements. Thus, they can be associated through their respective meta-elements.

2. **Map crosslinks** Using the PML→GReAT element-wise mapping established in the first step, crosslink definitions is the PML model are mapped onto GReAT crosslink definitions.

3. **Instantiate component skeletons** The next step is processing ComponentModel elements. In this step, the resulting transformation's Configuration model is extended with input models. These models contain component skeleton definitions according to the PML model. Additionally, the models are propagated into the transformation, and rules instantiating template skeletons are generated (as shown in Figures 37).

4. **Generate Mappings rules** Finally, GReAT rules implementing PML mappings blocks are generated, as illustrated in Figures 39.

The following sections discuss each step in detail.

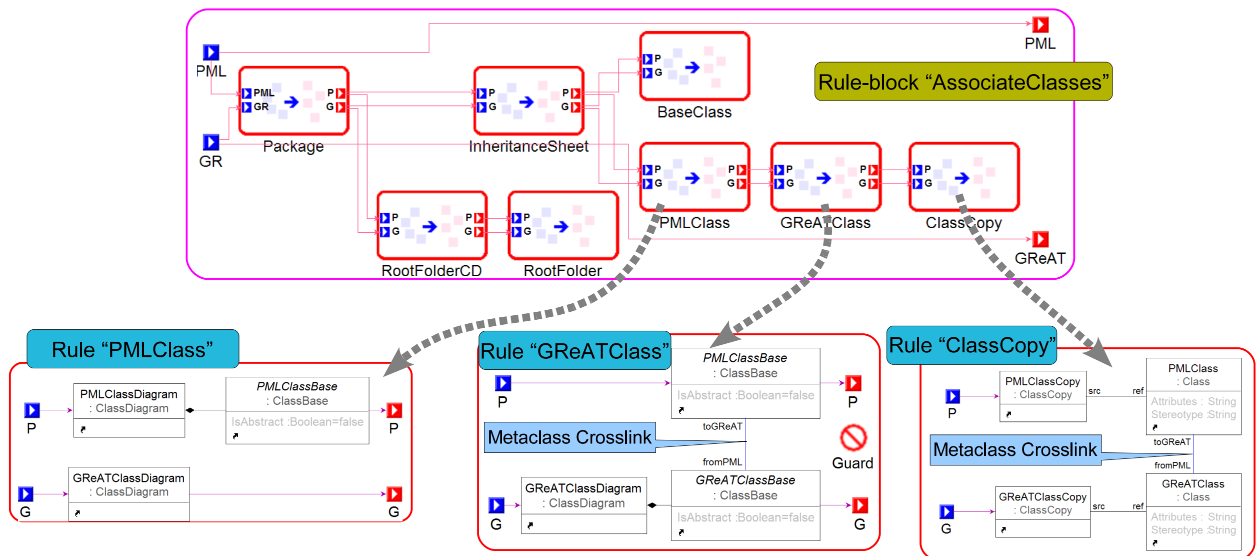


Figure 42: AssociateClasses rule-block from the compiler

Associating metamodel elements

In this step, metamodel elements in the PML model are associated with their counterparts in the GReAT transformation. The compiler creates *(fromPML,toGReAT)* crosslinks between the corresponding elements. These crosslinks will be used later in mapping patterns from PML to GReAT.

Figure 42 shows the overview and representative rules of this phase. The initial rules descend into the UML *Package* folders containing the metamodels in both the PML and GReAT models. There are three kinds of classes to associate:

- RootFolders (which are special classes both languages)
- PML *BaseClasses* to GReAT *MgaObjects* (all other classes derive from these special abstract classes in the metamodels).
- "regular" class definitions in metamodels.

Rules performing the last task are shown in detail in Fig. 42. The rules traverse the PML and GReAT metamodels in parallel. All PML metamodel elements are matched, and the

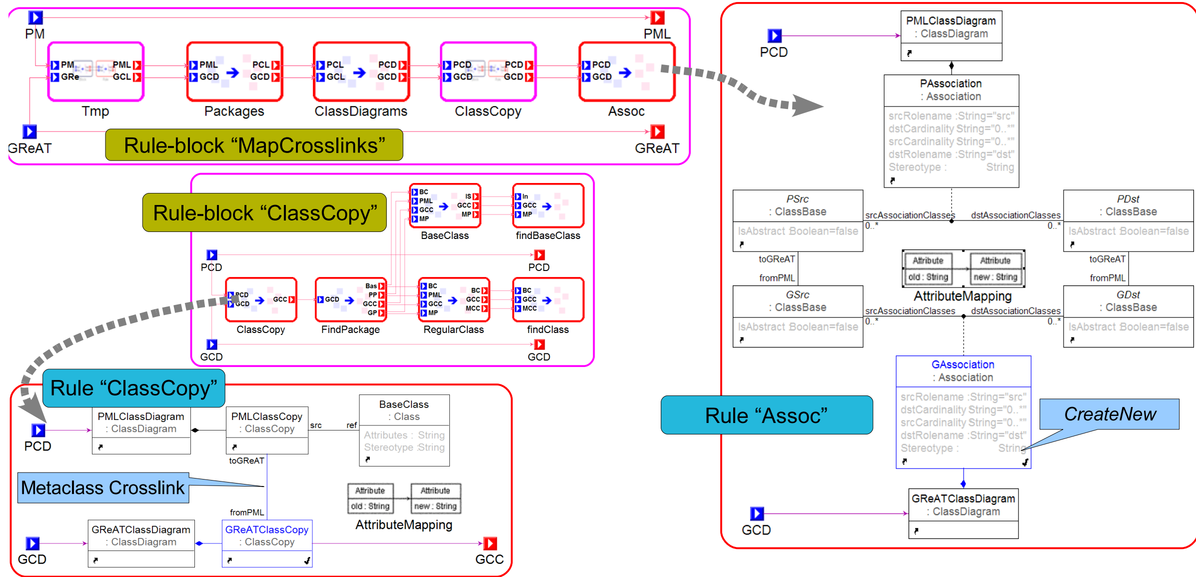


Figure 43: MapCrosslinks rule-block in the compiler

corresponding GReAT object located in the appropriate folder (ClassDiagram). Once the object pair is identified, a *(fromPML,toGReAT)* association is created.

Mapping crosslinks

The next step maps crosslink definitions in the PML model onto GReAT crosslink definitions. For the mapping, the crosslinks created in the previous step are used.

First, the folder containing crosslink definitions in the PML model is located. Then, each crosslink definition is matched within, along with the source and destination meta-objects. Following the *(fromPML,toGReAT)* crosslinks created previously, the corresponding GReAT meta-objects are determined. Finally, a crosslink definition involving these objects and an association using the rolenames from the PML model is created in the GReAT model.

The rule-blocks involved are illustrated in Figure 43. The most important rule is expanded on the right.

Generating component instantiation rules

This phase processes the ComponentModels defined in the PML model. The result is an updated transformation configuration, including the skeleton models. Additionally, rule

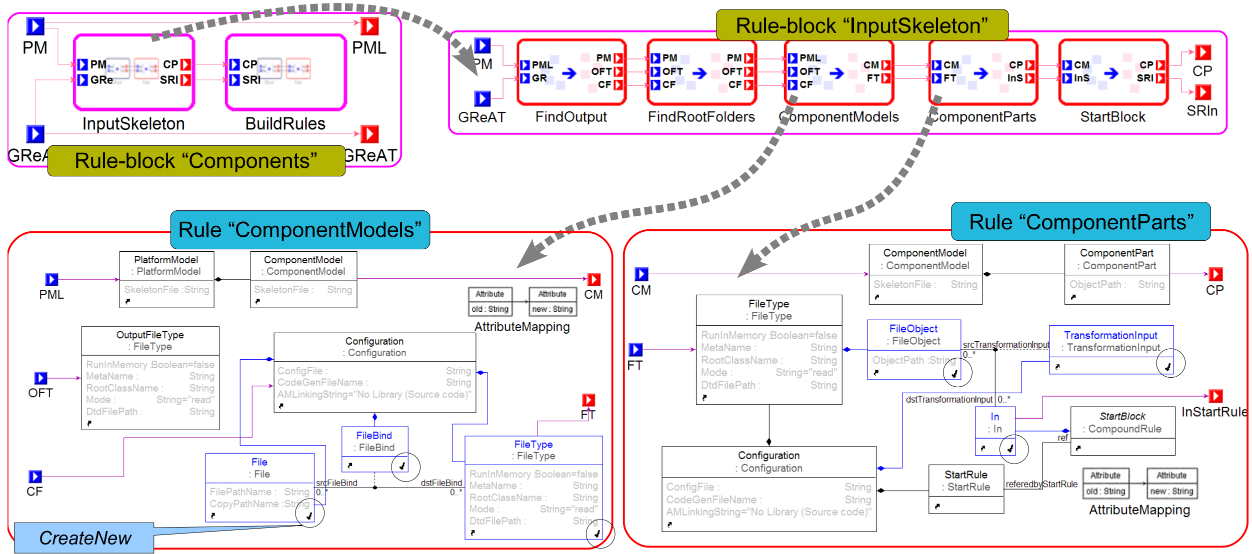


Figure 44: Rule-block Components with sub-block InputSkeleton

block `Components` is also extended with rules implementing component skeleton instantiation.

The two phases are executed by rule-blocks `InputSkeleton` and `BuildRules`, respectively. `InputSkeleton` extends the configuration models (illustrated in Fig. 44). `BuildRules` (Fig. 45) composes the instantiation rule patterns. (In the figures, the newly created objects are marked with a small circle).

`InputRules` creates a *(File,FileType)* pair for each component skeleton file into the configuration. For each ComponentPart within, a *FileObject* is created. The FileObject represents the top-level skeleton object within the model. It is propagated into the transformation via GReAT port objects and *Sequencing* connections. Thus, each skeleton object is made available for rule-block `Components`.

In `BuildRules`, the rule pair shown in Fig. 38 is created, and their patterns composed, according to the patterns in the ComponentModels and ComponentParts.

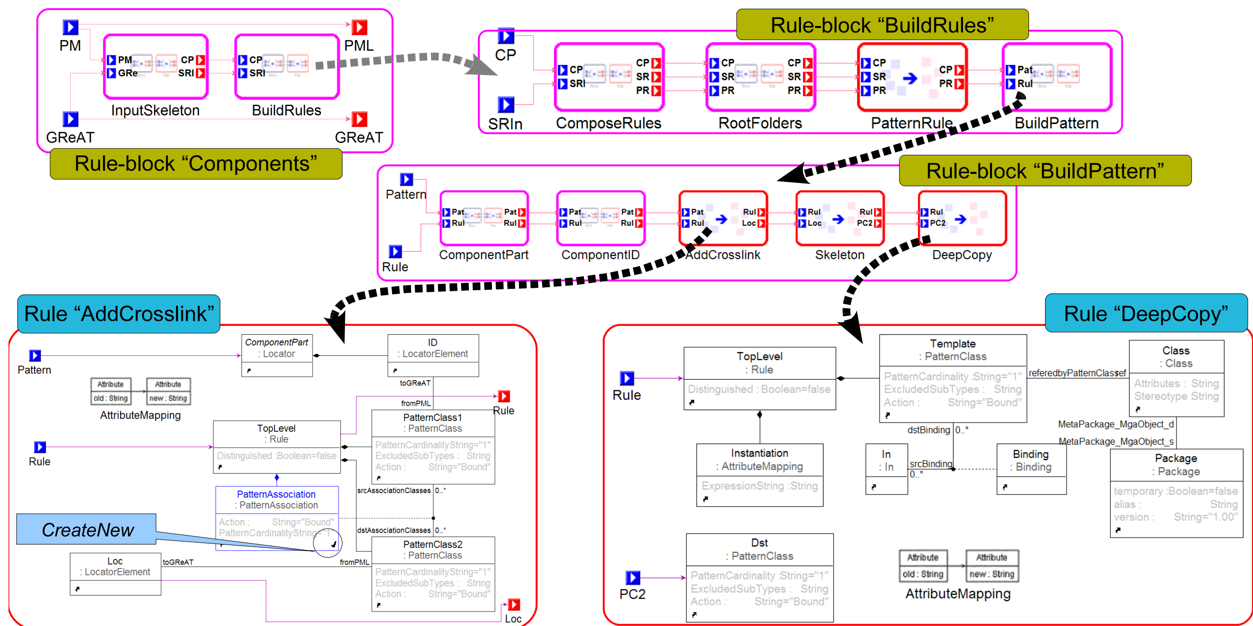


Figure 45: Rule-block BuildRules

Generating rules for mappings implementation

The final (and most complex) step for the compiler is to generate the rule sequence implementing PML mappings. The structure of the implementation was shown in Fig. 39.

The compiler traverses the mappings block hierarchy, and generates a Test block for each filter, and a rule for each action. Following *MatchReferences*, GReAT context passing paths are also generated.

The heart of the transformation is rule-block `CopyPattern`, shown in Figure 46. This block maps PML filter or action patterns onto GReAT test cases or rules. The block maps patterns following this sequence: works as follows:

1. First, all pattern objects (classes) are mapped, using the cross-metamodel associations established earlier.

2. Next, connections and associations are mapped. If necessary, this step also prepares a GReAT crosslink declaration

3. Finally, (for filters) GReAT output ports are created for MatchReferences.

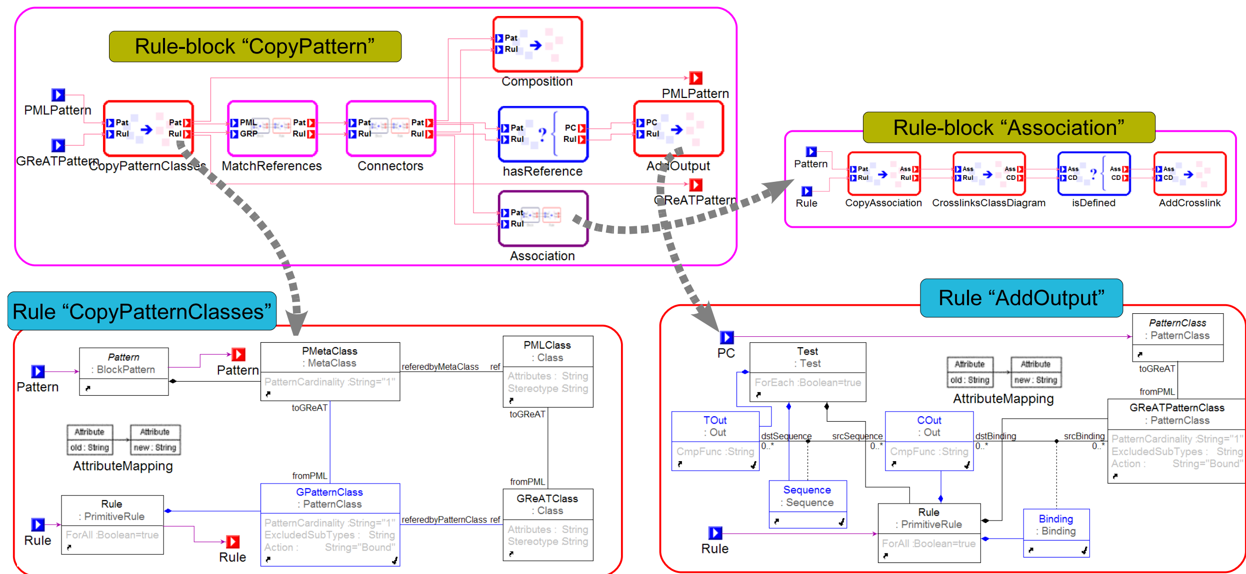


Figure 46: Rule-block CopyPattern within MapMappings

Conclusions

This chapter introduced, and argued for *explicit platform modeling*. After discussing the DSML → Platform → Analysis chain, the advantages of this approach were demonstrated (increased accuracy of the analysis models, and seamless integration into the MDA transformation chain).

Platform metamodeling was discussed, and an example presented, fitting to the case study introduced in the previous chapter. Taking arguments from the example, the two choices for platform metamodeling was presented (“generic” vs. “DSML-oriented”).

Next, the Platform → Analysis transformation was discussed in general, and a modeling language (PML) for capturing explicit platform models proposed. The language provides concepts very useful for platform modeling applications, but not present in GReAT, such as “skeleton” insertion.

For the PML language, a modeling framework was developed in GME. The framework contains the modeling language definition (paradigm), a few auxiliary tools (metamodel import tool, a model *decorator* for visualization). Most importantly, the framework also

contains a PML *compiler*, implemented in GReAT. This compiler creates a GReAT transformation based on the Platform → Analysis mapping captured in a PML model.

The rest of the chapter discussed the implementation of PML over GReAT, and the compiler performing the implementation.

CHAPTER VI

RESULTS AND FUTURE WORK

Platform Modeling

The most important result of this work is the introduction and discussion of platform modeling. As we had seen (e.g. in [47]), the platform concept has been conceptualized earlier. With the proliferation of cheap hardware and ubiquitous embedded computing, the need for platform-level analysis and verification is stronger than ever. Embedded system manufacturers, such as cell phone developers want to re-use their software components over evolving instances of hardware platforms, operating systems and middleware. They also push for standard- and model-based development efforts, in order to increase productivity and portability.

With the increase of the number of abstraction layers between application and computer, there is a demand to better understand systems. The formal modeling of interactions between layers and components, advocated by this work is definitely a solution for this problem.

The increased activity in the middleware community in this area (formal modeling of component interaction, such as the research presented in [50] and in related publications) also reinforces the view that undertaking this research was a step to the right direction.

The increasing complexity of computer-based systems also drives the need to study / model the systems from multiple aspects. The approach proposed in this work provides a very flexible component and platform concept. Using multiple platform models, analysis models exposing the system to multiple analysis aspects and methods can be constructed.

This work also provides an *automated* way for the construction of such analysis models. This construction is based on pre-configured templates captured within the *platform model*. Design engineers can automatically generate and check analysis models during the evolution of the design. Furthermore, using the proposed approach, the generation of analysis models

is integrated into the model-driven development flow, leveraging on many existing artifacts (such as metamodels, translators), and enhancing it with new ones (such as platform-level modeling). These new artifacts fit well into the development process, and might provide to be useful even beyond platform-level model analysis.

Explicit platform models

A major contribution presented here was the formalization of *explicit platform models*. These models assign formal semantics to platform-level structures. Using the demonstrated approach, platform-level structures can be handled at the *component level*, which is a natural abstraction. The demonstrated modeling language supports:

- The concept of a *component*, and the analysis mapping is centered around this concept.
- The specification and composition of analysis-model fragments ("skeletons") *at the analysis model's abstraction level*. This is important, since previous efforts captured these structures at the meta-level. Working at the modeling level makes studying these analysis model fragments ("component skeletons") allows analysis experts to work on these models. Analysis experts typically do not have metamodeling background, thus they are not comfortable working at the metamodeling level.

Specifying the Platform → Analysis mapping by the means of an explicit platform model (PML model) also makes the transformation simpler. Platform modelers do not have to cope with the sophisticated control flow mechanism supported by GReAT. PML is simple and clean.

In practice, developers could use PML to build platform model libraries, and validate designs over different platforms. With PML, quite often it is sufficient to modify the skeleton files only in order to capture a slightly different platform. This way, it becomes easy to customize platform models for different platform parameters, such as initalization delays or

resource limits. PML enables the investigation of the effects of such parameters without editing the analysis models.

Graph Transformation

Defining an UML-based graph transformation language on top of an existing one (and using the existing one to specify the compiler) was a novel work. The achievement is similar to defining a programming language, devising its implementation to assembly, and writing a compiler using the same assembly. The resulting system will not have “greater expressing power”, and will not be more efficient than assembly. Developers still prefer these higher-level languages over assembly, because they deliver useful abstractions and concepts users can leverage on.

This applies to the PML language and compiler demonstrated in this work as well. The demonstrated examples might have been too simple to drive the point through, but it should be clear that – for example – using UPPAAL fragments editable by the UPPAAL model editor is superior to using the meta-patterns of these structures in complex GReAT rules.

This work also contributed to the search of a practical GT language for MDA/MIC. It has been shown, that approaches and tools (such as GReAT) are mature enough to overtake such complex tasks as “writing a GReAT transformation that writes a GReAT transformation”. I hope that (beyond the numerous bug reports) I was able to contribute with useful feedback, ideas and feature requests for the developers of GReAT and GT approaches in general.

Future Work

Platform Modeling

Platform modeling (in the way proposed in this work) is a novel approach, and as such, the handful of examples and case studies presented here consitute the full body of experiences gained so far. The approach is promising and was able to deliver results even in this limited scope, but it is far from being solid.

The next step is to subject the approach (and the toolset) to real-life projects and perfect it using the experience gained. Medium-term plans include platform modeling experiments with time-triggered systems (the continuation of work started in [52] and [55]).

The extension for distributed systems with multiple kernel instances interacting through communication networks is also a direction worth pursuing.

So far, the only larger-scale experiments were conducted using timed automata analysis models, UPPAAL and IF ([16]. Platform models using different analysis languages should be specified and studied, such as discrete-time systems or hybid systems languages. A particularly interesting question here is how the complexity of the resulting plaform model could be controlled through the platform models used for generating it.

Improvements to the PML framework

The framework is also a prototype, and is of experimental nature. In order to make further case studies easier, several improvements could be made:

- Enhance the metamodel with constraints. This would help model editors by enforcing well-formedness rules, such as the prevention of "dangling" objects in patterns (objects not connected to anywhere).

- Provide means for metamodel evolution for UML diagrams embedded in PML models. For this, the GME *library* concept could be used.

- *Debugging* PML mapping rules is not easy, primarily because of the 2-layer abstraction (PML → GReAT →DSML). The PML language could be extended with "tracing" and "watch" primitives to show the progress of the mapping and spotlight attribute values.

Graph Transformation

One theoretically (and also practically) important question concerns with the verification of model transformations. In this area – generating analysis models – this is particularly important. The direction set out in [42] (checking bisimulation between the source and target models) seems very much relevant for platform modeling.

The presented PML compiler is a prototype implementation, and demonstrates how a transformation language with different semantics can be implemented over GReAT. As we could see, the implementation is simple and straightforward. The major drawback of this approach is low efficiency. It is mainly caused by the way non-deterministic PML action selection was implemented. In the demonstrated implementation, GReAT tries to evaluate the GFCs (Global Filter Conditions) for each PML action in parallel, which maps to a non-deterministic sequential order with the current GReAT implamentation.

There are several ways to improve this, the most promising one being to map the PML specification onto instructions for the *GR* language, which is the implementation platform for GReAT itself. This would be a straightforward but technically involved task.

As mentioned earlier, using GReAT at such an advanced level, this work has provided many insights to the application of GT in general. On the conceptual level, GReAT could be enhanced with additional idioms, such as "deep copy" I had to implement using the rather complex C++ API. A related concept, I found lacking was polymorphic or templatized rules. While working with GReAT, one often comes across recurring tasks such as hierarchy-flattening, polymorphic copy, or operations not dependent on the structure of the current

metamodel. Rule templates instantiated for concrete metamodels could save the recurring implementation of such tasks.

As a practical tool for programming, GReAT has been proved very capable. Still, for a language / framework to be fully accepted as a mature development tool it is necessary to have certain usability extensions, such as user-friendly editors or a debugger. GReAT has both, but both of them could use significant improvements. Many simple syntactic errors could be avoided by enhancing the GReAT metamodel with constraints. Also, the debugger should expose to some extent the internal workings of the pattern matcher. Currently, if a complex pattern does not match when it is expected to be, the only way to pinpoint the problem is to recode the pattern and split it up into parts which is rather time-consuming.

APPENDIX A

THE GME MODELING FRAMEWORK FOR PML

In order to study and demonstrate the feasibilty of the approach proposed in this work, a modeling environment was created for PML. The environment was specified in the GME [37] framework, with the following major parts:

- A GME metamodel for PML models
- A tool (`ImportMeta`) for importing GME metamodels (defining platform and analysis models) into PML models.
- A template (skeleton) GReAT transformation for PML→GReAT mappings
- A compiler transformation (implemented in GReAT). This compiler creates a GReAT transformation based on the mapping captured in the PML model. This transformation is discussed in detail in Chapter V.

The modeling language (metamodel)

PML models contain metamodels, crosslink definitions, component definitions with instantiation instructions, and mappings rules.

UML fundamentals

The higher-level concepts rely on UML patterns, using metamodel elements. Thus, a significant part of the metamodel is responsible for providing UML primitives. The PML metamodel uses the UML meta-library, available in the GME distribution, as a package of GME *paradigm sheets*, modeling UML. The same package is used internally by the GME meta-metamodel, and GReAT also uses this library.

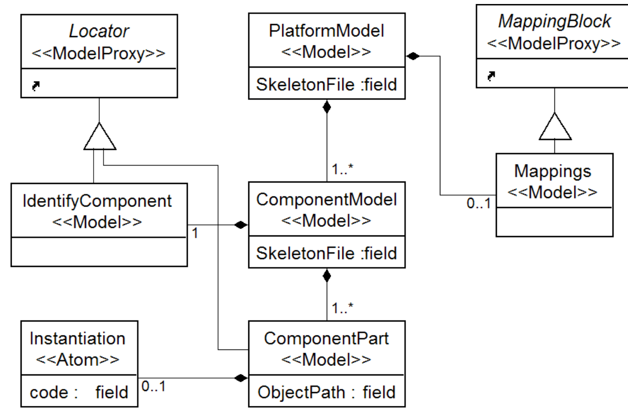


Figure 47: PML concepts in the metamodel

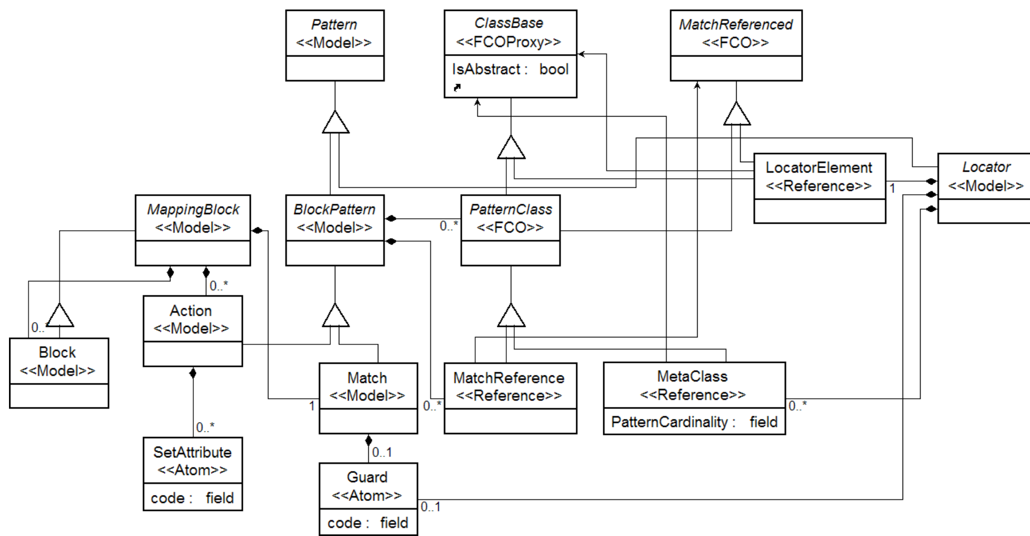


Figure 48: Hierachy of patterns in the PML metamodel

PML concepts

Figure 47 shows how the high-level concepts of PML are captured in the metamodel. A *PlatformModel* contains one or more *ComponentModels* and at most one *Mappings* blocks. ComponentModels have a *Locator* pattern to identify the component in the source model, and 1..∗ *ComponentParts*. ComponentParts are also Locator patterns (they designate one elements by a pattern), and may also contain an *Instantiation* block with C++ code.

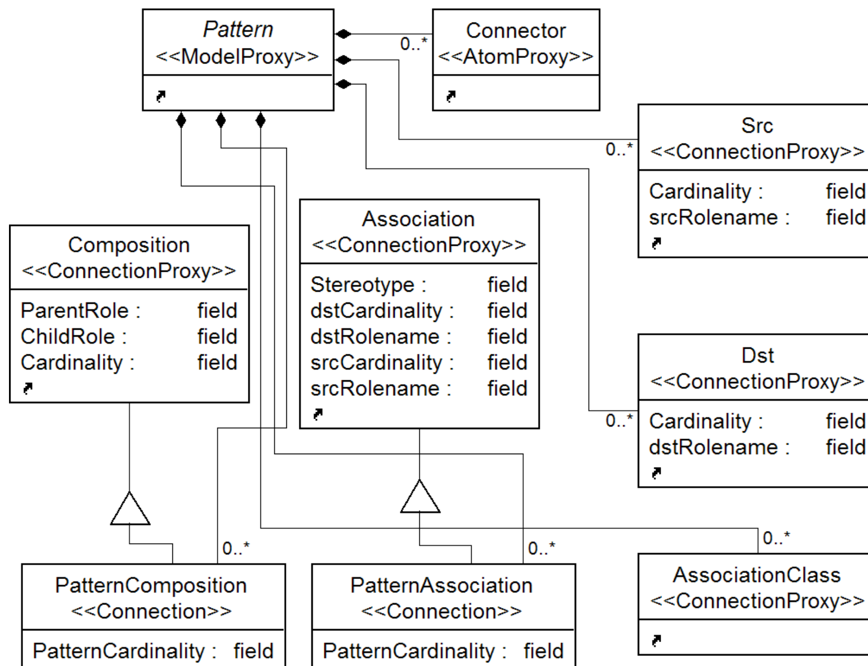


Figure 49: Patterns metamodel for PML

Pattern hierarchy

Figure 48 shows the different pattern objects employed by PML. From right to left: Each mapping *block* has exactly one *Filter* (with an optional Guard), and 0..∗ *Actions* (with optional SetAttr boxes). Filters and Actions are *BlockPatterns*, which may contain *MetaClasses* and *MatchReferences*. The main workhorse of the pattern language is the MetaClass: it refers to a class defined in the UML metamodel (*ClassBase*, on the top). Additionally, *Locator* patterns (such as ComponentIdentifiers and ComponentParts) have exactly one distinguished *LocatorElement*.

Pattern basics

In addition to the MetaClass and MatchReference elements, patterns also have elements to express their relation (associations). These are grouped into a different paradigmsheet, shown in Figure 49. Similar to the pattern language supported by GReAT, this language defines composition and association relations between pattern classes. These relations are derived from their respective class in the UML meta-package (*Composition* and *Association*)..

Auxiliary software in the PML framework

In order to make the PML framework easier to use, two additional software was developed.

The ImportMeta tool

The first one is `ImportMeta`, which is a tool to import a GME metamodel into a PML model. This tool was used to import the DFK and UPPAAL metamodels into the example models demonstrated in Chapter V. The tool was inspired by the tool with similar functionality in the GReAT framework, and it was implemented as a GReAT transformation.

The transformation reads a GME metamodel (source). It also opens the target PML model in *read-write* mode and creates a UML *Package* folder within, named by the metamodel. Within the package, a *ClassDiagram* is created for each GME ParadigmSheet, and the class and association definitions imported (copied). The tool also creates an abstract base class (*BaseClass*), and derives each class within the metamodel from it. Thus, the class is similar to *MgaObject* in GReAT transformations.

Decorator

A simple GME model decorator was also developed for improved model visualization. Decorators govern the visual appearance of models in the GME editor. The PML decorator specifies two non-default behavior:

1. *MatchReference* objects in patterns are shown with a dashed-frame rectange

2. *LocatorElements* in Locator patterns (such as ComponentIdentifier and ComponentPart) are shown with a shaded (grayed) rectange.

These visualization rules help understading PML patterns easier. Most of the PML patterns presented in figures in Chapter V are actual screenshots of the GME editor with PML models.

APPENDIX B

DETAILED PML EXAMPLES FROM THE DFK → UPPAAL MAPPING

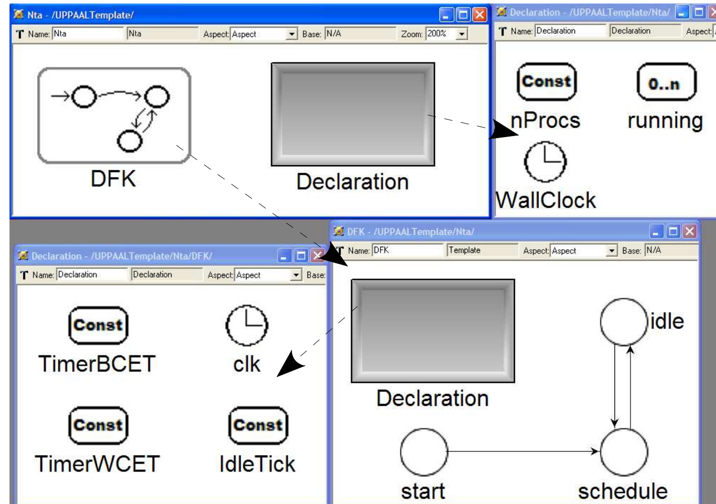


Figure 50: The DFK Kernel Skeleton for UPPAAL in GME

Kernel Skeleton

Figure 50 shows the kernel skeleton for the DFK → UPPAAL mapping:

1. The skeleton contains an UPPAAL *Nta* (Network of TA, top-level object).

2. This *Nta* contains one TA template definition (DFK) and a *Declarations* section for global variable declarations.

3. *Declarations* contains integer variable *running*, constant *nProcs* (initialized to 0), and a global clock ($WallClock$).

4. The DFK template contains the 3 kernel states (*start*, *schedule* and *idle* and their transitions), the declaration of a local clock variable (*clk*).

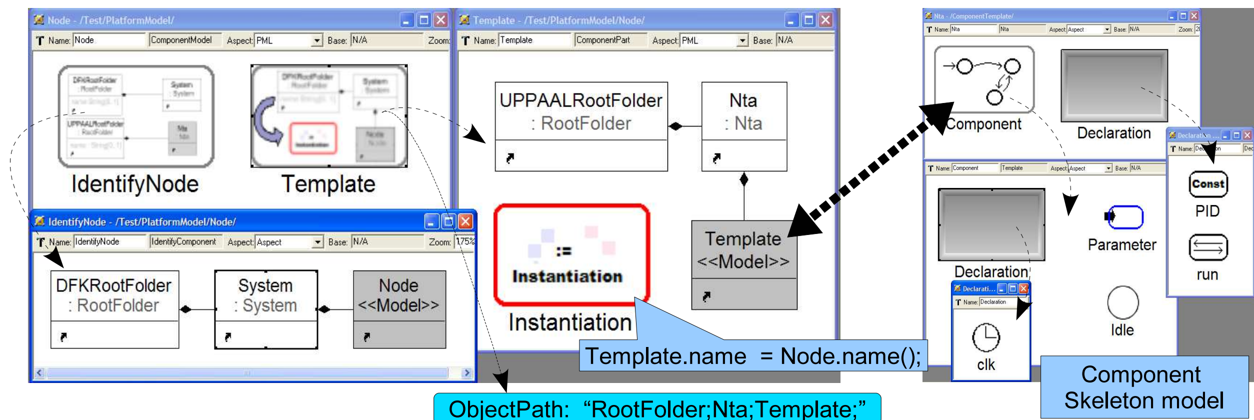


Figure 51: Component Skeleton for a Node in PML

During the subsequent mapping operations, the Nta will be extended with TA corresponding to the nodes of the DFK system, and the Kernel TA will be extended with states. Global and local variables will be also added, as described in the following sections.

Component skeletons

The ComponentModel responsible for instantiating a TA template for each DFK Node is shown in Figure 51. This ComponentModel has an associated filename, (`ComponentSkeleton.mga`). This file contains the UPPAAL component skeleton, shown on the far right.

- `IdentifyNode` on the top left contains the input pattern identifying a `Node` component (the *ComponentIdentifier*). The *Node* is designated (shaded rectangle) as the representative element of this component.

- `Template`, on the right from `IdentifyNode` is a *ComponentPart*, containing the definition of the target model fragment corresponding to *Node*. `Template` contains the following:

1. An *ObjectPath* identifier, to designate the object corresponding to $Node$ within `ComponentSkeleton.mga`. The the skeleton model is a syntactically correct UPPAAL model (so it can be edited within the UPPAAL modeling environment), and only a part of it is used as the component skeleton. This part is designated by the *ObjectPath* attribute. In this example the `ObjectPath` is `"RootFolder;Nta;Template;"` which navigates to the single UPPAAL TA defined in this model. Note that the global *Declaration* in the file is not part of this component skeleton designation, as it is not contained by the TA template.

2. A *ComponentLocator* pattern (`Template`), which specifies the location of the skeleton within the analysis model being composed. (`Template` is shown in the center of Fig. 51).

   The TA skeleton will be copied into this container. In the figure, the meta-pattern and the actual skeleton is associated by a thick dashed arrow on the right.

3. *Instantiation* is a procedural code fragment which is executed after the skeleton is copied into the target model. In the example it customizes the skeleton by renaming it.

Fig. 52 shows an additional part of the component model from Fig. 51. This definition locates the PID constant from the same skeleton model (`ComponentSkeleton.mga`), using the $ObjectPath$
`"RootFolder;Nta;Declarations;Const;"`. This “skeleton” (which is actually a single model element) is mapped it into the global declarations section of the output model using the pattern in the center of Fig. 52. The pattern also locates `nProcs` (using a *Guard* condition), a global constant maintaining the number of components in the system, and increments it for each component. The instantiation code is shown in the center top of the figure.

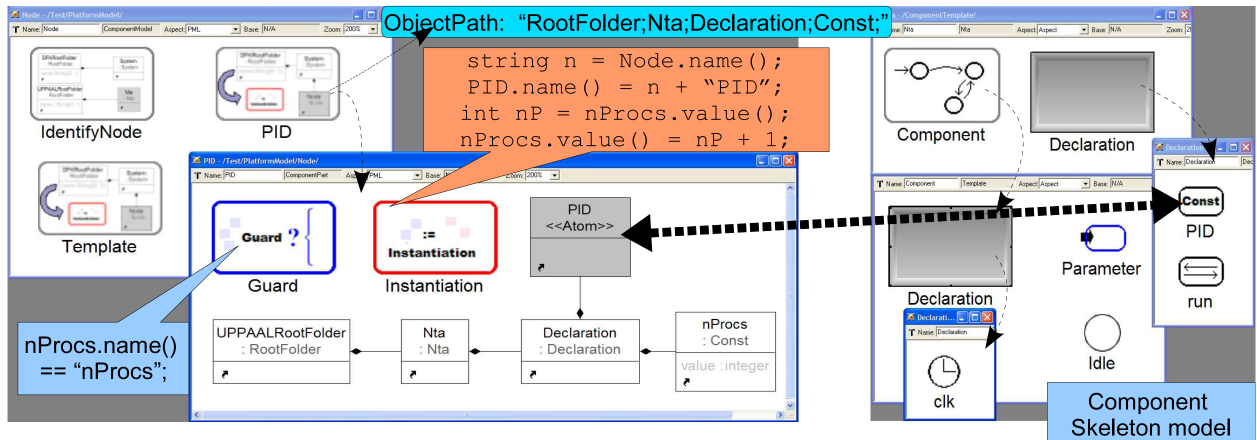


Figure 52: Component Skeleton part for a PID constant in PML

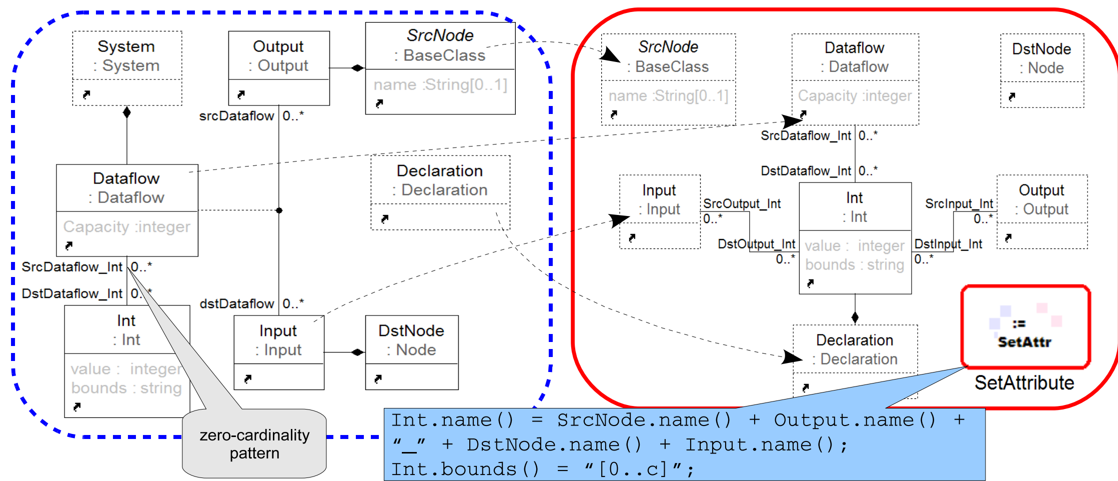


Figure 53: Dataflow→variable PML mapping block

Filter and action patterns

Figure 53 shows a typical *mapping block*. The block contains one filter (left, dashed frame) and one action (right, solid frame). It maps DFK dataflows onto UPPAAL integer variables, representing token count. The filter matches *Dataflows* (within a *System*) with no $DF \rightarrow DF_var$ associations only (*zero-cardinality match*).

The *action pattern* refers to instances identified by the filter using *MatchReferences* (indicated by dashed frame). Some of the MathReferences are visualized by dashed arrows. (Not all in order to avoid visual clutter). The patterns (with the dashed arrows and comments added) are actual screenshots form the GME implementation of PML.

For each match of the filter (each $Dataflow$ with zero associated Int objects) the action is executed, creating a new Int instance and creating the $Dataflow \rightarrow Int$ crosslink. In the filter objects with solid frame are MetaClasses (i.e. to be matched). In the action pattern, objects with dashed frame are MatchReferences to objects in the associated filter. Objects with solid frame (and associations) are created new on each action execution. Note that the Int object in the action is *not* a reference to the Int object from the filter (it as solid frame), since this object has to be created. Furthermore, no instance for the Int object in the filter can be matched, since it is only associated to bound objects with a zero-cardinality pattern, meaning that "no such instance exists".

Also, the pattern uses a *SetAttribute* box (expanded) to initialize the attributes of the new object (code partially shown).

The filter condition in this example is not a *top-level pattern*: It does not contain references to any of the $RootFolders$, thus it receives its context from a higher-level filter.

APPENDIX C

DETAILED EXAMPLES FOR THE PML → GReAT TRANSFORMATION

Component Skeleton instantiation examples

Figure 54 shows the overview of the rule-block implementing DFK Node →UPPAAL template component skeleton instantiation.

In the lower left corner, icons of the PML *ComponentModel* can be seen (three rounded rectanges arranged vertically). `IdentifyNode` contains the pattern identifying a "component" in the DFK model (shown expaned to the right). This pattern captures "what is a component" (how to recognise one) in the input model. In this example, it identifies each Node within a System, where the System is a top-level object in the DFK model (contained by the RootFolder).

For each *ComponentPart* model (`Template`, `PID`), a pair of rules are generated within rule-block `Components`. This rule-block is shown in the bottom right in the figure. It is also shown and related to the configuration in Figure 37 (within the block labeled "GReAT Transformation").

Each block receives four inputs (from top to bottom):

1. A reference to the RootFolder of the platform-level model (DFK).

2. References to the top-level objects of each component skeleton, as designated by their respective *ObjectsPaths* within the skeleton model(s). In this example, there are two of them:

   (a) A skeleton for the TA structure modeling a DFK node (`Template`).

   (b) A "skeleton" (a single object) for the PID constant (`PID`).

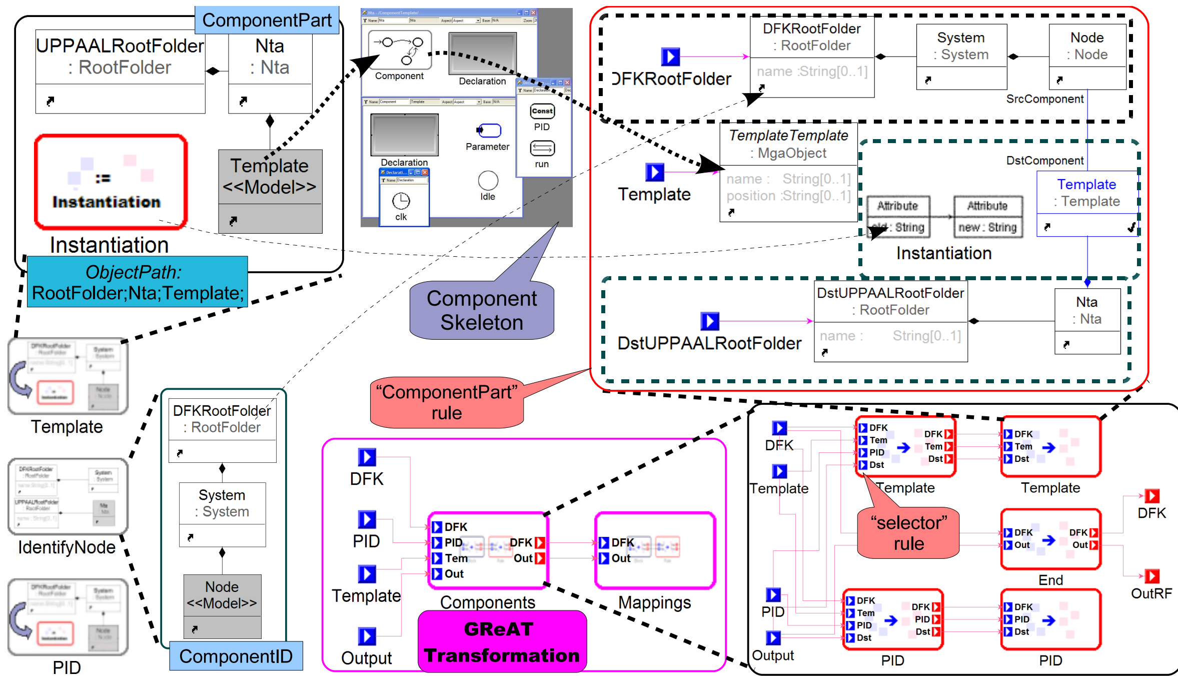


Figure 54: Component skeleton instantiation overview

3. A reference to the RootFolder of the analysis model (UPPAAL) being constructed. This model already contains the Kernel skeleton.

Within `Components`, the first rule of the pair simply selects one of the component skeletons for further processing (the details will be shown in Fig. 55, later in this section). The top-level objects of all skeletons (identified by the ObjectPaths in the skeleton model) are matched. One of them is propagated on, along with the RootFolders of the input and output models.

The next rule of the pair (expanded in the right top) shows how the instantiation is done:

- The *ComponentIdentifier* pattern is mapped into the GReAT rule to be matched against the input (DFK) model (top part of the rule shown on the top right in Fig. 54 in a dashed rectangle).

- The top-level object of the skeleton is propagated into the rule (`TemplateTemplate`) from the skeleton model.

- The *ComponentPart* pattern is also mapped into the GReAT rule (bottom part of the pattern, on top right in Fig. 54, in designated by a dashed, rounded rectangle) and matched against the output (analysis) model being built. The PML guard is mapped onto a GReAT guard, and the Instantiation becomes a part of the GReAT AttributeMapping box.

Thus, the above rule performs the following:

1. Locates and matches each "component" in the input model.

2. For each match, creates a corresponding new object within the output model (`Template` in the example), at the location designated by the *ComponentPart* pattern (within Nta in the example)..

3. Associates the new object with the "representative" object (`Node`) of the source component by creating a $(SrcComponent, DstComponent)$ crosslink.

4. Copies the contents of the component skeleton object (`TemplateTemplate`) into the newly created output model object.

Rule `End` (in rule-block `Components`, bottom right in 54) passes the context (the input and output RootFolders) on for the next rule-block implementing the Mappings rules. The rule also serves as a template for the compiler during constructing the above explained rules (i.e. references to both RootFolders are derived from this rule).

Figure 55 shows another example: the rule-blocks generated for the PID ComponentPart. This rule group creates a global PID constant for each TA Template (UPPAAL process), associated with a DFK Node. The most important rule is on the top right. It matches the ComponentIdentifier pattern (finds all the components in the DFK model). For each of them, it creates a PID constant, at the location determined by the ComponentPart pattern (bottom center). It also finds the global `nProcs` constant and increases it for each match. The figure also shows (top and center) how the skeleton object(s) are propagated from the

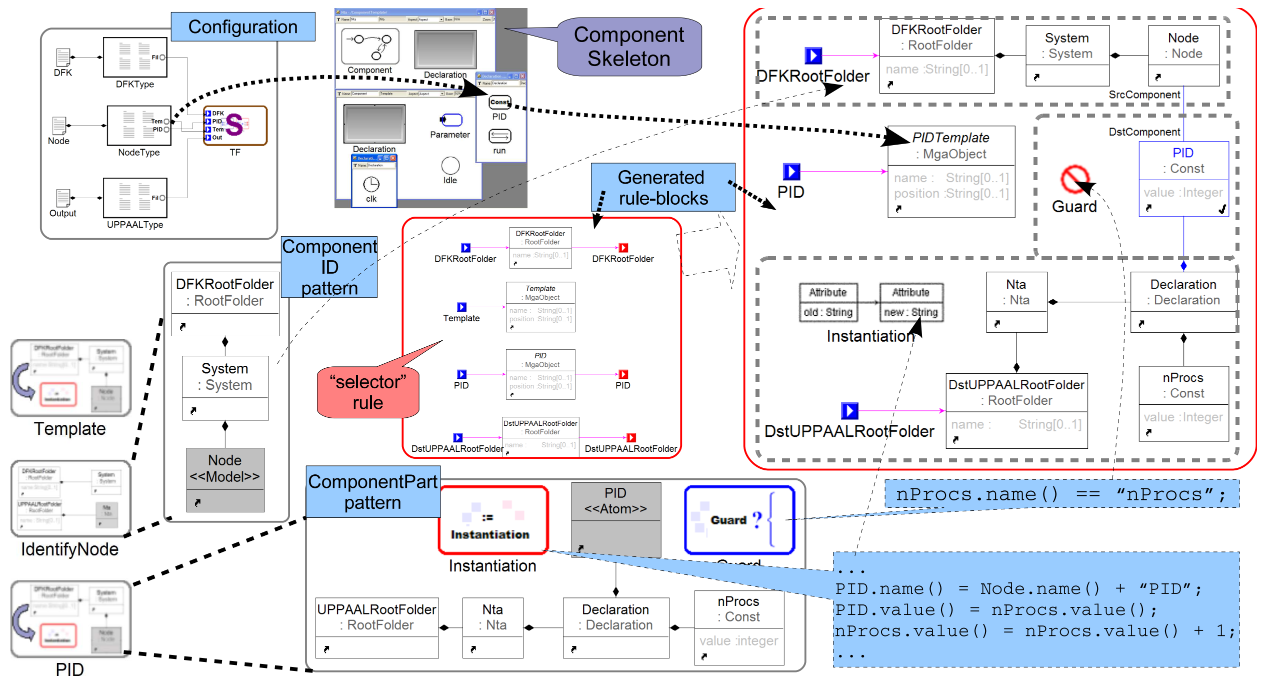


Figure 55: Generated rule-blocks for the PID skeleton

skeleton models via the configuration. On the bottom right, guard and instantiation C++ code fragments are shown. In the center, the expanded view of the "selector pattern". These patterns (one for each ComponentPart) select the template object for the instantiation rules (shown here on the top right). The relation of these selector patterns to the rest of the rules was shown in Fig. 54.

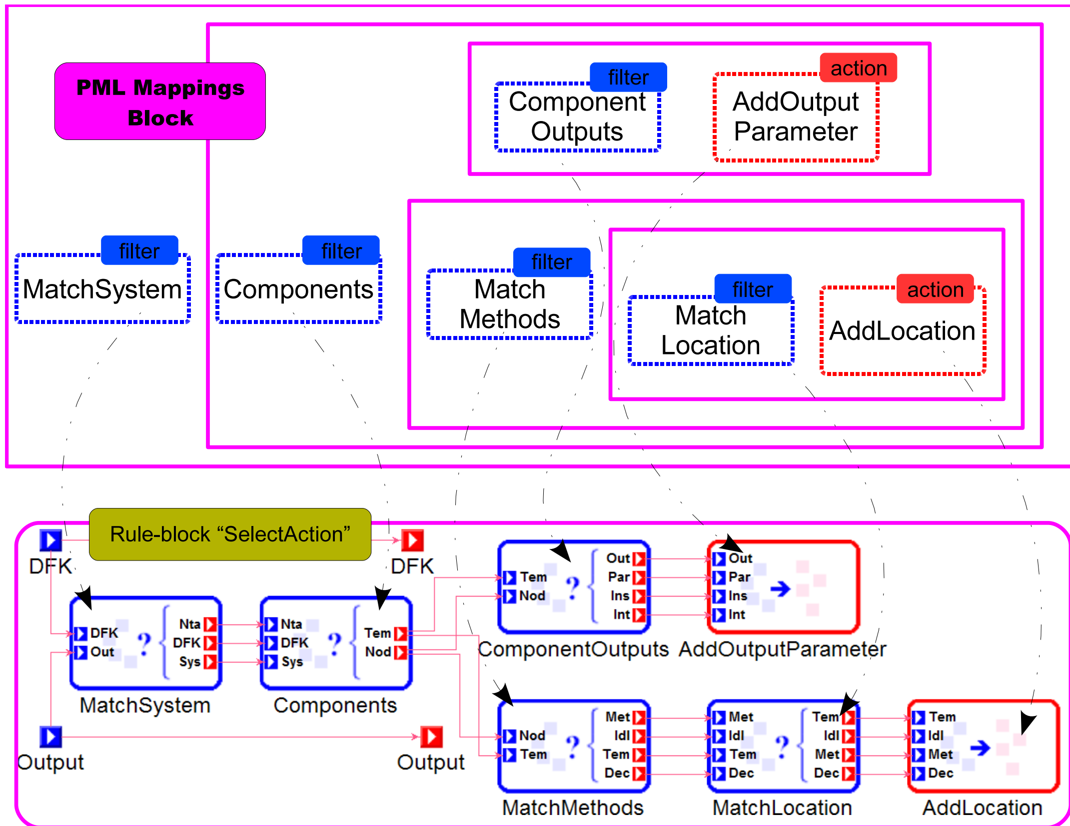


Figure 56: Example for GReAT rules generated for PML mapping patterns

Examples for the SelectAction rule-block

Figure 56 shows how the PML block hiearchy is mapped onto GReAT control flow. Test cases implement filter conditions, and these are chained in order to implement GFCs (Global Filter Conditions) along the block hierarchy. Each sub-block's filter is mapped onto a subsequent Test in the chain. Top-level rules match both RootFolders propagated from the previous rule-block (`Components`). For each PML *MatchReference* (elements referring to matches in the previous filter pattern) a GReAT context propagation path is generated.

The patterns in filter conditions are mapped onto GReAT Case patterns, as shown in Fig. 56. This is done by looking up the metamodel elements referred in the PML patterns, and finding the corresponding metamodels elements in the GReAT transformation. Then, GReAT patterns using PatternClasses and PatternAssociations are generated, referring to

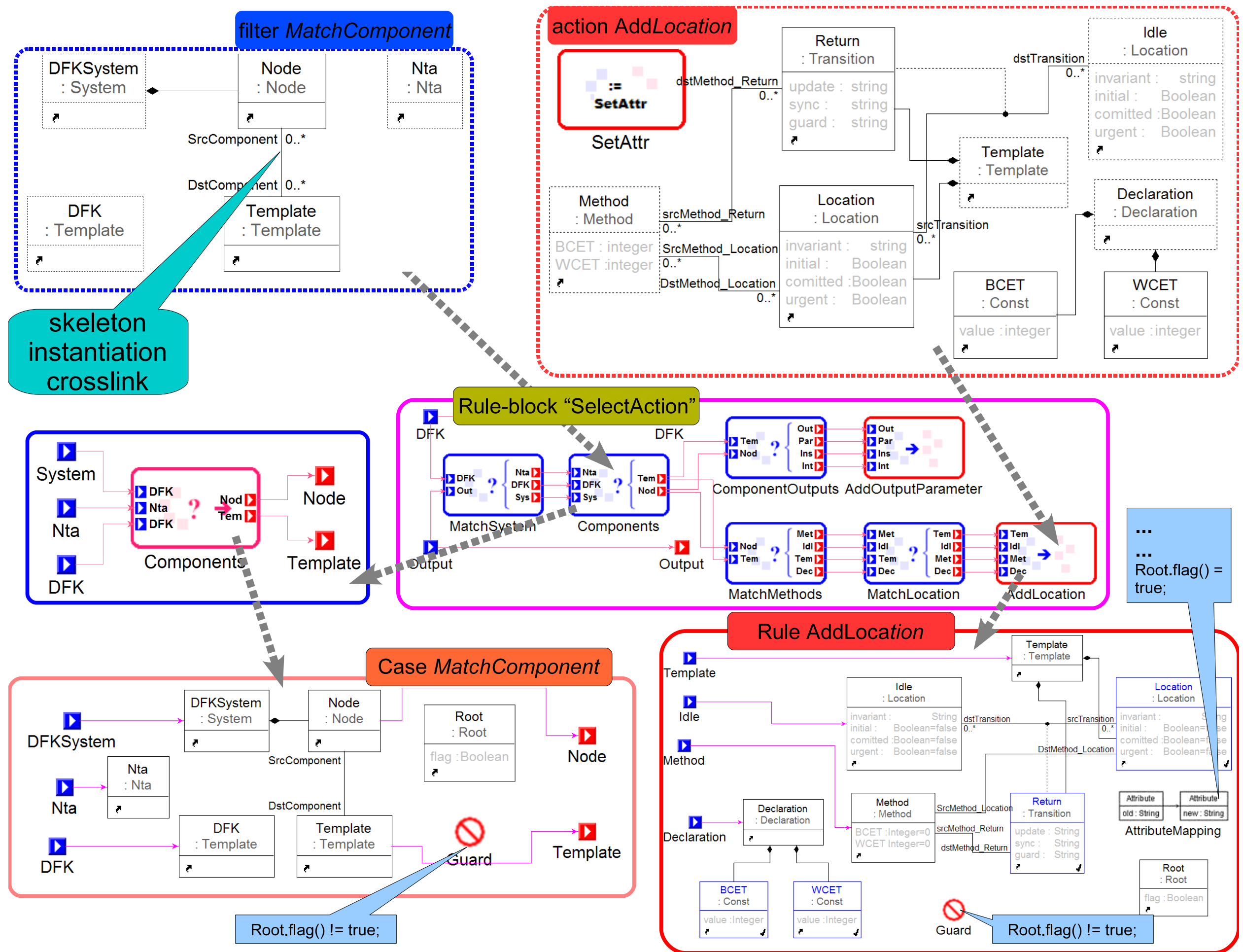


Figure 57: Details of GReAT Test cases and rules

the appropriate metamodel objects. Each element in a filter pattern is mapped onto a GReAT pattern object with the *Action* attributre set to *Bound* (i.e. to be matched within the context of the rule).

Rule(s) generated for action(s) terminate the rule chains. Action patterns are also mapped onto corresponding GReAT patterns. In action patterns, each element which is not a *MatchReference* is an element to be created in the output model. Thus, corresponding GReAT PatternClasses are generated with their *Action* attributre set to *CreateNew*. The same applies to all associations connected to them.

Figure 57 shows the details of the GReAT patterns generated by the compiler. The source PML patterns are shown on the top (a filter, left, and an action, on the right). The location

of the generated patterns within `SelectAction` is shown in the center. At the bottom, the GReAT patterns can be seen. The input bindings generated for *MatchReference* elements can be seen on the left. Filter patterns (test cases) also contain output bindings for downstream MatchReferences, these are shown on the right side of the pattern. The Guard conditions testing for the *Changed* flag are also shown (bottom). The C++ API call setting the flag is shown in the action rule (right side).For each action rule, an AttributeMapping box is generated, which contains the code from the PML *SetAttr* box, and the flag set operator is appended to this.

The example filter pattern (top left) also shows a $(SrcComponent, DstComponent)$ crosslink, generated by the component skeleton instantiation. The source $(Node)$ is the representative element of the component in the input model, and the target $(Template)$ is the top-level object of the skeleton inserted.

REFERENCES

[1] The MDA Website by OMG. Available online at `http://www.omg.org/mda`.

[2] OMG Model Driver Architecture Standards & Specifications
Website: `http://www.omg.org/mda/specs` .

[3] OMG The Meta Object Facility Homepage Website: `http://www.omg.org/mof/`.

[4] OMG XMI Specification Website: `http://www.omg.org/technology/documents/formal/xmi.htm`.

[5] The UPPAAL Website: `http://www.uppaal.com`.

[6] The GME Project's Website: `http://www.isis.vanderbilt.edu/Projects/gme/`.

[7] The Klasse OCL website. Available online at `http://www.klasse.nl/ocl`.

[8] NuSMV: a new symbolic model checker. Available online at `http://nusmv.irst.itc.it`.

[9] The sipher 2003 website. Online at `http://fountain.isis.vanderbilt.edu/teaching/2003`.

[10] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1992.

[11] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[12] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Sci. Comput. Program.*, 34(1):1–54, 1999.

[13] Eugene Asarin, Thao Dang, Oded Maler, and Olivier Bournez. Approximate reachability analysis of piecewise-linear dynamical systems. In *HSCC*, pages 20–31, 2000.

[14] Á. Bakay and E. Magyari. The UDM framework. Available online at `http://www.isis.vanderbilt.edu/Projects/mobies`.

[15] Beate Bollig and Ingo Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Trans. Computers*, 45(9):993–1002, 1996.

[16] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. Tools and applications: the if toolset. In M. Bernanrdo and F. Corradini, editors, *Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Real Time, SFM-04:RT*, volume 3185 of *LNCS*. Springer, 2004.

[17] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[18] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[19] J. R. Burch, E. M.Clarke, K. L. McMillian, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[20] Alongkrit Chutinan and Bruce H. Krogh. Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In *HSCC '99: Proceedings of the Second International Workshop on Hybrid Systems*, pages 76–90, London, UK, 1999. Springer-Verlag.

[21] Edmund Clarke, Masahiro Fujita, and Xudong Zhao. Applications of multi-terminal binary decision diagrams. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1995.

[22] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, Anaheim, CA, December 2003.

[23] D. Peled E. Clarke, O. Grumberg. *Model Checking*. MIT Press, 1999.

[24] Elena Fersman, Paul Pettersson, and Wang Yi. Timed automata with asynchronous processes: Schedulability and decidability. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 67–82, London, UK, 2002. Springer-Verlag.

[25] Agrawal A.and Karsai G., Kalmar Z., Neema S., Shi F., and Vizhanyo A. The design of a language for model transformations. *Journal of Software and System Modeling*, 2005.

[26] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In *ICGT*, pages 161–176, 2002.

[27] Thomas Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*, pages 278–292, New Brunswick, New Jersey, 1996.

[28] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2):110–122, 1997.

[29] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[30] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the use of graph transformation in the formal specification of model interpreters. *Journal of Universal Computer Science*, 9(11):1296–1321, 2003. `http://www.jucs.org/jucs_9_11/on_the_use_of`.

[31] Pavel Krčál and Wang Yi. Decidable and undecidable problems in schedulability analysis using timed automata. In Kurt Jensen and Andreas Podelski, editors, *Proc. of TACAS'04, Barcelona, Spain.*, volume 2988 of *Lecture Notes in Computer Science*, pages 236–250. Springer–Verlag, 2004.

[32] Jean J. Labrosse. *MicroC/OS-II: The Real-Time Kernel.* CMP Books, 1998.

[33] F. Laroussinie, N. Markey, and Ph. Schnoebelen. On model checking durational Kripke structures (extended abstract). In *Proc. 5th Int. Conf. Foundations of Software Science and Computation Structures (FOSSACS'2002), Grenoble, France, Apr. 2002*, volume 2303. Springer, 2002.

[34] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Model-checking for real-time systems. In *FCT '95: Proceedings of the 10th International Symposium on Fundamentals of Computation Theory*, pages 62–88, London, UK, 1995. Springer-Verlag.

[35] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[36] Dario Laverde, Giulio Ferrari, and Jurgen Stuber. *Programming Lego Mindstorms with Java.* Syngress, 2002.

[37] Ákos Lédeczi, Árpád Bakay, Miklós Maróti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, 2001.

[38] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on CAD*, 17(12), December 1998.

[39] Nicolas Markey and Ph. Schnoebelen. Symbolic model checking for simply-timed systems. In *FORMATS/FTRTFT*, pages 102–117, 2004.

[40] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[41] Christoph Meinel and Anna Slobodová. Speeding up variable reordering of OBDDs. In *International Conference on Computer Design*, pages 338–343, 1997.

[42] Anantha Narayanan and Gábor Karsai. Towards verifying model transformations. In *Proceedings for the Graph Transformation and Visual Modeling Techniques (GT-VMT), Vienna, Austria*, pages 185–194, 2006.

[43] Object ManagementGroup (OMG). Omg/rfp/qvt mof 2.0 query/views/transformations rfp. Misc, 2003.

[44] Detlef Plump. Termination of graph rewriting is undecidable. *Fundamenta Informaticae*, 33(2):201–209, 1998.

[45] Kekoa Proudfoot. Rcx internals. Technical report, Stanford University, 1998-99. Available online at `http://graphics.stanford.edu/~kekoa/rcx`.

[46] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 42–47, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[47] Alberto L. Sangiovanni-Vincentelli and Grant Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design & Test of Computers*, 18(6):23–33, 2001.

[48] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, Feb 2006.

[49] G. Smith. *The Object-Z Specification Language.* Kluwer Academic Publishers, 2000.

[50] Venkita Subramonian, Christopher Gill, Cesar Sanchez, and Henny Sipma. Composable time automata models for real-time embedded systems middleware. Technical report, Washington University, St. Louis, MO, USA, 2005.

[51] T. Szemethy. Implementing a mutithreaded, object-oriented multiplatform dataflow kernel. Technical report, Vanderbilt University, 2003.

[52] T. Szemethy. Model transformations for time-triggered languages. In *International Workshop on Graph and Model Transformation (GRaMoT 2005)*, Tallinn, Estonia, September 2005.

[53] T. Szemethy and G. Karsai. Platform Modeling and Model Transformations for Analysis. *Journal of Universal Computer Science*, 10(10):1383–1407, October 2004. `http://www.jucs.org/jucs_10_10/platform_modeling_and_model`.

[54] Janos Sztipanovits and Gabor Karsai. Model-integrated computing. *Computer*, 30(4):110–111, 1997.

[55] D. Balasubramanian T. Szemethy, G. Karsai. Model transformations in the model-based development of real-time systems. In *13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2006)*, Potsdam, Germany, March 2006.

[56] Ryan Thibodeaux. Exposing vanderbilt engineers to embedded systems modeling and analysis. Technical report, Institute for Software Integrated Systems, Vanderbilt University, 2005.

[57] Jos Warmer and Anneke Kleppe. *The object constraint language: precise modeling with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.