Simulation and Animation of Visual Languages based on Typed Algebraic Graph Transformation

Thesis · January 2006

DOI: 10.14279/depositonce-1417

1 author:

Claudia Ermel
Technische Universität Berlin
137 PUBLICATIONS **1,709** CITATIONS

SEE PROFILE

Simulation and Animation of Visual Languages based on Typed Algebraic Graph Transformation

vorgelegt von
Diplom-Informatikerin
Claudia Ermel

Fakultät IV – Elektrotechnik und Informatik –
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktorin der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. M. Alexa	(Technische Universität Berlin)
Berichter:	Prof. Dr. H. Ehrig	(Technische Universität Berlin)
Berichter:	Prof. Dr. H.-J. Kreowski	(Universität Bremen)

Tag der wissenschaftlichen Aussprache: 21. Juli 2006

Berlin 2006
D 83

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Herstellung und Verlag: Books on Demand GmbH, Norderstedt

Umschlaggestaltung: Leonard Ermel

ISBN-10: 3-8334-6321-X
ISBN-13: 978-3-8334-6321-1

“Then you should say what you mean,“ the March Hare went on. “I do,“ Alice hastily replied; “at least - at least I mean what I say - that’s the same thing, you know.“ “Not the same thing a bit!“ said the Hatter. “Why, you might just as well say that ‘I see what I eat’ is the same thing as ‘I eat what I see’ !“

Lewis Carroll. Alice’s Adventures in Wonderland (1865).

Abstract

In recent years, visual models represented by graphs have become very popular in systems development, as the wide-spread use of UML and Petri nets proves. Visual modeling techniques provide an intuitive, yet precise way to model the structure and behavior of systems at their natural level of abstraction. Validating model behavior is one of the main aims of visual behavior modeling. To express the semantics of visual models, the transformation of graphs plays a similar central role as term rewriting in the traditional case of textual models. The area of graph transformation provides a rule-based setting to define the syntax and semantics of visual models. The well-founded theoretical results support the formal reasoning about graph-based models at all levels.

The main objective of this thesis is to improve the validation process for visual behavioral models by providing a formal framework and tool support for simulation and animation based on graph transformation.

To attain this goal, we propose to use *animation views* for simulation instead of the notation of abstract diagrammatic languages. Animation views allow to simulate visual model behavior in the layout of the model's application domain, at a freely chosen level of abstraction. Thus, they provide better insights of model behavior and lead to an earlier detection of inconsistencies and possible missing requirements in the model. Moreover, *animation* of scenarios may be used, i.e. scenarios may be visualized as smooth movements. In the graph transformation framework, the model behavior is given by a graph transformation system (the *simulation specification*) typed over the visual language alphabet. An *animation view* defines the visual model's application domain by extending the alphabet. A simulation specification is mapped to an animation view by so-called simulation-to-animation model-and-rule transformation (*S2A transformation*). The formal description of simulation, animation and *S2A* transformation is based on the double-pushout approach to typed graph transformation. The formal basis is used not only to formalize visual models and their animation views, but also to reason about the semantical equivalence of the visual model's representation in the animation view.

Furthermore, the thesis describes the design and implementation of a prototypical tool environment for simulation, animation view definition, *S2A* transformation and animation. The existing generator for visual environments, GENGED, supports already the definition of visual modeling languages by type graphs and syntax grammars. The extensions of GENGED allow the definition of simulation specifications, scenarios, and animation views. *S2A* transformation is realized by applying so-called *meta rules* to the rules of an existing grammar. By the specification of continuous animation operations using the new *animation editor*, animation scenarios are visualized as smooth movements instead of discrete simulation steps.

Zusammenfassung

In den letzten Jahren haben visuelle Modelle in der Software- und Systementwicklung ständig an Bedeutung gewonnen, wie die weite Verbreitung von Diagrammtechniken wie UML und Petrinetze beweist. Bei der Formalisierung der Semantik visueller Modelle spielt die Transformation von Graphen eine zentrale Rolle. Das Gebiet der Graphtransformation bietet einen regelbasierten Rahmen zur formalen Definition von Syntax und Semantik visueller Modelle. Die wohlfundierte Theorie bildet die Basis für formale Resultate über Eigenschaften graphbasierter Modelle in verschiedenen Entwicklungsstadien.

Das Hauptziel dieser Arbeit ist es, den Validierungsprozess visueller Verhaltensmodelle zu verbessern, indem ein formaler Ansatz und eine Werkzeugunterstützung für Simulation und Animation, basierend auf Graphtransformation, entwickelt wird.

Um dieses Ziel zu erreichen, schlagen wir sogenannte *Animationssichten* (animation views) vor, die es erlauben, ausgewählte Aspekte des Modellverhaltens im Layout des entsprechenden Anwendungsbereichs zu visualisieren, anstatt, wie üblich, den Wechsel der Modellzustände durch Elemente der abstrakten Modellierungssprache darzustellen (wie z.B. durch Änderung der Markierung eines Petrinetzes). So gewinnt man einen besseren Einblick in das Modellverhalten und erkennt eher Unstimmigkeiten und bisher nicht erfüllte Anforderungen an das Modell. Darüberhinaus gestatten Animationssichten auch die *Animation* von Verhaltensszenarien, d.h. Zustandsübergänge werden als fließende Bewegungen dargestellt anstatt als diskrete Schritte. Formalisiert wird das Verhalten eines Modells als Graphtransformationssystem, die *Simulationsspezifikation*, getypt über dem Alphabet der visuellen Modellierungssprache. Eine Animationssicht erweitert dieses Alphabet durch Symbole des Anwendungsbereichs des Modells. Die Simulationsspezifikation wird in die Animationssicht abgebildet durch sogenannte *Simulation-Animation-Transformation* (*$S2A$-Transformation*). Die formale Beschreibung von Simulation, Animation und $S2A$-Transformation basiert auf dem Doppel-Pushout-Ansatz getypter Graphtransformation. Die formale Basis dient nicht allein der Formalisierung visueller Modelle und ihrer Animationssichten, sondern wird auch dazu genutzt, die semantische Äquivalenz der Diagrammrepräsentation eines Modells und seiner Darstellung in der entsprechenden Animationssicht zu zeigen.

Des weiteren beschreibt die Arbeit eine prototypische Werkzeugumgebung für Simulation, $S2A$-Transformation und Animation. Der existierende Generator visueller Modellierungsumgebungen, GENGED, unterstützt bereits die Definition visueller Modellierungssprachen durch Typgraphen und Syntaxgrammatiken. Die Erweiterung von GENGED gestattet die Definition von Simulationsspezifikationen, -szenarien und Animationssichten. $S2A$-Transformation wird realisiert durch die Anwendung sogenannter Meta-Regeln auf Graphregeln einer existierenden Grammatik. Über den neuen Animationseditor können Animationsszenarien kontinuierlich visualisiert werden.

Acknowledgements

First of all, I would like to thank my supervisor Hartmut Ehrig for introducing me to the fascinating field of graph transformation and for giving me the opportunity and freedom to pursue research of my own interest in the stimulating atmosphere of his research group TFS (Theoretical Computer Science / Formal Specification). His continuous attendance and his constructive support have been among the main resources throughout my studies. Moreover, he always encouraged me to publish relevant parts of my work at many international workshops and conferences and to participate at the European School on Graph Transformation and the School on Foundations of Visual Modelling Techniques, thus introducing me to the international graph grammar (GraGra) community.

I would also like to thank my co-supervisor Hans-Jörg Kreowski from Universität Bremen for accepting the task of examining this thesis. He also invited me to Bremen to give a talk and to discuss my work with his group. Apart from valuable advice, the visit led to joint work on animating UML model behavior. I thank Sabine Kuske, Karsten Hölscher and Paul Ziemann for this fruitful cooperation.

Furthermore, I am much obliged to my colleague Gabriele Taentzer not only for reading this thesis and for proposing numerous improvements, but also for sharing an office with me for so many years and for being there when needed. To my former colleague Rosi Bardohl, I owe the GENGED approach and tool environment. The visualization powers of GENGED inspired my idea of animation views. Formal concepts and their implementation based on GENGED thus could be developed hand in hand which turned out to be a fruitful procedure. I am also grateful to Karsten Ehrig for extending GENGED by an animation environment in his master thesis, and for his competent suggestions concerning the tool development part of my thesis.

Many thanks also go to present and former members of our research group TFS and to our guests on behalf of the European Research Training Network SEGRAVIS (Syntactic and Semantic Integration of Visual Modelling Techniques) for exciting and enlightening discussions, especially: Benjamin Braatz, Esther Guerra, Stefan Haensgen, Frank Hermann, Kathrin Hoffmann, Markus Klein, Andi Rayo Kniep, Leen Lambers, Juan de Lara, Toni Modica, Julia Padberg, Ulrike Prange, Leila Ribeiro, Olga Runge, Gunnar Schröter, Milan Urbášek, Daniel Varró, Szilvia Varró-Gyapay, Ingo Weinhold, Jessica Winkelmann. Thanks also go to Käte Schlicht and Mara Oswald for their help in any situation.

Last but not least, I would like to thank my husband Michael, my son Leonard, and my daughter Dorothee for their patience (sometimes) and their support. Leonard designed the cover after a beautiful idea of Dorothee. Michael also did a great job in proof-reading numerous pre-versions of the thesis. My family gave me the necessary diversion from my work which helped me not to lose the belief that there will be a life after the thesis.

Contents

Chapter 1

Introduction

Motivation

Software engineering aims at developing large software systems that meet quality and cost requirements. The development process that advances from the initial problem to the software solution is based on models that describe and support the development during the different phases. Models are key elements of many software engineering methodologies for capturing structural, functional and non-functional aspects. The particular strength of models is based on the idea of *abstraction*, i.e. a model is a simplified description of a complex entity or process. Usually the model is not merely related to one particular object or phenomenon, but to many, possibly to an infinite number of them, to a *class*. It is through models, by relating cause and effect, that we gain an understanding of nature. In engineering, models are used to combine elements of knowledge and data to make accurate predictions of future events.

Models may be formal (like the Turing machine), informal (like the specification of a software system by natural language), or semi-formal in the sense that the syntax and/or semantics of the models are only partly defined giving rise to different, sometimes even incompatible interpretations (like UML diagrams, the industry-standard notation for object-oriented analysis and design [UML04b]).

During the last decades the growing complexity of software systems led to a shift of paradigm in software modeling from textual to visual modeling languages [MM98, Erw98] which are used to represent aspects like e.g. distribution, components, parallelism and processes in a more natural way. The success of visual modeling techniques resulted in a variety of mostly semi-formal methods and notations (like the UML) addressing different application domains, perspectives and different phases of the software development process. The scientific community agrees on the need to improve current practice by increasing the degree of formality of these notations [Bot03, Mey97]. Two main approaches to

visual language definition can be distinguished: the declarative way, called *meta-modeling* and the constructive way, based on *(graph) grammars*. UML is defined by the Meta Object Facilities (MOF) approach [MOF05] which uses classes and associations to define symbols and relations of a VL. Additional well-formedness rules formulated in OCL [OCL03] complete the abstract syntax definition of a visual language. The concrete syntax level (the layout) and the semantics of the visual language features are not formalized. They are provided in the MOF approach by natural-language descriptions. While constraint-based formalisms such as MOF provide a declarative approach to VL definition, grammars are more constructive, i.e. closer to the implementation. In [MM98], for example, textual as well as graph grammar approaches are considered for VL definition. Textual grammar approaches (still in wide-spread use to define visual languages), such as picture-layout grammars [Gol91a, Gol91b] or constraint multiset grammars [Mar94], have to code multi-dimensional representations into one-dimensional strings. The use of graph grammars instead of textual grammars allows to represent multi-dimensional structures by graphs which is more natural. Moreover, the layout (or concrete syntax) of the language elements can be visualized directly. Analogously to the grammar-based definition of textual languages, graph grammars are used to define the structure of visual sentences as well as their construction.

A main aim in the software modeling process is the early validation of the model with respect to the informal user requirements. Validating the model is of paramount importance because although an implementation may be proven to be correct with respect to the specification, this is no help at all if the specification does not reflect adequately the user's needs and requirements. Validation of user requirements has traditionally been concentrated on testing program code prior to system installation. However, experiences have shown that requirement errors detected at late stages of the development process lead to a dramatic increase of the software costs. The main problem of requirements validation is that it concerns the interface between formality and informality. The validation process can only increase the confidence that the model represents a system which will meet the real needs of the user. Validation requires an active participation of users because they provided the original requirements. So users should be able to understand the model in order to find out possible misconceptions [KC95]. Because of the complex syntax and semantics, a limitation of formal specifications is that they cannot be readily understood by users unless they have been specially trained. A technique for facilitating the user participation in the validation process consists in exploiting the executability of formal models. The execution of behavior models is called *simulation*. Through simulation, users can observe, experiment and test the dynamic behavior of the model in different scenarios to see if it meets their real needs. In the context of visual behavior modeling, *simulation* means to show the before- and after-states of an action as diagrams of the visual language used to define the model. *Scenarios* (or simulation runs) then are given as sequences of actions, where

the after-state of one action is the before-state of the next action. In Petri nets, for example, a simulation is performed by playing the token game [Rei85]. Different system states are different markings, and a scenario is determined by a firing sequence of transitions resulting in a sequence of markings. Using graph transformation to define the behavior of visual models, simulation is specified by graph transformation rules [Kre81], and a scenario corresponds to a transformation sequence where the transformation rules are applied subsequently, beginning with a start diagram given as graph.

Despite these benefits, often the simulation of formal visual behavior models (e.g. Petri nets, graphs or Statecharts) is not adequate and can be ineffective in the model validation process. The simulation of a model based on (semi-) formal and abstract notations may not always be comprehensible to users, due to several reasons:

- Different aspects like control flow and data flow are represented in a similar way (a common problem in understanding Petri nets).
- Information belonging to one model is distributed in several sub-models based on different visual modeling languages (like in the UML). This can also lead to inconsistencies between the sub-models.
- Different levels of detail are represented in one model (like the visual model of a network protocol where both the components of the network (high relevance) and the substates of a minor connectivity check (low relevance) are represented at the same model level by the same graphical representation due to the chosen modeling technique.
- In order to be able to perform code generation or a complete formal analysis of the model, models have to contain details which raise their complexity.
- If the chosen modeling formalism does not allow to model a distinguished feature by an adequate model element, auxiliary elements (such as stereotypes in UML) or workarounds are used which make it difficult for other people but the modelers themselves to understand what is meant.
- The formal model may be (partly) generated automatically (like the graph transformation system generated from UML diagrams [ZHG04b]). Automatic model transformation necessarily involves auxiliary constructs and yields more complex diagrams than a hand-written specification would contain.

To this end, instead of simulating the model behavior within the modeling formalism itself, using a domain-specific layout (a *view*) for the simulation is one of the main propositions of this thesis. Using such a domain-oriented view, simulations can show changes directly, rather than having to indicate a state change in the abstract diagram indirectly by

using auxiliary markings such as arrows to the current state or by high-lighting the active diagram elements. The view removes the need for added markings so that displays can be simpler and less cluttered. In addition, the user does not have to interpret the auxiliary markings and try to infer the changes that they summarize. Such interpretation and inference may demand skills that the domain-oriented user does not possess. With depictions in the domain-oriented view, information about the changes involved is available to be read straight from the display without the user needing to perform mental animation.

Such a domain-oriented view is used in this thesis as basis to perform domain-specific animation of the model behavior. Hence, we call the view *animation view* [EE05b, EB04, EBE03]. *Animation* is the illusion of motion created by the consecutive display of images of static elements. Actions are visualized in a movie-like fashion, such that not only discrete steps (a before-state and an after-state) are shown but rather a continuously animated change of the scene. The popularity of using animations to help people understand and remember information has greatly increased since the advent of powerful graphics-oriented computers [Wik06]. Because animations can explicitly depict changes over time (temporal changes), they are well-suited to present processes and procedures. Animations can mirror both changes in position (translation), and the changes in form (transformation) that are fundamental to capture dynamic behavior.

Please note that in the fields of formal specification development and algorithm design, the term *animation* is often used in the sense of inspection, execution, testing or checking some property of a given set of test data [Utt00, Jia95]. Animations for visual models in the sense of continuous behavior visualization (as we use the term in this thesis) usually are realized either by ad-hoc implementations where a mapping from the formal model to a programming language is formulated by the animation programmer [GS00, JMBH93], or by program annotations where the code of a prototype implementation of the model is extended by operations that perform the graphical visualization of the current program state [Die01, SDBP98]. In both cases, the connection of the original model and its animation is a very loose one as the proceeding from the model to the animation is not formally defined. This may lead to animations that behave differently than the model and thus are not a reliable way for model validation. Moreover, both ways to proceed from the model to the animation require an implementation of the model behavior, thus performing the next development step from model to code before the validation of the model itself. This step may introduce new errors or inconsistencies in addition to those in the model which we want to discover by the validation.

Due to these limitations, this thesis proposes to *formally derive animation specifications* from visual models based on graph transformation, which has the following advantages: In contrast to algorithm or specification animation in the literature, in our approach the animation is defined *at modeling level*, i.e. at an early development step, and does not need an implementation of the model. The animation view for a model can be defined in

a flexible way, allowing free choice of the level of abstraction and of the domain-oriented layout. There is no restriction to a library of visual elements or actions. Having defined the elements of the animation view, the translation of diagram elements to the animation view is performed in a visual, rule-based way by a special kind of translation rules. The requirements for these translation rules ensure that the resulting animation specification is *semantically correct* in the sense that the transformation steps underlying the animation specification correspond to the semantics of the original model. The animation rules resulting from the translation may be extended by continuous animation operations.

Last but not least, our approach fits well in the generic approach of visual language definition by graph transformation. Hence, the approach is applicable to all possible kinds of visual modeling languages provided that the behavioral semantics of the visual models can be expressed by graph transformation rules.

Aims of the Thesis

The main objective of this thesis is to improve the validation process for visual behavior models. Using animation views for model execution instead of the notation of abstract modeling languages provides better insights of model behavior and may thus lead to an earlier detection of inconsistencies and possible missing requirements in the model.

Graph transformation is a formally defined calculus (see e.g. [EEPT06, Roz97]) and offers many well-founded theoretical results that support the formal reasoning about graph-based models at all levels. Graph transformation has a number of applications to system modeling and software engineering based on concrete specification languages and supporting tools [EEKR99, EKMR99, BTMS99]. In addition to formally describe the concrete and abstract syntax of visual modeling languages, graph transformation can also formalize the semantic aspects, and thus provides a strong basis for simulating and reasoning on diagrammatic models at all levels. Hence, we want to use those comprehensible formal concepts not only to formalize visual models and their animation views, but also to reason about the semantical equivalence of the visual model's representation in the animation view. More precisely, we rely on the double-pushout approach (DPO) for typed graph transformation systems (TGTS) [EEPT06]. Most of the presented results are available also for typed *attributed* graph transformation systems (TAGTS) [EEPT06], except for the use of a retyping functor between TGTS typed over different type graphs. This functor plays a role in some of the proofs of semantical correctness and equivalence of models and their corresponding behavior in the animation view. Applying typed graph transformation to visual language definition (as discussed in Section 2.2), usually needs attributed nodes. In fact, the sample application in this thesis all use attributes for naming or numbering of symbols. Fortunately, the concept of retyping can be extended to node attributes, if at-

tributes are not changed during transformations. This kind of typed attributed graphs can be defined by ordinary typed graphs. Data values are considered as nodes (data nodes) in contrast to object nodes denoting all other nodes of an attributed graph.

Furthermore, tool support for visual modeling based on typed attributed graph transformation is available in the form of the GENGED environment [Bar02], a meta-CASE tool for visual language definition based on the graph transformation engine AGG [Tae04]. The GENGED environment allows us to consider various visual behavior modeling languages such as Petri net and Statechart variants within one unifying VL modeling framework on the basis of typed attributed graph transformation systems. For the concrete syntax of visual languages, GENGED allows to define not only graph-like languages (where symbols are represented as icons and links between symbols are arcs or lines connecting these icons), but covers also more complex layout conditions like an *inside* relation between two symbol types, or flexible size or position dependencies between symbol graphics. Images in gif or jpg format for symbol graphics is also possible, which allows the alphabet designer to create arbitrary symbols modern graphical drawing tools, or even to use photos.

Against the background of the rich available theory for graph transformation, and the existing tool environment GENGED which covers already the foundations of visual modeling and flexible 2D-layouting based on graphical constraints, the main design decision of this thesis is to provide a formal framework and tool support for simulation and animation of visual models based on typed algebraic graph transformation.

In particular, the aims of this thesis are

1. to relate the behavior of visual behavioral models to *semantically correct simulation specifications* given by graph transformation systems,

2. to define an *animation view* for a visual model's application domain by extending the visual modeling language alphabet,

3. to map a visual model (and its simulation specification) to its animation view by so-called simulation-to-animation model-and-rule transformation ($S2A$ *transformation*), resulting in a graph transformation system called *animation specification*,

4. to develop requirements that ensure the semantical correctness of the $S2A$ transformation, and the equivalence of the original simulation specification and the resulting animation specification,

5. to enrich the resulting animation rules at the concrete syntax level to allow smooth animations when viewing animation scenarios, and

6. to apply the formal approach to diverse visual models based on different visual modeling languages (e.g. Petri nets, Statecharts) covering different application domains.

Main Results

The main results realizing the aims stated above are summarized as follows:

Simulation of visual behavioral models is described in three different ways by typed graph transformation systems (called *interpreter, compiler* and *amalgamation semantics*). We show the general semantical compatibility of the *interpreter semantics* to its translation into a model-specific compiler semantics, which is the form of semantics needed as basis for animation (Theorem 3.4.2). For formal behavior modeling languages such as Petri nets, the semantical correctness of the corresponding simulation specification is crucial. Hence, as a Petri-net specific result, we show the semantical compatibility of the firing behavior of Algebraic High-Level Petri nets (AHL nets) and their compiler and amalgamation semantics (Theorem 3.5.10 and Theorem 3.5.18).

Animation Views are defined by extending the alphabet of the visual modeling language (which is given as type graph) by elements visualizing the application domain (Def. 4.2.2). The nature of an animation view can be chosen to adapt to the needs of the user and/or to emphasize certain model aspects. In one application, an abstract model may be instantiated by a concrete, well-known application domain. An example for this kind of metaphorical animation view is to visualize the abstract producer-consumer system by showing a cook in a kitchen producing cakes and a person sitting at a table consuming these cakes [EB04]. In another application, certain details of the model algorithm simply may be omitted in an animation view which then shows only selected parts of a current model state (e.g. the broadcasting of messages in a network, modeled by the Echo algorithm).

$S2A$ transformation from simulation to animation specifications is defined on the basis of an animation view definition. We here use the characteristic of graph transformation, where the simulation of graphs is defined by rules which themselves consist of (rule) graphs. Hence, $S2A$ transformation is defined also by a typed graph transformation system, together with control conditions defining how to apply the $S2A$ transformation rules to the simulation rules (Def. 4.2.16). Requirements are stated which ensure the functional behavior of an $S2A$ transformation. In this context, termination of $S2A$ transformation is shown in general (Theorems 4.3.5 and 4.3.6). Furthermore, criteria for the semantical correctness and completeness of $S2A$ transformation are developed (Theorem 4.5.11 and Theorem 4.6.9) leading to the semantical equivalence of simulation and animation specifications (Theorem 4.6.15) which is based on a semantically correct $A2S$ backward transformation from animation to simulation specifications (Theorem 4.6.8).

Applications described in this thesis include simulation specifications for different kinds of Petri nets and Statecharts, modeling the behavior of an *Echo Algorithm* for checking computer network connectivity [BE03], the well-known *Dining Philosophers* [EE05b], a *Radio Clock*, an *ATM Machine*, and a *Drive-Through Restaurant* [KGKK02]. Based on the simulation specifications, suitable animation views and $S2A$ transformation rules are developed, and the semantical equivalence of the resulting animation specifications and their original simulation specifications is shown (Facts 4.6.16, 4.7.4, 4.7.8, 4.7.12).

Tool Support for our approach is realized as extension of the existing meta case tool GENGED [Bar02, Gen] for the generation of visual language environments. Simulation of visual model behavior is supported by a simulation environment allowing to apply simulation rules in a structured way. Animation view definition is supported by means allowing the integration and restriction of visual alphabets. $S2A$ model transformation is supported by application of so-called *meta rules* to the rules and the start graph of an existing grammar. The specification of *animation operations* for rules allows their execution to be visualized as smooth movements instead of discrete state changes. Animation scenarios can be exported to the Scalable Vector Graphics format SVG.

Overview of the Chapters

Chapter 2 (Definition of Visual Languages by Typed, Algebraic Graph Transformation) presents the formal background of the work, namely the basic notions and definitions of typed algebraic graph transformation and their use for the definition of visual languages (VLs). A running example of a VL for Condition/Event Petri nets illustrates the concepts for defining the abstract and concrete syntax by graph transformation (e.g. as VL alphabet and syntax grammar) and compares them to the meta-modeling approach which is common for VL definition in the UML. The *Applications* section contains further examples for VL definitions for different Petri net and Statechart variants.

Chapter 3 (Simulation of Visual Languages and Models) discusses three different ways to define model behavior and its simulation by typed graph transformation systems (TGTS), called *simulation specifications*, and compares the resulting semantics definitions for visual behavior languages and models. The *interpreter semantics* is defined by a set of general, model-independent simulation rules which can be applied to all models in a visual language, thus comprising a VL interpreter. The *GTS-compiler semantics* is given by a model transformation from a concrete model into the semantic domain of graph transformation systems, resulting in a set of model-specific simulation rules which are applicable only to one specific behavior model. The *amalgamation semantics* is defined by a combi-

nation of the other two approaches, based on a general set of model-independent simulation rule schemes which must be instantiated and amalgamated along a concrete model, to obtain a set of amalgamated simulation rules. Simulation in all three approaches is defined as execution of the simulation rules for a concrete model. We show that semantically equivalent model-specific simulation rules can be obtained from model-independent simulation rules or rule schemes by constructing rule instances over a so-called virtual model.

In the *Applications* section, we compare the GTS-compiler and the amalgamation semantics for the case of Algebraic High-Level nets (AHL nets) and show that both approaches are semantically equivalent. Furthermore, we define the interpreter and the amalgamation semantics for different Statechart variants, and discuss a GTS-compiler semantics for integrated UML models.

Chapter 4 (From Simulation to Animation) presents the extension of simulation specifications to animation specifications using $S2A$ transformation. On the basis of formal simulation specifications as defined in Chapter 3, Chapter 4 is concerned with the definition of animation views [EB02, EBE03, EB04] for a simulation specification, and with the concept of $S2A$ transformation realizing a consistent mapping from simulation to animation specifications. The most important properties of $S2A$ transformation are termination and semantical correctness and completeness, i.e. we show the correspondence between the behavior given by animation scenarios in the animation view and the original model behavior defined by the simulation specification. An Echo algorithm for testing computer network connectivity which is modeled as Condition/Event Petri net, serves as a running example. In the *Applications* section, animation views are defined for three further models of different visual modeling languages (a Radio Clock modeled as Statechart, the *Dining Philosophers* as AHL net, and a Drive-Through restaurant which was originally given by an integrated UML model and translated into a simulation specification [EHKZ05]).

Chapter 5 (Tool Support for Simulation and Animation in GENGED) describes the implementation for simulation and animation realized as extension of the VL environment GENGED [Gen]. At first, Chapter 5 gives a review of the graph transformation engine AGG and the GENGED environment for visual language definition, both developed at the TU Berlin [Tae04, AGG, Bar02, Gen]. We then describe the new components added to GENGED , namely a flexible graphical layouter, a simulator, and the support for animation view definition, integration and restriction on the basis of VL alphabets in GENGED. Moreover, an animation editor has been implemented [Ehr03] supporting the definition of continuous animation and the generation of scenario animations in SVG format.

Chapter 6 (Conclusion) summarizes the main achievements and outlines some open problems and directions for future work.

Chapter 2

Definition of Visual Languages by Typed Algebraic Graph Transformation

Textual specification languages are symbolic, and their basic syntactic expressions are put together in linear character sequences. In visual modeling languages (VLs), basic expressions include lines, arrows, circles and boxes, and composition mechanisms involve connectivity, partitioning, and "insideness". Visual modeling languages are proving extremely helpful in software and systems development. To be useful in computer-supported system development, any visual language must be defined by rigid rules that clearly state allowable syntactic expressions (or sentences) and give rigid description of their meaning.

Although visual modeling languages are becoming increasingly popular, there are controversial opinions about which notation would be best for describing them. For textual languages, using grammars for the syntax is widely accepted, but visual languages have two major competing approaches. One involves graph grammars [BTMS99], which extend grammar concepts from textual languages to diagrams. The other approach, called *metamodeling*, is based on MOF [MOF05] and uses UML class diagrams to model a visual language's abstract syntax. While class diagrams appear to be more intuitive than graph grammars, they are also less expressive. Therefore, metamodeling also uses context condition written in the Object Constraint Language (OCL) [OCL03] that help to overcome the weaker expressive power. The advantage of metamodeling is that UML users, who probably have basic UML knowledge, don't need to learn a new external notation to be able to deal with syntax definitions. But however intuitive the metamodeling technique is, using it to define UML is still limited to describing abstract syntax; the problems of diagram representations (concrete syntax) and of defining semantics remains.

Similar to grammars for textual languages, graph transformation systems on the one hand can formally describe the concrete *and* abstract syntaxes of modeling languages. On the other hand, they are also used to formalize the semantic aspects, and thus provide a strong basis for simulating and reasoning on diagrammatic models at all levels.

Graph transformation is a formally defined calculus (see e.g. [Roz97]) and offers many well-founded theoretical results that support the formal reasoning about graph-based models at all levels. Graph transformation has a number of applications to system modeling and software engineering based on concrete specification languages and supporting tools (cf. [EEKR99, EKMR99]).

The main idea of graph grammars and graph transformation is the rule-based modification of graphs where each application of a graph transformation rule leads to a graph transformation step. Graph grammars can be used on the one hand to generate graph languages by Chomsky grammars in formal language theory. On the other hand, graphs can be used to model the states of all kinds of systems which allows to use graph transformation to model state changes of these systems (which is in the focus of this thesis). Especially, graph transformation has been investigated as a fundamental concept for programming, specification, concurrency, distribution, visual modeling and model transformation.

The aspect to support visual modeling by graph transformation is one of the main intentions of the ESPRIT TMR Network SEGRAVIS (2002-2006). In fact, there is a wide range of applications to support visual modeling techniques, especially in the context of the UML, by graph transformation techniques.

A detailed presentation of different graph transformation approaches, is given in volume 1 of the *Handbook of Graph Grammars and Computing by Graph Transformation* [Roz97]. A state of the art report for applications, languages and tools for graph transformation on the one hand and for concurrency, parallelism and distribution on the other hand is given in volumes 2 and 3 [EEKR99] and [EKMR99].

The core of a graph transformation rule (or production) $p = (L, R)$ is a pair of graphs (L, R), called left hand side L and right hand side R. Applying the rule $p = (L, R)$ means to find a match of L in the source graph and to replace L by R leading to the target graph of the graph transformation. The main technical problem is how to connect R with the context in the target graph. In fact, there are different solutions how to handle this problem leading to different graph transformation approaches, which are summarized below.

Overview of Different Graph Transformation Approaches

1. The *node label replacement approach*, mainly developed by Rozenberg, Engelfriet and Janssens, allows replacing a single node as the left hand side L by an arbitrary graph R. The connection of R with the context is determined by embedding rules depending on node labels.

2. The *hyperedge replacement approach*, mainly developed by Habel, Kreowski and Drewes, has as the left hand side L a labeled hyperedge, which is replaced by an arbitrary hypergraph R with designated attachment nodes corresponding to the nodes

of L. The gluing of R with the context at the corresponding attachment nodes leads to the target graph.

3. The *algebraic approach* is based on pushout constructions, where pushouts are used to model the gluing of graphs. In fact, there are two main variants of the algebraic approach, the double and the single pushout approach. The algebraic double pushout approach, mainly developed by Ehrig, Schneider and the Berlin- and Pisa-groups, is the formal basis for simulation and animation in this thesis, and is reviewed in Section 2.1 in more detail.

4. The *logical approach*, mainly developed by Courcelle and Bouderon, allows expressing graph transformation and graph properties in monadic second order logic.

5. The *theory of 2-structures* was initiated by Rozenberg and Ehrenfeucht as a framework for decomposition and transformation of graphs.

6. The *programmed graph replacement approach* by Schürr used programs in order to control the nondeterministic choice of rule applications.

2.1 Typed Graph Transformation in the DPO Approach: State of the Art

In this section, the basic concepts and formalizations from the area of typed, algebraic graph transformation are reviewed.

Especially for the application of graph transformation techniques for visual modeling, *typed graph transformation systems* [HCEL96, CEL$^+$96] have proven to be an adequate formalism. A set of graphs is defined by a *type (scheme) graph* together with type-consistent operations on these graphs. This allows to model a visual language by a type graph, and sentences of the visual language by graphs typed over the type graph. The concepts of typed attributed graph transformation has recently been enhanced to *typed attributed graph transformation* [EPT04, Ehr04a]. Typed, attributed graphs are well suited for various applications in software engineering and visual languages.

Unfortunately, the theory for typed graph transformation concerning retyping and its properties cannot easily be extended to typed attributed graph transformation systems. Important categorical properties like the existence of categorical structures such as pullbacks do not hold in general for the category of typed attributed graph transformation systems due to the additional algebraic data type used for the attribution of nodes and edges. Hence, we decided in this thesis to build our approach for graph-transformation based simulation and animation on the formal basis of typed graph transformation systems. Nevertheless, in some of the presented applications we use node attributes, but in these cases we make sure

that the retyping of graph transformation systems always preserves the data type, so that the categorical constructions and results from typed graph transformation systems are still applicable for these applications.

Section 2.1.1 deals with graphs, graph morphisms and type graphs as well as the categories of graphs and of typed graphs. In Section 2.1.2 we present the gluing construction as the basis of graph transformation steps, and give the definition of typed graph transformation rules and typed graph transformation systems (TGTS). We discuss the applicability conditions for graph transformation rules and the conditions under which two direct transformations can be applied in arbitrary order or in parallel, leading to the same result. Section 2.1.3 defines morphisms of TGTS based on morphisms between type graphs and mappings between sets of rules. As a TGTS describes a specific behavior in terms of transformations obtained via application of its rules, an important property of TGTS morphisms is the preservation of the behavior in the sense that each direct transformation in the start TGTS is mapped to a direct transformation in the target TGTS. At last, in Section 2.1.4, we review the notion of *parallel graph transformation* which is defined by amalgamating rules at common subrule.

2.1.1 Graphs and Typing

A graph has nodes and edges that link two nodes. We allow parallel edges between two nodes as well as loops.

Definition 2.1.1 (*Graph*)
A *graph* $G = (V, E, s, t)$ consists of a set V of nodes (or vertices), E of edges and two mappings $s, t : E \to V$, the source and target functions.

$$E \underset{t}{\overset{s}{\rightrightarrows}} V$$

△

Remark: In the literature, a graph G is often represented by a set V of nodes and a set $E \subseteq V \times V$ of edges. This notion is almost the same as the one in definition 2.1.1: For an element $(v, w) \in E$, v represents its source and w its target node, but parallel edges are not expressible. □

Graphs are related by (total) graph morphisms, that map the nodes and edges of a graph to those of another one, preserving source and target of each node.

Definition 2.1.2 (*Graph Morphism*)
Given graphs G_1, G_2 with $G_i = (V_i, E_i, s_i, t_i)$ for $i = 1, 2$. A *graph morphism* $f : G_1 \to$

G_2, $f = (f_V, f_E)$ consists of two mappings $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ that preserve the source and target functions, i.e. $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$.

$$\begin{array}{ccc} E_1 & \overset{s_1}{\underset{t_1}{\rightrightarrows}} & V_1 \\ {\scriptstyle f_E}\downarrow & (=) & \downarrow{\scriptstyle f_V} \\ E_2 & \overset{s_2}{\underset{t_2}{\rightrightarrows}} & V_2 \end{array}$$

A graph morphism f is *injective* (*surjective*), if both mappings f_V, f_E are injective (surjective). f is called *isomorphic*, if it is bijective, that means both injective and surjective. △

Fact 2.1.3 (*Composition of Graph Morphisms*)
Given two graph morphisms $f = (f_V, f_E) : G_1 \to G_2$ and $g = (g_V, g_E) : G_2 \to G_3$, the composition $g \circ f = (g_V \circ f_V, g_E \circ f_E) : G_1 \to G_3$ is again a graph morphism. △

Proof: As a composition of mappings, $g_V \circ f_V : V_1 \to V_3$ and $g_E \circ f_E : E_1 \to E_3$ are well-defined. Using the associativity of the composition of mappings and that f and g as graph morphisms preserve the source and target functions, we conclude that

1. $g_V \circ f_V \circ s_1 = g_V \circ s_2 \circ f_E = s_3 \circ g_E \circ f_E$ and
2. $g_V \circ f_V \circ s_1 = g_V \circ s_2 \circ f_E = s_3 \circ g_E \circ f_E$.

Therefore also $g \circ f$ preserves the source and target functions. □

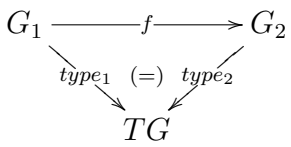
The class of all graphs (as defined in Def. 2.1.1) as objects and of all graph morphisms (see Def. 2.1.2) form the category **Graphs**, with the composition given in Fact 2.1.3 and the identities are the pairwise identities on nodes and edges.

A type graph defines a set of types that can be used to type the nodes and edges of a graph. The typing itself is done by a graph morphism between the graph and the type graph.

Definition 2.1.4 (*Typed Graph and Typed Graph Morphism*)
A *type graph* is a distinguished graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$. V_{TG} and E_{TG} are called node and edge label alphabet, respectively.

Then a tuple $(G, type)$ of a graph G together with a morphism $type : G \to TG$ is called *typed graph*.

Given typed graphs $G_1^T = (G_1, type_1)$ and $G_2^T = (G_2, type_2)$, a typed graph morphism $f : G_1^T \to G_2^T$ is a graph morphism $f : G_1 \to G_2$ such that $type_2 \circ f = type_1$.



△

Given a type graph TG, typed graphs over TG and typed graph morphisms form the category $\mathbf{Graphs_{TG}}$.

Example 2.1.5 (*Typed Graph*)
Consider the following type graph for Petri nets (without tokens) $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$ with $V_{TG} = \{Place, Trans\}$, $E_{TG} = \{ArcPT, ArcTP\}$, $s_{TG} : E_{TG} \to V_{TG} : ArcPT \mapsto Place, ArcTP \mapsto Trans$ and $t_{TG} : E_{TG} \to V_{TG} : ArcPT \mapsto Trans, ArcTP \mapsto Place$.

Then the graph G (a simple Petri net) together with the morphism $type = (type_V, type_E) : G \to TG$ with $type_V : V_S \to V_T : p_1, p_2 \mapsto Place; t \mapsto Trans$ and $type_E : E_S \to E_T : a_1 \mapsto ArcPT; a_2 \mapsto ArcTP$ is a typed graph (typed over TG).

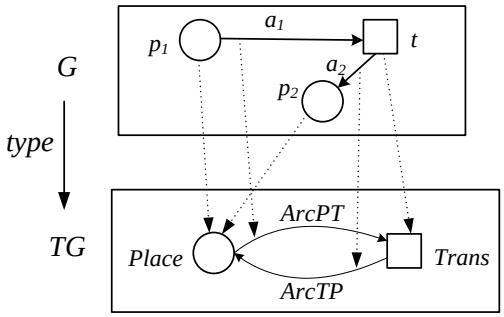


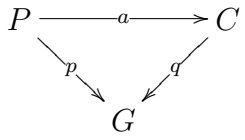
Figure 2.1: Type Graph for Petri Nets

△

In order to formulate properties that should be satisfied by a graph or by all graphs belonging to a specific set of graphs, so-called *graph constraints* can be defined. Graph constraints are useful e.g. for defining structural constraints of visual languages in a descriptive way (see Section 2.2.4.1). We distinguish *positive graph constraints* stating which properties have to be fulfilled by the graphs, and *negative graph constraints* for properties which the graphs must not have.

Definition 2.1.6 (*Graph Constraints*)
An *atomic graph constraint* is of the form $AC(a)$ where $a\colon P \to C$ is a graph morphism. A *graph constraint* is a Boolean formula over atomic graph constraints, i.e. every atomic

graph constraint is a graph constraint and, for graph constraints c and c_i with $i \in I$ for some index set I, $\neg c$, $\wedge_{i\in I} c_i$, and $\vee_{i\in I} c_i$ are graph constraints.
A graph constraint c can be *positive* (written $c = PC(a)$), or *negative* (written $c = NC(a)$).
A graph G *satisfies a positive graph constraint* $c = PC(a)$ if for every injective graph morphism $p\colon P \to G$ there exists an injective graph morphism $q\colon C \to G$ such that $q \circ a = p$. A graph G *satisfies a negative graph constraint* $c = NC(a)$ if for every injective graph morphism $p\colon P \to G$ there *does not* exist an injective graph morphism $q\colon C \to G$ such that $q \circ a = p$.



G *satisfies* $\neg c$ if G does not satisfy c. G *satisfies* $\wedge_{i\in I} c_i$ [$\vee_{i\in I} c_i$] if G satisfies all [some] c_i with $i \in I$. We write $G \models c$ to denote that G satisfies c. Two graph constraints c and c' are *equivalent*, denoted by $c \equiv c'$, if, for all graphs G, $G \models c$ if and only if $G \models c'$. △

2.1.2 Typed Graph Transformation Systems

In this section we introduce (typed) graph transformation systems. Graph transformation is based on graph transformation rules that describe in a general way how graphs can be transformed. The application of such a rule to a graph is called a *direct graph transformation*. For the application of graph transformation rules to a graph we need a technique to glue graphs together along a common subgraph. Intuitively we use this common subgraph and add all other nodes and edges from both graphs. The idea of a pushout generalizes the gluing construction in the sense of category theory, i.e. a pushout object emerges from gluing two objects along a common subobject (for a formal definition of pushouts and their properties, see Appendix A).

Definition 2.1.7 (*Typed Graph Transformation Rule and Rule Morphism*)
A *(typed) graph transformation rule* $p = (L \xleftarrow{l} I \xrightarrow{r} R)$ consists of (typed) graphs L, I and R, called left hand side, interface graph and right hand side respectively, and two injective (typed) graph morphisms l and r.

A *(typed) graph transformation rule morphism* $f = (f_L, f_I, f_R) : p \to p'$ is given by typed graph morphisms $f_L : L \to L'$, $f_I : I \to I'$ and $f_R : R \to R'$, such that $f_L \circ l = l' \circ f_I$ and $f_R \circ r = r' \circ f_I$.

$$\begin{array}{ccccc} L & \xleftarrow{l} & I & \xrightarrow{r} & R \\ {\scriptstyle f_L}\downarrow & (=) & \downarrow{\scriptstyle f_I} & (=) & \downarrow{\scriptstyle f_R} \\ L' & \xleftarrow[l']{} & I' & \xrightarrow[r']{} & R' \end{array}$$

Let $Rule_{TG}$ denote the set of all rules typed over the type graph TG. △

Special rule morphisms are embedding morphisms, where one rule is a subrule of another rule. There are two different notions of subrule embeddings,

Definition 2.1.8 (*Subrule Embeddings*)
Given TG-typed rules $p = (L \xleftarrow{l} I \xrightarrow{r} R)$, and $p' = (L' \xleftarrow{l'} I' \xrightarrow{r'} R')$ and a TG-typed rule morphism $f = (f_L, f_I, f_R) : p \to p'$ acc. to Def. 2.1.7 with f_L, f_I and f_R injective, we say that

- p is *DPB-subrule* (or *subrule*) of p' if the squares in the diagram in Def. 2.1.7 are pullbacks.
- p is *DPO-subrule* (or *strict subrule*) of p' if the squares in the diagram in Def. 2.1.7 are pushouts.

The rule morphism $f : p \to p'$ is called *(strict) subrule embedding*. In this context, p' is called *extending rule*. In the case that f_L, f_I and f_R are isomorphic morphisms, p is called isomorphic to p'. △

Definition 2.1.9 (*Graph Transformation*)
Given a (typed) graph transformation rule $p = (L \xleftarrow{l} I \xrightarrow{r} R)$ and a (typed) graph G with a (typed) graph morphism $m : L \to G$, called match. A *direct graph transformation* $G \overset{p,m}{\Longrightarrow} H$ from G to a typed graph H is given by the following double pushout (DPO) diagram, where (1) and (2) are pushouts.

$$\begin{array}{ccccc} L & \xleftarrow{l} & I & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & (1) & \downarrow{\scriptstyle i} & (2) & \downarrow{\scriptstyle m^*} \\ G & \xleftarrow[g]{} & C & \xrightarrow[h]{} & H \end{array}$$

A sequence $G_0 \Rightarrow G_1 \Rightarrow ... \Rightarrow G_n$ of direct graph transformations is called a *transformation* and is denoted as $G_0 \overset{*}{\Rightarrow} G_n$. For $n = 0$ we have the identical transformation $G_0 \overset{id}{\Rightarrow} G_0$. △

Now we define graph transformation systems and grammars. The language of a graph grammar are the graphs that can be derived from the start graph.

Definition 2.1.10 (*Graph Transformation System , Grammar and Language*)
A *graph transformation system* $GTS = (P)$ consists of a set of graph transformation rules P.

A typed graph transformation system $TGTS = (TG, P, \pi)$ consists of a type graph TG and a set of rule names P and a mapping $\pi : P \to Rule_{TG}$ associating with each rule name p a typed graph transformation rule $\pi(p)$.

A *(typed) graph grammar* $GG = ((T)GTS, S)$ consists of a (typed) graph transformation system GTS and a (typed) start graph S.

The *(typed) graph language* L of GG is defined by

$$L = \{G \mid \exists\, (typed)\ graph\ transformation\ S \overset{*}{\Rightarrow} G\}.$$

Let $Trans(GTS)$ [$Trans(TGTS)$] denote the set of all transformations over GTS [TGTS]. △

Remark: A graph grammar without its start graph is a graph transformation system. □

Example 2.1.11 (*Graph Grammar* Petri Net Drawing)
As an example we define a typed graph grammar GG_{Petri} generating Petri nets, i.e. graphs typed over the Petri net type graph TG shown in Fig. 2.1. One of the graphs of the typed graph language over this graph grammar is the graph G shown in Fig. 2.1. We define the graph grammar by $GG_{Petri} = (GTS_{Petri}, S_{Petri})$, where the start graph S is the empty graph (no nodes and no edges). The graph transformation system $GTS_{Petri} = (TG, P, \pi)$ is given by the type graph TG as shown in Fig. 2.1, and a set $P = \{addPlace, addTrans, insertArcPT, insertArcTP\}$ of Petri net drawing rules, shown in Fig. 2.2, which are typed over TG as well.

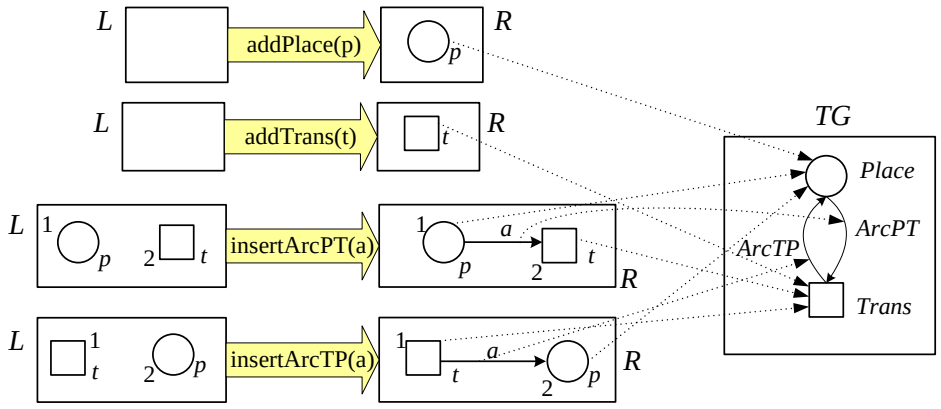


Figure 2.2: Typed Graph Transformation Rules for Petri Net Drawing

We use parameters in the rules that determine a partial match. The rule morphisms l and r are indicated by equal numbers for graph objects in a mapping. Note that we do not draw the interface graph of the rules explicitly. Implicitly, the interface graph is given by the graph objects that occur in L and R and have the same number. The typing of the rules is indicated in Fig. 2.2 by the type morphisms from the right-hand sides into the type graph TG. The left-hand sides and the interfaces are typed analogously.

The first two rules in Fig. 2.2 allow the drawing of a place or a transition. They are always applicable, as their left-hand sides are empty. The next two rules allow the insertion of an arc, either leading from a place to a transition or vice versa. These two rules require the existence of a place and a transition, where the respective arc is inserted between.

The sample Petri net G from Fig. 2.1 is the result of applying first the rules addPlace(p_1), addPlace(p_2) and addTrans(t) in any order, and afterwards the rules insertArcPT(a_1), where the place p in the rule is mapped by the match to place p_1 in the graph, and rule insertArcTP(a_2), where place p in the rule is mapped to place p_2 by the match morphism. △

2.1.2.1 Construction of Graph Transformations

Let us now analyse under which conditions a rule $p = (L \leftarrow I \rightarrow R)$ can be applied to a graph G via a match m. In general, the existence of a context graph C is required, that leads to a pushout. This allows to construct a direct transformation $G \overset{p,m}{\Longrightarrow} H$, where in a second step the graph H is constructed as the gluing of C and R via I.

Definition 2.1.12 (*Applicability of Rules*)
A (typed) graph transformation rule p is *applicable* to G via the match m, if there exists a context graph C such that (1) is a pushout.

$$\begin{array}{ccccc} L & \xleftarrow{l} & I & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & (1) & \downarrow{\scriptstyle i} & & \\ G & \xleftarrow[g]{} & C & & \end{array}$$

△

This definition gives us no criteria to decide wether the rule is applicable or not. A more constructive approach is to check the gluing condition. Both concepts are equivalent, as shown in the following fact.

Definition 2.1.13 (*Gluing Condition*)
Given a (typed) graph transformation rule $p = (L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R)$, a (typed) graph G and a match $m : L \rightarrow G$ with $X = (V_X, E_X, s_X, t_X)$ for all $X \in \{L, I, R, G\}$.

- The *gluing points* GP are those nodes and edges in L, that are not deleted by p, i.e. $GP = l_V(V_I) \cup l_E(E_I) = l(I)$.
- The *identification points* IP are those nodes and edges in L, that are identified by m, i.e. $IP = \{v \in V_L \mid \exists w \in V_L, w \neq v : m_V(v) = m_V(w)\} \cup \{e \in E_L \mid \exists f \in E_L, f \neq e : m_V(e) = m_V(f)\}$.

- The *dangling points* DP are those nodes in L, whose images are the source or target of an edge that does not belong to $m(L)$, i.e. $DP = \{v \in V_L \mid \exists e \in E_G \backslash m_E(E_L) : s_G(e) = m_V(v) \text{ or } t_G(e) = m_V(v)\}$.

p and m satisfy the *gluing condition*, if all identification and all dangling points are also gluing points, i.e. $IP \cup DP \subseteq GP$. △

Fact 2.1.14 (*Existence and Uniqueness of Context Graph*)
For a (typed) graph rule p, a (typed) graph G and a match $m : L \to G$ the context graph C with PO (1) exists $\Leftrightarrow$ the gluing condition is satisfied. If C exists, it is unique up to isomorphism. △

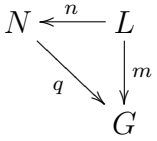
Remark: In categorical terms, the construction of C together with morphisms $i : I \to C$ and $g : C \to G$ is called the pushout complement of $l : I \to L$ and $m : L \to G$ leading to the PO (1) in the diagram above. □

Proof: see [EEPT06]. □

The applicability of rules can be further restricted by the use of *negative application conditions (NACs)* [HHT96] which only allow a rule application if some context specified in one or more additional NAC-graphs does not occur in the current graph.

Definition 2.1.15 (*Negative Application Condition*)

Let $p = (L \leftarrow I \rightarrow R)$ be a typed graph transformation rule. A *negative application condition (NAC)* (N, n) is a graph N together with an injective morphism $n : L \to N$. A match $m : L \to G$ satisfies the NAC if there exists no injective morphism $q : N \to G$ with $q \circ n = m$.



△

If a rule is applicable to a graph via a match, i.e. the gluing condition is fulfilled and all NACs are satisfied, then we can construct the direct graph transformation as follows.

Fact 2.1.16 (*Construction of Direct Graph Transformations*)
Given a (typed) graph transformation rule p and a match $m : L \to G$ such that p is applicable to G via m. Then the direct graph transformation can be constructed in two steps:

1. Delete those nodes and edges in G, that are reached by the match m, but have no preimage in I.

2. Add new nodes and edges, that are newly created in R.

This construction is unique up to isomorphism. △

Proof: see [EEPT06]. □

2.1.2.2 Local Church-Rosser and Parallelism Theorem for Graph Transformation Systems

In this section we study under which conditions two direct transformations applied to the same graph can be applied in arbitrary order leading to the same result. This leads to the notions of parallel and sequential independence of direct graph transformations and to the Local Church-Rosser Theorem. Moreover the corresponding rules can be applied in parallel in this case leading to the Parallelism Theorem. Both results have been shown already in the 70ies [ER76, EK76, Ehr79].

Definition 2.1.17 (*Parallel and Sequential Independence*)
Two direct graph transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ are *parallel independent* if all nodes and edges in the intersection of both matches are already gluing items with respect to both transformations, i.e.

$$m_1(L_1) \cap m_2(L_2) \subseteq m_1(l_1(I_1)) \cap m_2(l_2(I_2)).$$

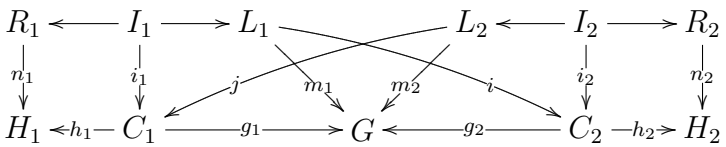
Two direct graph transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m_2}{\Longrightarrow} H_2$ are *sequentially independent* if all nodes and edges in the intersection of the comatch $n_1 : R_1 \to H_1$ and the match m_2 are already gluing items with respect to both transformations, i.e.

$$n_1(R_1) \cap m_2(L_2) \subseteq n_1(r_1(I_1)) \cap m_2(l_2(I_2)).$$

△

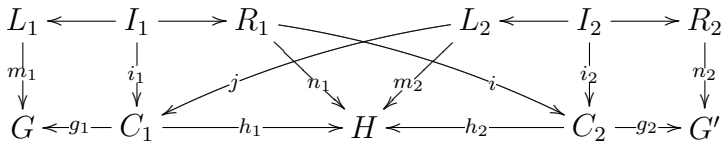
Remark: $G_1 \overset{p_1}{\Rightarrow} G_2 \overset{p_2}{\Rightarrow} G_3$ are sequentially independent iff $G_1 \overset{p_1^{-1}}{\Leftarrow} G_2 \overset{p_2}{\Rightarrow} G_3$ are parallel independent. □

Fact 2.1.18 (*Characterization of Parallel Independence*)
Two direct transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ are parallel independent $\Leftrightarrow$ there exist morphisms $i : L_1 \to C_2$ and $j : L_2 \to C_1$ such that $g_2 \circ i = m_1$ and $g_1 \circ j = m_2$.



△

Remark: With remark 2.1.2.2 the characterization of sequential independence follows: Two direct transformations $G \overset{p_1,m_1}{\Longrightarrow} H \overset{p_2,m_2}{\Longrightarrow} G'$ are *sequentially independent* $\Leftrightarrow$ there exist morphisms $i : R_1 \to C_2$ and $j : L_2 \to C_1$ such that $h_2 \circ i = n_1$ and $h_1 \circ j = m_2$.

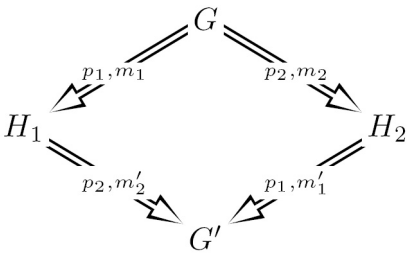


□

Proof: see [EEPT06]. □

Theorem 2.1.19 (*Local Church-Rosser Theorem for GT Systems*)
Given two parallel independent direct graph transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$, there is a graph G' and direct graph transformations $H_1 \overset{p_2,m_2'}{\Longrightarrow} G'$ and $H_2 \overset{p_1,m_1'}{\Longrightarrow} G'$ such that $G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m_2'}{\Longrightarrow} G'$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2 \overset{p_1,m_1'}{\Longrightarrow} G'$ are sequentially independent.

Given two sequentially independent direct graph transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m_2'}{\Longrightarrow} G'$, there is a graph H_2 and direct graph transformations $G \overset{p_2,m_2}{\Longrightarrow} H_2 \overset{p_1,m_1'}{\Longrightarrow} G'$ such that $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ are parallel independent.



△

Proof: see [EEPT06]. □

Definition 2.1.20 (*Parallel Graph Rule and Transformation*)
Given two (typed) graph rules $p_1 = (L_1 \overset{l_1}{\leftarrow} I_1 \overset{r_1}{\rightarrow} R_1)$ and $p_2 = (L_2 \overset{l_2}{\leftarrow} I_2 \overset{r_2}{\rightarrow} R_2)$. The *parallel graph rule* $p_1 + p_2$ is defined by the disjoint union of the corresponding objects and morphisms: $p_1 + p_2 = (L_1 \dot\cup L_2 \overset{l_1 \dot\cup l_2}{\longleftarrow} I_1 \dot\cup I_2 \overset{r_1 \dot\cup r_2}{\longrightarrow} R_1 \dot\cup R_2)$.

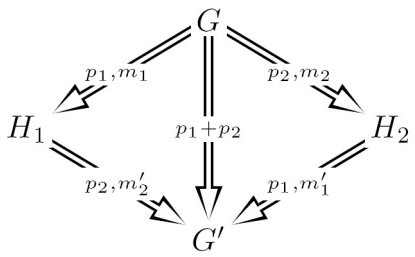
The application of a parallel graph rule is called a *parallel direct graph transformation*, or short *parallel graph transformation*. △

Theorem 2.1.21 (*Parallelism Theorem for GT systems*)
or a (typed) graph transformation system $GTS = (P; S)$ we have:

1. Synthesis: Given a sequentially independent direct graph transformation sequence $G \Rightarrow H_1 \Rightarrow G'$ via (typed) graph rule p_1 and p_2, then there is a parallel graph transformation $G \Rightarrow G'$ via the parallel graph rule $p_1 + p_2$, called synthesis construction.

2. Analysis: Given a parallel graph transformation $G \Rightarrow G'$ via $p_1 + p_2$, then there is a construction leading to two sequentially independent graph transformation sequences $G \Rightarrow H_1 \Rightarrow G'$ via p_1 and p_2 and $G \Rightarrow H_2 \Rightarrow G'$ via p_2 and p_1, called analysis construction.

3. Bijective correspondence: The constructions synthesis and analysis are inverse to each other up to isomorphism.



△

Proof: see [EEPT06]. □

The Parallelism Theorem is the basis for a shift construction which can be applied to parallel graph transformations. This allows to shift the application of a rule p within a parallel graph transformation t to the left (beginnning) as long as p is sequentially independent in t from the previous rules leading to a "canonical transformation". Construction and uniqueness of canonical transformations have been analyzed by Kreowski in [Kre78] leading to a concurrent semantics of algebraic graph transformation systems (see [Roz97]).

2.1.3 Retyping of Typed Graph Transformation Systems

In the literature there are various different proposals for morphisms of graph transformation systems [EHC05, GPS00, GPS99, HEET99, CEL$^+$96, Rib96, HCEL96]. They represent different objectives, like inclusions, projections or refinements, and have different semantic properties. The idea for behavioral refinement is to relate the transformation rules of the involved graph transformation systems. Große-Rhode et al. [GPS00, GPS99] propose

a refinement relationship between abstract and concrete rules that can be checked syntactically. They can prove that the application of a refined (concrete) rule expression yields the same behavior as the corresponding abstract rule. The slight disadvantage of this approach is that it cannot handle those cases where the refined rule expression should have additional effects on more concrete elements that do not occur already in the abstract rule. As in this thesis we are interested in defining views on systems by embeddings of typed graph transformation systems in the DPO approach, we will base our definitions in Chapter 4 on the notion of *weak refinement* in [GPS00, GPS99], which corresponds to TGTS embeddings in this thesis. To specify embeddings of graph transformation systems, an embedding morphism requires that the part of each extended rule which is typed over the type graph of the embedded rule, coincides with the embedded rule.

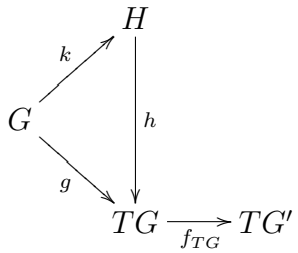
In order to express relations of graph transformation systems, TGTS embedding morphisms will be defined next which gives rise to a category of typed graph transformation systems. Hence, in this section we define the two main ingredients of TGTS morphisms, i.e., translations between type graphs and subrule relations. We start with forward and backward retyping using the notation of [GPS99].

Retyping of graphs w.r.t a change of the type system from TG to TG' is given via a graph morphism $f_{TG} : TG \to TG'$ that induces two operations: a backward retyping that maps TG'–typed graphs to TG–typed graphs, and a forward retyping in the other direction. Intuitively, the forward retyping is just another way of looking at the same graph, while the backward retyping, if f_{TG} is injective, removes all the edges and nodes of G which are mapped by the typing function to TG' but not to (the image under f_{TG} of) TG.

Definition 2.1.22 (*Retyping Functors*)
Let $f_{TG} : TG \to TG'$ be a type graph morphism. f_{TG} induces a *forward retyping functor* $f^{>}_{TG} : \mathbf{Graphs_{TG}} \to \mathbf{Graphs_{TG'}}$, defined as follows:

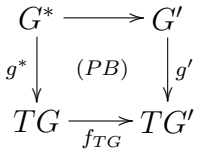
on objects: $f^{>}_{TG}((G, g : G \to TG)) = (G, f_{TG} \circ g)$,

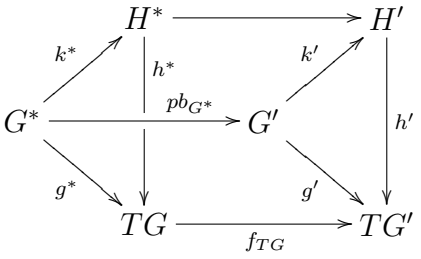
on morphisms: $f^{>}_{TG}(k) = k$, as shown in the following diagram:



f_{TG} induces a *backward retyping functor* $f^{<}_{TG} : \mathbf{Graphs_{TG'}} \to \mathbf{Graphs_{TG}}$, defined as follows:

on objects: $f^{<}_{TG}((G', g' : G' \to TG')) = (G^*, g^* : G^* \to TG)$, where G^* is the pullback object in $\mathbf{Graphs}$ of g' and f_{TG} in the diagram to the left,
on morphisms: $f^{<}_{TG}(k' : G' \to H') = k^* : G^* \to H^* \in \mathbf{Graphs_{TG'}}$, by the pullbacks $G' \times_{G^*} TG = TG'$ and $H' \times_{H^*} TG = TG'$. The existence and uniqueness (up to isomorphism) of the morphsm k^* follows from the universal property of the pullback object H^*, as shown in the diagram to the right.





△

Fact 2.1.23 (*Adjunction*)

1. Forward retyping is a functor $f^{>} : \mathbf{Graph_{TG}} \to \mathbf{Graph_{TG'}}$, with the identity given by $id^{>}_{TG} = id_{\mathbf{Graph_{TG}}}$ and the composition $(e \circ f)^{>} = e^{>} \circ f^{>}$.

2. Backward retyping is a functor $f^{<} : \mathbf{Graph_{TG'}} \to \mathbf{Graph_{TG}}$, with the identity given by $(id_{TG})^{<} \cong id_{\mathbf{Graph_{TG}}}$ and the composition $(e \circ f)^{<} \cong f^{<} \circ e^{<}$.

3. Forward and backward retyping functors are left and right adjoints, i.e. for each $f_{TG} : TG \to TG'$ we have $f^{>}_{TG} \dashv f^{<}_{TG} : \mathbf{Graphs_{TG'}} \to \mathbf{Graphs_{TG}}$.

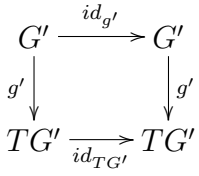
△

Proof: (from [GRPS97a, GRPS97b, BK06])

1. Identity: $id^{>}_{TG}((G, g)) = (G, id_{TG} \circ g) = (G, g)$

   Composition: $(e \circ f)^{>}(k) = k = e^{>}(k) = e^{>}(f^{>}(k))$

2. The diagram to the right is a pullback, composition of pullbacks is a pullback, and pullbacks are unique up to natural isomorphism.



3. (*Proof Sketch:*) It has to be shown that there is a natural isomorphism $u : (\_, f^<(\_)) \cong (f^>(\_), \_)$. This means, that there must be $u((G,g),(G',g')) : (G \xrightarrow{i} f^<(G',g')) \mapsto (f^>(G) \xrightarrow{j} (G',g'))$ and $u^{-1}(G,G') : (f^>(G) \xrightarrow{j} (G',g')) \mapsto (G \xrightarrow{i} f^<(G',g))$ which are sets of isomorphisms of morphism sets. $u(G,G')$ is constructed by $u(G,G')$ $(G \xrightarrow{i} f^<(G',g')) = pb_{G^*} \circ i$, and $u^{-1}(G,G')(f^>(G) \xrightarrow{j} (G',g'))$ is constructed by the induced pullback morphism from G to G^* of the pullback $G' \times_{TG'} TG = G^*$. Now it can be shown that the functions $u(G,G')(i)$ and $u^{-1}(G,G')(j)$ are inverse by using the defining properties of pullbacks. (For a full proof please see [BK06].)

□

An embedding morphism of typed graph transformation systems must first of all relate their type systems, i.e. it must contain a type graph inclusion morphism. The backward retyping functor then induces translations to compare the rules of both systems.

Definition 2.1.24 (*TGTS Morphism, TGTS Embedding*)
Given typed graph transformation systems $TGTS = (TG, P, \pi)$ and $TGTS' = (TG', P', \pi')$, a $TGTS$ morphism $f = (f_{TG}, f_P) : TGTS \to TGTS'$ is given by a type graph inclusion morphism $f_{TG} : TG \to TG'$, and a mapping $f_P : P \to P'$ between sets of rule names, such that for each $p \in P$, $\pi(p) = f^<_{TG}(\pi'(f_P(p)))$.

We call the TGTS morphism f *TGTS embedding* if f_{TG} is injective and write $TGTS = TGTS'|_{TG}$ if there is a TGTS embedding $TGTS \xrightarrow{f} TGTS'$. △

Fact 2.1.25 (*Category* $\mathbf{TGTS}$)
Typed graph transformation systems and $\mathbf{TGTS}$ morphisms form a category, called $\mathbf{TGTS}$.
△

Proof:
identity: The identity of a typed graph transformation system $TGTS = (TG, P, \pi)$ is given by $id_{TGTS} = (id_{TG}, id_P)$.

composition: Given two $\mathbf{TGTS}$ embedding morphisms $f : TGTS_1 \to TGTS_2$ and $g : TGTS_2 \to TGTS_3$, their composition is defined componentwise by $g \circ f = (g_{TG} \circ f_{TG}, g_P \circ f_P) : TGTS_1 \to TGTS_3$. We show that the composition is well-defined: The type graph morphism $g_{TG} \circ f_{TG}$ is composed of two type graph inclusions, and is thus also a type graph inclusion from the type graph TG_1 of $TGTS_1$ to the type graph TG_3 of $TGTS_3$. The mappings f_P and g_P of rule names satisfy the conditions c1: $\pi_1(p) = f^<_{TG_1}(\pi_2(f_P(p)))$ and c2: $\pi_2(f_P(p)) = f^<_{TG_2}(\pi_3(g_P(f_P(p))))$. We have to show that the composition of the backward retyping functors $f^<_{TG_3} = f^<_{TG_1} \circ f^<_{TG_2}$ applied to a rule $\pi_3(g_P(f_P(p))) \in TGTS_3$

yields rule $\pi_1(p)$:

$$\begin{aligned} f^<_{TG_3}(\pi_3(g_P(f_P(p)))) &= f^<_{TG_1}(f^<_{TG_2}(\pi_3(g_P(f_P(p))))) \\ &= f^<_{TG_1}(\pi_2(f_P(p))) \quad \text{due to cond. c2} \\ &= \pi_1(p) \quad \text{due to cond. c1} \end{aligned}$$

□

Fact 2.1.26 (*TGTS Embedding Preserves Injective Morphisms*)
Let $f = (f_{TG}, f_P) : TGTS \to TGTS'$ be a TGTS embedding according to Def. 2.1.24, and the morphism $G_1 \xrightarrow{m} G_2$ be an injective graph morphism in $\mathbf{Graphs_{TG'}}$. Then, the morphism m is mapped by the backwards retyping functor to an injective morphism $f^<_{TG}(m)$. △

Proof: As $G_1 \xrightarrow{m} G_2$ is injective, diagram (1) is pullback in $\mathbf{Graph_{TG'}}$ (see Property A.6, 2). As $f^<_{TG}$ is right adjoint functor, it preserves pullbacks. Hence, diagram (2) is a pullback in $\mathbf{Graph_{TG}}$, and thus, the morphism $f^<(m)$ is injective.

$$\begin{array}{ccc} G_1 & \xrightarrow{id} & G_1 \\ {\scriptstyle id}\downarrow & (1) & \downarrow{\scriptstyle m} \\ G_1 & \xrightarrow[m]{} & G_2 \end{array} \qquad \begin{array}{ccc} f^<_{TG}(G_1) & \xrightarrow{id} & f^<_{TG}(G_1) \\ {\scriptstyle id}\downarrow & (2) & \downarrow{\scriptstyle f^<_{TG}(m)} \\ f^<_{TG}(G_1) & \xrightarrow[f^<_{TG}(m)]{} & f^<_{TG}(G_2) \end{array}$$

□

The most important property of TGTS embeddings is the reflection of behavior, where behavior is given by the set of all transformations $Trans(TGTS)$. The behavior reflection theorem for TGTS embeddings $f : TGTS \to TGTS'$ compares the behavior $Trans(TGTS')$ to the behavior $Trans(TGTS)$. Each transformation in $Trans(TGTS')$ using rule $f_P(p)$ gives rise to the corresponding transformation in $Trans(TGTS)$ using rule p.

Theorem 2.1.27 (*Reflection of Behavior*)
Let $f = (f_{TG}, f_P) : TGTS \to TGTS'$ be a TGTS embedding and $G' \overset{\pi'(f_P(p))/m'}{\Longrightarrow} H' \in Trans(TGTS')$ a direct transformation in $Trans(TGTS')$, where $f^<_{TG}(G') = G$ and $f^<_{TG}(m') = m$. Then this direct transformation is mapped by the backwards retyping functor to the direct transformation $G \overset{p/m}{\Longrightarrow} H$ in $Trans(TGTS)$, where $H = f^<_{TG}(H')$, provided that the NACs N_i of the rule $\pi(p) = f^<_{TG}(\pi'(f_P(p)))$ are all satisfied for the NAC-morphisms $f^<_{TG}(n'_i)$ at match $m = f^<_{TG}(m')$:

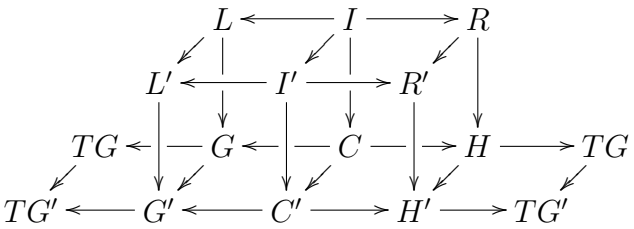
$$f^<_{TG}(G' \overset{\pi'(f_P(p))/m'}{\Longrightarrow} H') = (G \overset{\pi(p)/m}{\Longrightarrow} H)$$

△

Proof: Given rule $p' = f_P(p)$, we construct $f^<_{TG}(\pi'(f_P(p))) = \pi(p)$, and show that the backwards retyping functor $f^<_{TG}$ preserves pushouts since these are constructed separately for each type. Since the satisfaction of NACs is required, we do not consider the NACs in the proof. The direct transformation $G' \overset{f_P(p),m'}{\Longrightarrow} H'$ in $Trans(TGTS')$ is given by the left DPO diagram below. We have to show that the left diagram is mapped by $f^<_{TG}$ to the right DPO diagram, where the top row is the rule $f^<_{TG}(\pi'(f_P(p))) = \pi(p)$, and the match is $f^<_{TG}(m') = m$, and the graph the rule is applied to is $f^<_{TG}(G') = G$.

$$\begin{array}{ccccc} L' & \xleftarrow{l'} & I' & \xrightarrow{r'} & R' \\ {\scriptstyle m'}\downarrow & & \downarrow & & \downarrow \\ G' & \longleftarrow & C' & \longrightarrow & H' \end{array} \qquad \begin{array}{ccccc} f^<_{TG}(L') & \xleftarrow{f^<_{TG}(l')} & f^<_{TG}(I') & \xrightarrow{f^<_{TG}(r')} & f^<_{TG}(R') \\ {\scriptstyle f^<_{TG}(m')}\downarrow & & \downarrow & & \downarrow \\ f^<_{TG}(G') & \longleftarrow & f^<_{TG}(C') & \longrightarrow & f^<_{TG}(H') \end{array}$$

Starting with the left diagram above, in the first step we construct the graph G and the rule graphs L and I as pullbacks by restricting the graphs G' and the rule graphs L' and I' to the type graph TG. The match $L \xrightarrow{m} G$ and the morphism $L \leftarrow I$ are the respective unique pullback morphisms. By pullback decomposition property A.7, 4, we now have a pullback in the left face of the left cube, and a pullback in its upper face.



In the second step, we construct C as pullback object of the pullback $TG \times_C C' = TG'$. The morphisms $I \to C$ and $C \to G$ are the unique pullback morphisms. By pullback decomposition, we now have a pullback in the bottom of the left cube, and the left cube is commutative. As the top and the left squares are pullbacks, and the bottom is a pullback, we can use pullback decomposition property A.7, 4, and conclude that the right square of the left cube is a pullback as well. Considering the left cube, we now have pullbacks in the bottom, top, left and the right squares, and a pushout in the front square with injective $L' \leftarrow I'$. Using the Van-Kampen property A.14, we conclude that the back square of the left cube is a pushout.

In the third step we construct H as pullback object of the pullback $H' \times_H TG = TG'$. To show that we get the morphism $C \to H$ as unique pullback morphism, we have to show that $C \to G \to TG \to TG' = C \to C' \to H' \to TG'$. As the composition of the left bottom of the left cube with the square (1) is a pullback, we know that $C \to G \to TG \to TG' = C \to C' \to G' \to TG'$. Due the commutativity of the typing morphisms, we get that $C \to C' \to G' \to TG' = C \to C' \to H' \to TG'$, and hence,

$C \to G \to TG \to TG' = C \to C' \to H' \to TG'$ which we wanted to show. Thus, the morphism $C \to H$ is the unique pullback morphism of the pullback $H' \times_H TG = TG'$.

Analogously, we construct R as pullback object of the pullback $R' \times_R TG = TG'$ and get the morphisms $R \to H$ and $I \to R$ as unique pullback morphisms. By pullback decomposition, we conclude that the bottom square and the right square of the right cube are pullbacks as well, and the right cube is commutative. By pullback composition and decomposition, we conclude that the top square is a pullback as well. Considering the right cube, we now have pullbacks in the top, left, right and bottom squares, and a pushout in the front face with injective $I' \to R'$. Using the Van-Kampen property A.14, we conclude that the back square of the right cube is a pushout. □

2.1.3.1 Towards Retyping for Attributed Graph Transformation

Applying typed graph transformation to visual language definition (as discussed in Section 2.2), usually needs attributed nodes. Thus, we have to clarify how the concept of retyping can be extended to node attributes. If we use attributes only as labels, i.e. they are not changed during a transformation, this kind of typed attributed graphs can be defined by ordinary typed graphs. (Potentially infinite) sets of data values are considered as nodes. They are called data nodes in contrast to object nodes denoting all other nodes of an attributed graph. Data nodes and object nodes are linked by attribution edges, i.e. edges with an object node as source and a data node as target. We assume that there are no edges starting at some data node. If this property is satisfied within the type graph, it also holds for the instance graphs due to the typing morphisms.

In the case of attribute labels, it might be convenient to add variable nodes of data types to rule graphs which are matched by concrete labels when applying such a rule.

Summarising, graphs and graph transformation with node attributes which are not changed during transformation are already captured by our formalisation of typed graph transformation systems. If a more general attribution concept is needed where computations can take place on attributes, future work has to be done to extend the formal approach.

2.1.4 Amalgamated Graph Transformation

Concepts from amalgamated graph transformation will be used in Chapter 3 to express the behavior of Petri nets and Statecharts by rule schemes.

Parallel graph transformation in the double-pushout approach has been introduced in [Tae96]. For an extension of the concepts to attributed graphs and rules with attribute conditions, see [ETB05].

The main idea of parallel graph transformation is to apply a number of rules in one

parallel step. Their matches are allowed to overlap and can even be conflicting in the general case. Common subactions are described by subrules. Therefore, the notion of subrule embedding is basic to the whole approach.

To apply a set of rules in parallel in a synchronized way, we have to decide how and how often the rules can be applied to a host graph G. One possibility is to allow a rule to be applied at all different matches it has in G. This would result in a massively parallel application of rules which is not always wanted. To restrict the degree of parallelism, two control features are introduced: the *interaction scheme* and the *covering construction*. The interaction scheme is a set of subrule embeddings and restricts the synchronization possibilities of rule applications. The covering construction restricts the matching possibilities for the rules of the interaction scheme. One special covering construction, called *local*, allows to match a subrule s exactly once to a part $m(s)$ of G, and to match all rules extending s as often as possible to the surroundings of $m(s)$. In this way, a kernel action can be described in a variable context. Another important covering construction, called *fully synchronized* forbids conflicting rule matches, i.e. two rule matches of rules extending the same subrule s have to overlap completely at a match of their common subrule.

Formally, a covering is described by an instance interaction scheme and a set of matches. The instance interaction scheme contains the concrete number of instances of each rule in the scheme, depending on how many matches into G have been found for each rule of the interaction scheme. Thus, an interaction scheme can be seen as type information for instance interaction schemes.

Definition 2.1.28 (*Interaction Scheme*)
An *interaction scheme* IS consists of a set of subrule embeddings such that the following conditions hold:

- for each two subrule embeddings $t_1 : s_1 \to p_1$ and $t_2 : s_2 \to p_2$ we have $s_1 \neq s_2$ or $p_1 \neq p_2$.

IS is called *local interaction scheme*, if there is one subrule s being the source of at least one subrule embedding to each extending rule. △

Definition 2.1.29 (*Instance Interaction Scheme*)
Given an interaction scheme IS, an interaction scheme IIS is an *instance interaction scheme* of IS, if there is a mapping $ins : IIS \to IS$ such that $\forall t \in IIS$: if there is an isomorphic subrule embedding $t \xrightarrow{\sim} u$ then $ins(t) = u$. △

Definition 2.1.30 (*Covering Construction*)
Let IS be an interaction scheme and G a (typed) graph . A partial covering $COV = (IIS, MA)$ consists of an instance interaction scheme IIS of IS and a set MA of matches

from all rules of all subrule embeddings in IIS to G such that they commute with the subrule embeddings, i.e. for any two subrule embeddings $t_1 : s \to p_1$ and $t_2 : s \to p_2$ in IIS there are two matches $m_s : L_s \to G$ and $m_p : L_p \to G$ in MA with $m_p \circ f_L = m_s$. Let $t_1 : s \to p_1$ and $t_2 : s \to p_2$ be any two subrule embeddings in IIS and $m_{p_1} : L_{p_1} \to G$ and $m_{p_2} : L_{p_2} \to G$ corresponding matches in MA.

1. COV is called *local* , if IIS is local, and if p_1 is isomorphic to p_2, then m_{p_1} has to be non-isomorphic to m_{p_2}.
2. COV is called *fully synchronized*, if there are two subrule embeddings $u_1 : s' \to p_1$ and $u_2 : s' \to p_2$ such that $m_{p_1}(L_{p_1}) \cap m_{p_2}(L_{p_2}) = m_{s'}(L_{s'})$.

△

Since category $\mathbf{Graphs_{TG}}$ has an initial object (the empty graph), and pushouts (constructed componentwise), it is finitely cocomplete [Mac71], i.e. has all finite colimits. This is the basis to build the amalgamated rule of any partial covering which glues all parallel rules according to their subrule embeddings. Applying the amalgamated rule afterwards according to Def. 2.1.9 completes a parallel graph transformation step.

Definition 2.1.31 (*Amalgamated Rule and Transformation*)
Let G be a graph and $COV = (IIS, MA)$ be a covering construction with $IIS = \bigcup_{n\in N}(t_n : s_n \to p_n)$ being an instance interaction scheme with subrule embeddings $t_n = (f_{L_n}, f_{I_n}, f_{R_n})$ from the subrules $s_n = (L_{s_n} \overset{l_{sn}}{\leftarrow} I_{s_n} \overset{r_{sn}}{\rightarrow} R_{s_n})$ to the extending rules $p_n = (L_{p_n} \overset{l_{pn}}{\leftarrow} I_{p_n} \overset{r_{pn}}{\rightarrow} R_{p_n})$, and the set of matches $MA = \cup_{n\in N} m_n : L_n \to G$. The *amalgamated rule* $p_{COV} = (L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R)$ is constructed by the following steps:

1. Let L be the colimit object of $\bigcup_{n\in N} f_{L_n} : L_{s_n} \to L_{p_n}$ with $a_n : L_{p_n} \to L$.
2. Let I be the colimit object of $\bigcup_{n\in N} f_{I_n} : I_{s_n} \to I_{p_n}$ with $b_n : I_{p_n} \to I$.
3. Let R be the colimit object of $\bigcup_{n\in N} f_{R_n} : R_{s_n} \to R_{p_n}$ with $c_n : R_{p_n} \to R$.
4. Morphisms l and r are uniquely determined by the universal property of colimit (I, b_n) such that $a_n \circ l_{p_n} = l \circ b_n$ and $c_n \circ r_{p_n} = r \circ b_n$. Due to the colimit construction and the subrule embedding being injective, l and r are injective as well.

The amalgamated match $m_{COV} : L \to G$ is uniquely determined by the universal property of colimit (L, a_n), i.e. $m_{COV} \circ a_n = m_n$. An *amalgamated graph transformation* is a direct transformation $G \overset{p_{COV}, m_{COV}}{\Longrightarrow} H$ applying amalgamated rule p at the amalgamated match m_{COV}.

A parallel (typed) graph transformation system $PGTS = (S, IScheme)$ based on $\mathbf{Graphs_{TG}}$ consists of a (typed) graph S, called start graph, and a set $IScheme$ of (typed) interaction schemes.

△

Parallel (typed) graph transformations of a parallel (typed) graph transformation system are defined over their amalgamated rules and matches analogously to sequential (typed) graph transformations in Def. 2.1.9.

2.2 Definition of Visual Languages

In this section, the concepts for formal visual language definition based on typed algebraic graph transformation are reviewed (based on [Tae06]) and compared to related approaches. The *Applications* section contains visual language definitions (i.e. visual alphabets and syntax grammars) for sample VLs for different variants of Petri nets and Statecharts. The sample VLs defined here are the basis for behavior and animation specifications discussed in Chapters 3 and 4, and for their implementation in GENGED, a generator for simulation and animation environments based on VL definition by typed attributed graph transformation systems (see Chapter 5).

2.2.1 Meta-Modeling versus Graph Transformation

Two main approaches to visual language definition can be distinguished: the declarative way, called *meta-modeling* and the constructive way, based on *(graph) grammars*.

2.2.1.1 Metamodeling in the MOF Approach

UML is defined by the Meta Object Facilities (MOF) approach [MOF05] which uses classes and associations to define symbols and relations of a VL. Within the MOF approach, each UML metamodel is structured in four sections: (1) Class diagrams, (2) explanations of the class diagram features, (3) well-formedness rules formulated in OCL [OCL03], and (4) informal description of the semantics of the features as natural-language comments.

1. Class diagrams specify an abstract syntax of the various UML modeling concepts. As example, Fig. 2.3 shows an excerpt from the UML Metamodel Core Package. Rectangles represent classes, connecting lines symbolize associations, and arrows with open triangles show generalization hierarchies. This class diagram says, for example that an Attribute is a StructuralFeature, which itself is a Feature being a

ModelElement, and an Association is related to at least two AssociationEnds (role name connection with multiplicity 2..*) which in turn are related to a single Classifier (role name type with multiplicity 1).

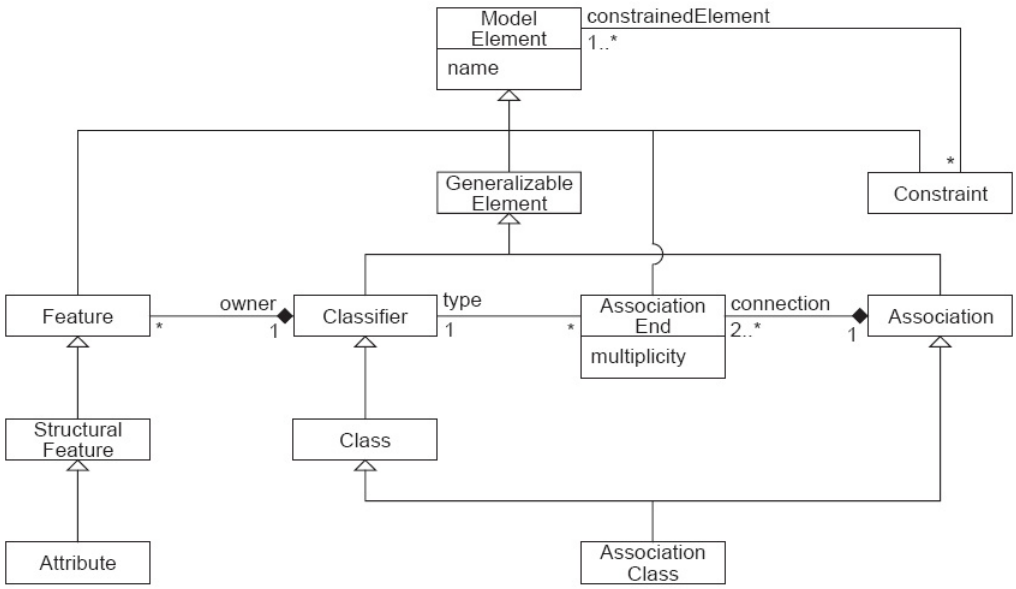


Figure 2.3: Part of the UML Metamodel Core Package

2. For the specified class diagrams also verbal descriptions of classes, attributes, and associations are given. The class Association in Fig. 2.3, for example, is described in the UML documentation [UML04b], p. 44, by the following text:

   > *An association describes a set of tuples whose values refers to typed instances. An instance of an association is called a link. [..] [An association] has at least two ends represented by properties, each of which is connected to the type of the end. More than one end of the association may have the same type. [..]*

3. Well-formedness rules given as OCL constraints then restrict the syntactical possibilities specified by the class diagrams. These constraints roughly correspond to context-sensitive conditions in textual language descirptions. For example, a condition like "*An association specializing another association has the same number of ends as the other association*" is formulated in [UML04b], p. 55 , by the following OCL constraint (within the context of the class Association):

   ```
   self.parents()->
   forAll(p | p.memberEnd.size() = self.memberEnd.size())
   ```

4. Finally, verbal descriptions of the semantics of the diagram features are given. The

concept of an association and its semantics is explained in the UML 2.0 metamodel [UML04b], p. 55, as follows (only a part of the description is cited):

> *An association declares that there can be links between instances of the associated types. A link is a tuple with one value for each end of the association, where each value is an instance of the type of the end. [..] The multiplicity of the association end constrains the size of this collection. If the end is marked as ordered, this collection will be ordered. If the end is marked as unique, this collection is a set; otherwise it allows duplicate elements. [..]*

The UML metamodel described above allows to represent a UML diagram as an object diagram being an instantiation of the UML metamodel.

2.2.1.2 VL Definition using Graph Grammars

While constraint-based formalisms such as MOF provide a declarative approach to VL definition, grammars are more constructive, i.e. closer to the implementation. In [MM98], for example, textual as well as graph grammar approaches are considered for VL definition. Textual grammar approaches (still a wide-spread way to define visual languages), such as picture-layout grammars [Gol91a, Gol91b] or constraint multiset grammars [Mar94], have to code multi-dimensional representations into one-dimensional strings. The use of graph grammars instead of textual grammars allows to represent multi-dimensional structures by graphs which is more natural. Moreover, the layout (or concrete syntax) of the language elements can be visualized directly. Analogously to the grammar-based definition of textual languages, graph grammars are used to define the structure of visual sentences as well as their construction.

Using graph transformation, a *type graph* defines the visual alphabet, i.e. the symbols and symbol relations of a visual language. Layout information is integrated in the type graph by special layout-related nodes or edges connected to symbol nodes, and by constraints on the relations of visual representations. The layout-related nodes or edges include information about the symbol's shape (any kind of graphical figure or line), and the constraints establish certain visual relations (like "The shape for this symbol type is always drawn inside the shape for another symbol type." or "The shape for this symbol type has always a minimal size of ...").

A type graph together with a *syntax graph grammar* can directly be used as high-level visual specification mechanism for VLs [BTMS99]. The syntax grammar restricts the allowed visual sentences over the alphabet to the meaningful ones. Syntax grammar rules define language generating syntax operations. A syntax operation is modeled as a typed graph rule (typed over the VL type graph) being applied to the concrete syntax graph of the current diagram. Thus, only syntactical changes are allowed which are described

by a syntax rule and which result again in a valid VL diagram. A syntax operation (i.e. the application of a syntax rule) results in a corresponding change of the internal abstract syntax graph of the diagram and its layout. The induced graph language determines the corresponding VL. Visual language parsers can be immediately deduced from a syntax graph grammar. Furthermore, abstract syntax graphs are also the starting point for model simulation, model transformation and model analysis by graph transformation [BP02, dLVA04, Var02, HKT02b].

There exist approaches to integrate meta-modeling and graph grammar techniques for defining visual languages and manipulating UML diagrams. In [BEdLT04], type graphs with inheritance are introduced leading to a more abstract and MOF-like form of type graphs. The typed graph transformation systems then may include abstract rules typed over abstract types. In [Gog00], graph transformations are used to formalize equivalence transformations on UML diagrams on the UML metamodel layer. In general, the MOF and the graph transformation approach can be integrated by identifying symbol classes with node types and associations with edge types. In this way, declarative as well as constructive elements may be used for language definition.

2.2.2 Definition of a Visual Alphabet

The underlying structure of a diagram is naturally described by an *abstract syntax graph (ASG)*. Also considering the conrete syntax, the kind of figures, lines, and their relations, can be represented by a graph, the so-called *spatial relations graph (SRG)*. The connection of both graphs covers all aspects of diagram representation.

The abstract syntax graph contains symbols and links. Symbols may be attributed by additional data. The concrete layout is described by visuals which may be any kind of figures and lines, and layout constraints to establish certain visual relations. Additional attributes are needed to define the properties of visual representations. This general approach is captured by so-called meta type graphs MT_A and MT_C where MT_A covers the abstract syntax and MT_C the concrete syntax.

Definition 2.2.1 (*Meta Type Graphs for VL Definition*)
The *meta type graphs* (MT_C, MT_A) for visual language definition consist of a meta type graph MT_C, modeling the definition of the concrete syntax of visual languages, and a meta type graph MT_A, which is included in MT_C: $MT_A \subseteq MT_C$, and which models how the abstract syntax of visual languages is defined. These meta type graphs are depicted in Fig. 2.4. All non-filled node types as well as their adjacent edge types belong to MT_A. The whole meta type graph depicts M_C. △

Symbols and links of a specific visual language as well as their specific visual representations are defined in type graphs T_A and T_C where T_A is included T_C. Both type graphs

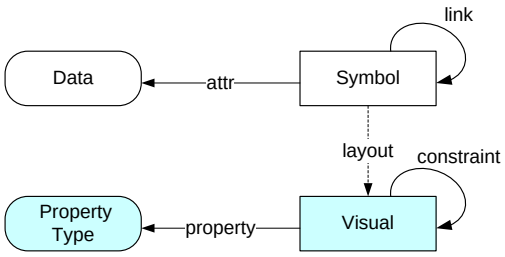


Figure 2.4: Meta Type Graphs for VL Definition

have to correspond to their meta type graph, i.e. TA is typed over MT_A and TC is typed over MT_C.

Definition 2.2.2 (*Visual Alphabet*)

A *visual alphabet* $Alph = (TG_C, TG_A)$ of a visual language consists of two type graphs $TG_C, TG_A \in \mathbf{Graphs_{TG}}$ where TG_C represents the concrete syntax of the visual language and is typed over MT_C, and TG_A represents the abstract syntax and is typed over MT_A.

According to the (meta-)typing, we have $TG_A \subseteq TG_C$. The restriction of TG_C to the meta types in MT_A leads to type graph T_A, i.e. the diagram to the right is a pullback in $\mathbf{Graphs_{TG}}$.

$$\begin{array}{ccc} TG_A & \hookrightarrow & TG_C \\ \downarrow & (PB) & \downarrow \\ MT_A & \hookrightarrow & MT_C \end{array}$$

△

Example 2.2.3 (*Visual Alphabet for Condition/Event Nets*)
As example we present the visual alphabet for Condition/Event nets (C/E nets for short), a Petri net variant which allows at most one black token at each place. Hence, no arc inscriptions are necessary as each transition firing step removes one token from each predomain place and puts one token to each postdomain place. Please note that our visual alphabet does not yet contain symbols for tokens, but models unmarked nets only. Token symbols are added later in Chapter 3, where an extended alphabet for simulation is defined.

Fig. 2.5 (a) shows the visual alphabet (TG_C, TG_A) for the C/E net language type over the meta type graph in Fig. 2.4. The alphabet contains four kinds of symbols, places, graphically drawn as ellipses, transitions which are visualized as rectangles, and arcs between places and transitions and vice versa, visualized by polylines. Additionally, places and transitions may be attributed by strings (their names). In Fig. 2.5, we use unfilled rectangles for the abstract syntax of symbols in TG_A (typed over Symbol in MT_A), and unfilled rounded rectangles for the abstract syntax of data attribute symbols (type Data in MT_A). Edges from unfilled rectangles to unfilled rounded rectangles show attributes. All unfilled rectangles and unfilled rounded rectangles, together with their adjacent edge types

comprise the abstract type graph TG_A of the C/E net alphabet. At the concrete syntax level, we use filled rectangles for visuals in TG_C (typed over Visual in MT_C), and filled rounded rectangles for their properties (type Property in MT_C). Edges from filled rectangles to filled rounded rectangles denote the obvious properties of figures, such as font, font size, color, fill color etc. Constraints between visuals are e.g. the inside-relation, meaning that the source figure is depicted completely inside the borders of the target figure, and the relations of poly lines to their respective source and target figures. Edges of type layout connect the abstract symbols with their visuals. The complete graph in Fig. 2.5 (a) is the concrete type graph TG_C of the C/E net alphabet.

In Fig. 2.5 (b), the type graph TG_C is depicted in a more compact form where the symbol attributes and the properties of the visuals are denoted inside the rectangles.

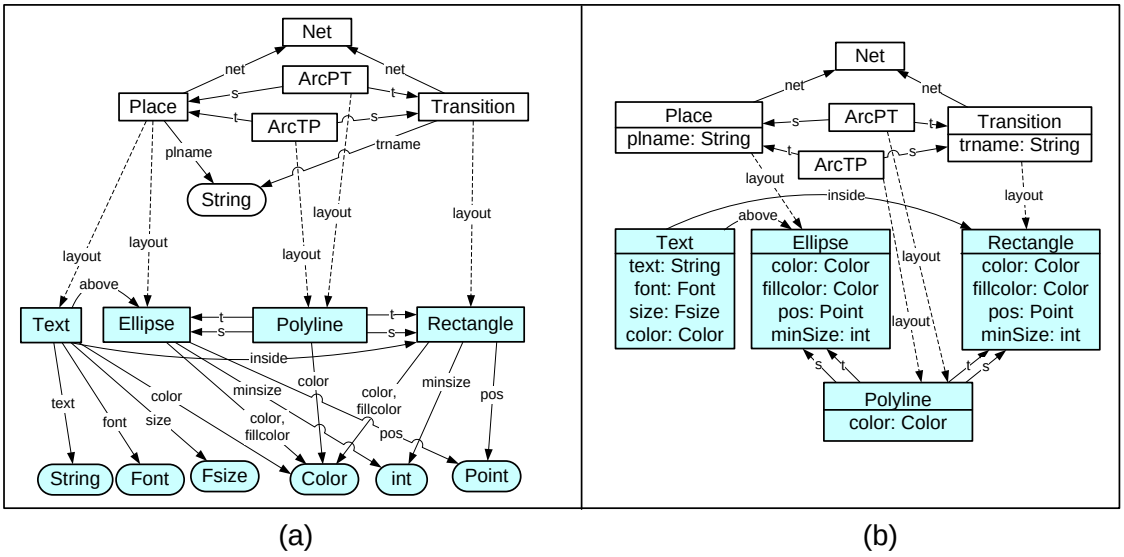


Figure 2.5: (a) Visual Alphabet of the C/E Net VL and (b) Compact Representation

△

2.2.3 Visual Models over a Visual Alphabet

A visual model over an alphabet (TG_C, TG_A) is given by a pair of graphs (M_C, M_A). The abstract syntax graph (ASG) M_A shows the abstract syntax structure of this model. It is typed over type graph T_A. Correspondingly, the concrete representation M_C of a visual model, i.e. the spatial relations graph (SRG) and its connection to the ASG, is typed over type graph TG_C.

Definition 2.2.4 (*Visual Model over Visual Alphabet*)
Let $Alph = (TG_C, TG_A)$ be a visual alphabet. A *visual model* over alphabet A is de-

fined by the concrete and abstract graphs (M_C, M_A) with $M_C \in \mathbf{Graphs_{TG_C}}$, and $M_A \in \mathbf{Graphs_{TG_A}}$.

According to the typing, we have $M_A \subseteq M_C$. The restriction of M_C to the types in TG_A yields the abstract syntax graph M_A, i.e. the diagram to the right is a pullback in $\mathbf{Graphs_{TG}}$.

$$\begin{array}{ccc} M_A & \hookrightarrow & M_C \\ \downarrow & (PB) & \downarrow \\ TG_A & \hookrightarrow & TG_C \end{array}$$

△

Example 2.2.5 (*Condition/Event Net as Visual Model over the C/E Net Alphabet*)

The concrete syntax of a C/E net is shown at the left side of Fig. 2.6 by the concrete and abstract graphs (M_C, M_A). The nodes belonging to the abstract syntax graph (ASG) M_A are drawn as unfilled rectangles. The typing of the ASG part M_A over TG_A and of the SRG part M_C over TG_C, where (TG_C, TG_A) are defined in Example 2.2.3, is shown explicitly in Fig. 2.6 by type names inside of all nodes and at all edges. At the right-hand side of Fig. 2.6, the C/E net is depicted in its concrete notation as defined by M_C.

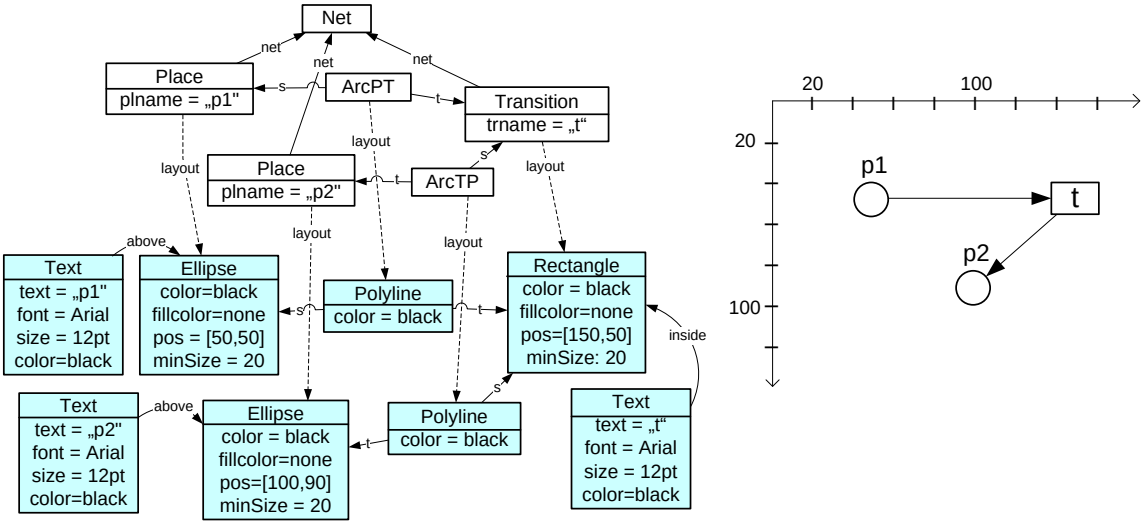


Figure 2.6: Concrete Syntax of a C/E Net

△

2.2.4 Visual Language over a Visual Alphabet

To define a VL, declarative as well as constructive approaches can be used. The Meta Object Facilities (MOF) approach which has been to define the Unified Modeling Language (UML), is a declarative one where classes of symbols and relations are defined and associated to each other. Constraints describe additional language properties. Defining a VL

by a graph grammar, the constructive way is followed where the application of graph rules builds up abstract syntax graphs of models.

2.2.4.1 Descriptive Approach

Using the descriptive approach for VL definition, the visual alphabet is defined, and a set of additional language constraints describing additional language properties. A diagram is a valid visual model if it is typed over the visual alphabet, and if all language constraints are fulfilled by the diagram.

Definition 2.2.6 (*Visual Language Constraints over Visual Alphabet*)
Let $Alph = (TG_C, TG_A)$ be a visual alphabet. A visual language constraint is a graph constraint (see Def. 2.1.6) $c = (c_C : X_C \to Y_C, c_A : X_A \to Y_A)$ where c_C is typed over TG_C, and c_A is typed over TG_A and defined by the restriction of c_C to TG_A. If the constraint c is called *positive* (pc), then a pair of injective morphisms $(X_C \xrightarrow{p_C} G_C, X_A \xrightarrow{p_A} G_A)$ satisfies c if $\exists q = (Y_C \xrightarrow{q_C} G_C, Y_A \xrightarrow{q_A} G_A)$ with q_A, q_C injective, and $q_C|_{TG_A} = q_A$, such that $q_C \circ c_C = p_C$ and $q_A \circ c_A = p_A$. If c is called *negative* (nc), then a pair of injective morphisms $(X_C \xrightarrow{p_C} G_C, X_A \xrightarrow{p_A} G_A)$ satisfies c if $\nexists q = (Y_C \xrightarrow{q_C} G_C, Y_A \xrightarrow{q_A} G_A)$ with with q_A, q_C injective, and $q_C|_{TG_A} = q_A$, such that $q_C \circ c_C = p_C$ and $q_A \circ c_A = p_A$. A visual model M over $Alph$ satisfies c if each $X \xrightarrow{p} M = (p_C, p_A)$ satisfies c. △

Definition 2.2.7 (*Visual Language over Visual Alphabet and Constraints*)
Let $Alph = (TG_C, TG_A)$ be a visual alphabet, and $Cons$ be a set of visual language constraints. The visual language over $Alph$ and $Cons$ is given by all models typed over $Alph$ that satisfy the constraints:

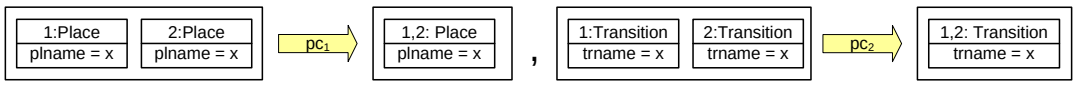
$$VL = \{M | M \in \mathbf{Graphs_{TG_C}}, M \models Cons\}$$

△

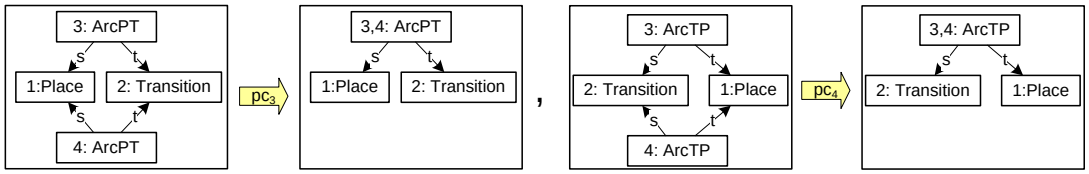
Example 2.2.8 (*Visual Language over C/E Net Alphabet and Constraints*)

In addition to the visual alphabet for C/E nets defined in Example 2.2.3, we state requirements for the C/E net language and express them by positive graph constraints $pc_1, .., pc_7$ at the abstract syntax level. Note that for the visual language of C/E nets, no constraints at the concrete syntax level are needed, since the necessary layout constraints are defined already by the alphabet (e.g. a place name is always written above the place figure, and a transition name is always written inside the transition figure). For more complex visual languages, it may be advisable to define additional graph constraints at the concrete syntax level.

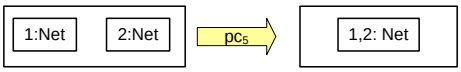
1. Place and transition names are unique.



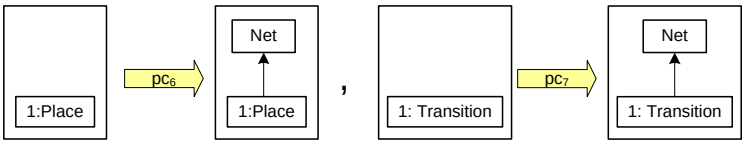
2. There exist at most one arc of type ArcPT and one arc of type ArcTP between a place and a transition.



3. There is exactly one Net node in the diagram.



4. All place and transition nodes are connected to the Net node.



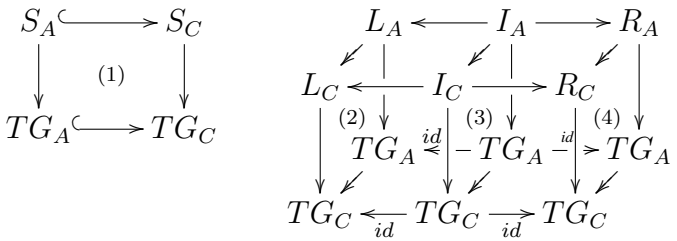
The visual language of C/E nets over the visual C/E net alphabet and the set of visual constraints for C/E nets $Cons_{CE} = \{pc_1, .., pc_7\}$is defined by the set of all visual models over the C/E net alphabet which satisfy the visual constraints in $Cons_{CE}$. △

2.2.4.2 Constructive Approach

Using the constructive approach for VL definition, also a visual alphabet (TG_C, TG_A) is defined. The valid visual models of a visual language then can be further restricted to the meaningful ones by a visual *syntax grammar* typed over the alphabet. Syntax rules using negative application conditions are a well-defined and constructive way to express which models belong to a VL.

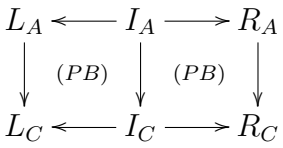
In general, a *VL syntax specification* is a pair of graph grammars (GG_C, GG_A) where $GG_C = (TG_C, S_C, P_C)$ is the VL syntax grammar and contains the complete syntax description of the VL.

Definition 2.2.9 (*VL Syntax Grammar*)
A *VL syntax grammar* $GG = (GG_C, GG_A)$ consists of a concrete syntax grammar and an abstract syntax grammar GG_A. The *concrete syntax grammar* $GG_C = (TG_C, S_C, P_C)$ contains the concrete type graph TG_C of the visual alphabet, a start graph S_C, and a set of syntax rules P_C. All graphs in GG_C, i.e. the start graph S_C and all rule graphs, are typed over TG_C. The *abstract syntax grammar* $GG_A = (TG_A, S_A, P_A)$, contains the abstract type graph TG_A of the visual alphabet, a start graph S_A and a set of abstract syntax rules P_A. The abstract start graph S_A is constructed by restricting S_C to the types in TG_A, i.e. square (1) in the diagram below is a pullback in $\mathbf{Graphs}$. Analogously, for each concrete rule $p_C = (L_C \leftarrow I_C \rightarrow R_C) \in P_C$, we define the abstract rule $p_A = (L_A \leftarrow I_A \rightarrow R_A) \in P_A$ by the pullbacks $(2), (3)$ and (4) in all diagonal squares in the diagram below.



△

Lemma 2.2.10 (*Abstract Rules are Subrules of Concrete Rules*)
Let $GG = (GG_C, GG_A)$ be a syntax grammar, $p_C \in P_C$ a concrete syntax rule typed over TG_C, and $p_A \in P_A$ the correspnding abstract rule typed over TG_A, constructed by the three pullbacks $(2), (3)$ and (4) given in Def. 2.2.9. Then p_A is a subrule of p_C, according to Def. 2.1.8, i.e. both squares in the diagram to the right are pullbacks in $\mathbf{Graphs}$.



△

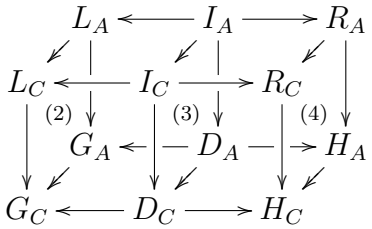
Proof: Let us consider again the double cube diagram of Def.2.2.9. We know that the diagonal squares $(2), (3)$ and (4) are pullbacks due to the construction of abstract syntax rules. As the bottom squares are also pullbacks due to Property A.6, 1, the composition of square (3) with the left bottom square yields again a pullback due to Property A.7, 3. Hence, the composition of the left top square with square (2) is a pullback as well. As (2) is a pullback, we can apply the pullback decomposition property A.7, 4, and obtain that the left top square is a pullback. By the same argumentation over the right cube, we conclude that the right top square is a pullback as well. □

All graphs which can be generated by the rules of the syntax grammar GG are elements of its visual language VL.

Definition 2.2.11 (*Visual Language over VL Syntax Grammar GG*)
The *visual language* $VL = (VL_C, VL_A)$ over a VL alphabet $Alph = (TG_C, TG_A)$ and a VL syntax grammar $GG = (GG_C, GG_A)$ consists of a *concrete visual language* VL_C, the elements of which are typed over TG_C and constructed by the rules of the concrete syntax grammar GG_C: $VL_C = \{G_C | S_C \overset{*}{\Longrightarrow} G_C\}$, and an *abstract visual language* VL_A, the elements of which are typed over TG_A and constructed by the rules of the abstract syntax grammar GG_A: $VL_A = \{G_A | S_A \overset{*}{\Longrightarrow} G_A\}$. △

Lemma 2.2.12 (*Abstract and Concrete Transformations*)
Let G_C be a concrete graph, and $p_C \in P_C$ be a concrete syntax rule, both typed over TG_C. Furthermore, let $G_C \overset{p_C,m_C}{\Longrightarrow} H_C$ be a direct concrete transformation using rule p_C at match m_C. Then, the corresponding direct abstract transformation $G_A \overset{p_A,m_A}{\Longrightarrow} H_A$ can be constructed as restriction of the direct concrete transformation $G_C \overset{p_C,m_C}{\Longrightarrow} H_C$ to the type graph TG_A. △

Proof: We use the Van Kampen Property A.14 for the left cube and the right cube in the diagram below to show that the left and right back squares are pushouts. We start with the left cube: The diagonal squares (2) and (3) are pullbacks by construction of the abstract syntax rule. The top square is pullback as p_A is subrule of p_C (see Lemma 2.2.10), and the bottom square is pullback due to Property A.6. The front square is a pushout due to the rule application of rule p_C to graph G_C at match m_C. Applying the Van Kampen Property, we conclude that the left back square is a pushout, as well. By the same argumentation for the right cube, we can show that the right back square is also a pushout. Hence, we have constructed a direct abstract transformation $G_A \overset{p_A,m_A}{\Longrightarrow} H_A$ as restriction of the direct concrete transformation $G_C \overset{p_C,m_C}{\Longrightarrow} H_C$ to the type graph TG_A.



□

Proposition 2.2.13 (*Abstract and Concrete Visual Languages*)
Let $VL = (VL_C, VL_A)$ be a *visual language* over VL alphabet $Alph = (TG_C, TG_A)$ and VL syntax grammar $GG = (GG_C, GG_A)$. Then, we have an inclusion of $VL_C|_{TG_A} \subseteq VL_A$. △

Proof: We have to show that for all elements $G_C \in VL_C$ the restriction to the abstract type graph is an element of the abstract visual language, i.e. $G_C|_{TG_A} \in VL_A$. For the

start graph S_C it holds by Def. 2.2.9 that $S_A = S_C|_{TG_A}$. For all other graphs $H_C \in VL_C$ with $H_C \neq S_C$ there exists a transformation $S_C \stackrel{*}{\Longrightarrow} H_C$. By Lemma 2.2.12, we know that for each direct concrete transformation the corresponding direct abstract transformation $G_A \stackrel{p_A,m_A}{\Longrightarrow} H_A$ can be constructed as restriction of the direct concrete transformation $G_C \stackrel{p_C,m_C}{\Longrightarrow} H_C$ to the type graph TG_A. Hence, for each $H_C \in VL_C$, constructed by a concrete transformation, the restriction $H_C|_{TG_A}$ is a valid element of the abstract visual language VL_A. □

Example 2.2.14 (*VL Syntax Grammar for C/E Nets*)
A visual syntax grammar for C/E nets has to ensure the following language requirements (which were stated already in Example 2.2.8, where they were formalized in the descriptive approach by graph constraints):

1. Place and transition names are unique.
2. There exist at most one arc of type ArcPT and one arc of type ArcTP between a place and a transition.
3. There is exactly one Net node in the diagram.
4. All place and transition nodes are connected to the Net node.

Fig. 2.7 shows the VL syntax rules for the VL of C/E nets, realising the insertion of places, transitions and arcs. (Tokens will be introduced later when extending the C/E net VL for simulation in Section 3.2.3.)

The start graph of the VL syntax grammar for C/E nets consists of a single Net node only. As none of the syntax rules adds Net nodes, condition 4 is fulfilled by the syntax grammar. The first two rules realize the insertion of places and transitions. Each newly inserted place or transition is connected to the Net node. Thus, condition 4 is fulfilled by the grammar. The name of a new place or a new transition is given by the respective input parameter PlName or TrName. Negative application conditions (NAC) of these first two rules forbid that the name chosen for a new place has been used already for a previously inserted place, and, analogously, that the name for a new transition has been used for an existing transition in the current net. In this way, condition 1 is fulfilled by the syntax grammar. The last two rules realize the insertion of arcs. Here, their NACs ensure that there is not already an arc between the same place and transition nodes in the same direction as the one to be inserted. Thus, condition 2 is satisfied as well.

Note that the positions of places and transitions which are inserted by the first two rules, are default values. In order to generate all possible layouts, the concrete grammar should comprise two more rules for moving places and transition to arbitrary positions, indicated

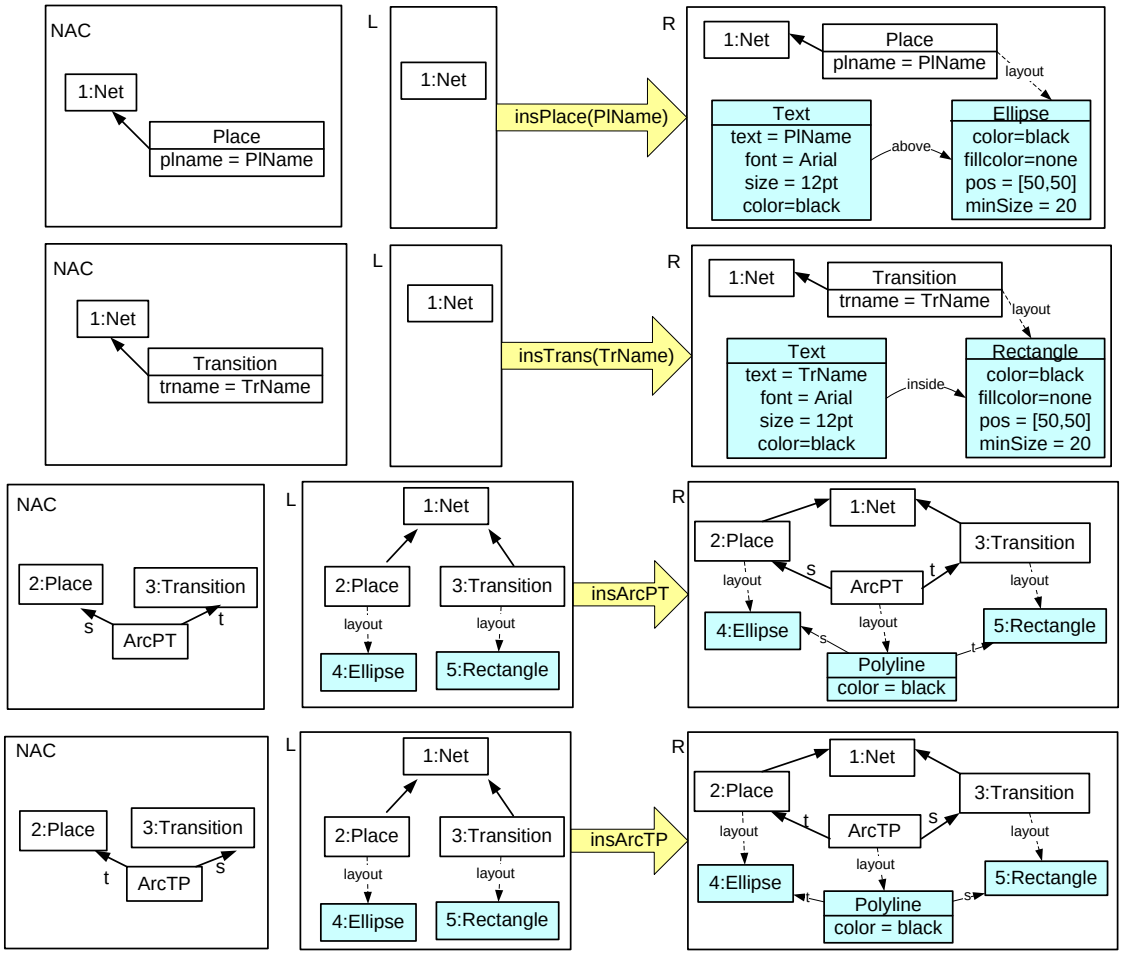


Figure 2.7: VL Syntax Rules for Condition/Event Nets

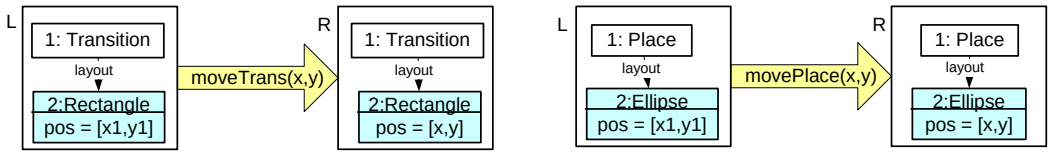


Figure 2.8: More VL Syntax Rules for Condition/Event Nets

by the rule parameters x and y, as shown in Fig. 2.8. The move rules are special in the sense that they are identical on the abstract syntax level.

The visual language of C/E nets over the C/E net alphabet (Fig. 2.5) and the C/E syntax grammar (Fig. 2.7 and Fig. 2.8) is now defined as the set of visual models which are derived by one or more syntax rule applications, starting at the start graph of the syntax grammar. One example for a visual model over the C/E net alphabet and the C/E net syntax grammar is the C/E net in Fig. 2.6, as it is not only a model over the alphabet, but conforms also to the additional requirements of the C/E net language, and therefore can be constructed by

applying the syntax rules. The net in Fig. 2.6 is derived by applying the following rules, where the match of rule insArcPT maps the place node to place p1, and the match of rule insArcTP maps the place node to place p2:

insPlace(”p1”); movePlace(50,50); insPlace(”p2”); movePlace(100,90); insTrans(”t”); moveTrans(150,50); insArcPT; ins ArcTP; △

While the restriction of VL_C to TG_A always yields valid abstract visual models (see Proposition 2.2.13), it is not true that there is a always a valid concrete visual model for every abstract visual model. For instance, imagine that we define a rectangle of a fixed size as layout figure for the abstract Net node in our C/E net alphabet, and let us have a layout constraint inside between the ellipse figure for places and the net rectangle. Moreover, we add a constraint that place figures must not overlap. As all places have a minimal size, there is a maximal number of places that can be drawn inside the net figure. But, according to the abstract syntax grammar, we could apply the abstract syntax rule insPlace infinitely often, thus generating abstract visual models for which no concrete visual models exist.

In tools like GENGED [Bar02] or TIGER [EEHT04], which generate visual editors from visual language specifications, only the abstract syntax rules are applied when editing visual models. Purely concrete rules like move-rule are incorporated directly as editor features. In the case that no valid concrete visual model can be computed, the restriction of which to TG_A corresponds to the derived abstract visual model, then the previous state before the rule application is reestablished in the editor, and the user is informed that the rule is not applicable.

Obviously, graph-like visual languages like Petri nets or activity diagrams can be defined easily in a way that there generally is at least one concrete representation for each abstract visual model in VL_A. The language designer should avoid to restrict the size of figures which contain other figures, and the layout constraints in the alphabet should be defined in a way that they do not contradict each other. For the C/E net VL as specified by the visual alphabet in Fig. 2.5 and the visual syntax rules in Fig. 2.7, all abstract models of the abstract VL can be extended to valid concrete models.

We will consider techniques to compute valid concrete models from abstract ones in Chapter 5, where we present details concerning the realization of layout constraints in the visual language environment GENGED.

In Chapters 3 and 4, we define the formal notions for simulation and animation at the abstract syntax level and require that the visual alphabets are defined in a way that for each abstract model of the visual language at least one concrete visual model exists and can be computed.

From VL Syntax Grammars to VL Editing Grammars

As mentioned above, from the definition of a visual language, a rule-based visual editor for models of the VL can be generated [Bar02, EEHT04]. The sequence of editing steps then corresponds to a sequence of graph transformation steps from start graph S to a model M. The application of a syntax rule corresponds to an insertion of a new symbol or link. In general, the VL syntax grammar is not convenient enough to serve for *syntax-directed editing*. Therefore, the syntax grammar should be extended such that insertion of larger parts or manipulations of already created diagrams becomes possible. More convenient insertion rules may be special insertion rules in the sense that they require a special context for insertion which allows to combine several insertion steps standardly performed in this order. Besides insertion and manipulation of symbol properties, deletion is a main editing operation. Roughly spoken, deletion rules can be seen as the inverse rules of insertion rules.

Adding rules to the VL syntax grammar, we arrive at an editing grammar GG_E. We have to show that the generated languages of both grammars GG and GG_E are the same, i.e. adding editing rules to a VL syntax grammar must not result in an extension of the defined visual language.

An alternative for syntax-directed editing based on graph transformation is *free-hand editing*. A free-hand editor offers more general symbol editing commands. (emulated by simple editing rules without NACs), and requires parsing of the edited diagram, internally realized by applying parsing rules, to ensure that the diagram is a valid model of the VL. In general, these *parsing rules* are the inverted rules of the visual syntax grammar. The application of the parsing rules tries to reduce the abstract syntax graph of the diagram edited so far to a stop graph (see e.g. [BE01] for a parse grammar for Statecharts). If the reduction to the stop graph is successful, the diagram is a valid visual model; otherwise an error message informs the user that the diagram is invalid. The advantage of the free-hand editing approach is that the editing of intermediate invalid diagrams is tolerated by the editor.

Definition 2.2.15 (*VL Editing Grammar*)
Let $GG = (GG_C, GG_A)$ be a VL syntax grammar, and VL the generated visual language over GG and the visual alphabet $Alph = (TG_C, TG_A)$. A *VL editing grammar* is a graph grammar $GG_E = (T_C, S_C, P_E)$ with S_C and P_E typed over T_C and $P_C \subseteq P_E$, where S_C is the start graph and P_C the set of syntax rules of GG_C, provided that the visual language VL_E over $Alph$ and GG_E is the same as the visual language VL over $Alph$ and GG. The extended rules in P_E are called *editing rules*. △

2.3 Applications

To illustrate the definition of visual languages by further examples, this section presents selected visual alphabets and visual syntax grammars for the Petri net and Statechart VLs, namely for algebraic high-level nets (Section 2.3.1.1), Statecharts with nested OR-states (Section 2.3.2.1), and Statecharts with AND-states (Section 2.3.2.1). The presented VL definitions serve as basis for the simulation and animation applications considered in Chapters 3 and 4. Further VL definitions based on typed graph transformation concepts can be found in various papers co-authored by the author of this thesis, e.g. for activity diagrams [EEHT05], Place/Transition nets [EBE03], Timed-Transition Petri nets [dLETE04], specification architectures [BEP02b, EBP01], and finite automata [BEE+02].

2.3.1 VL Definitions for Petri Nets

2.3.1.1 Algebraic High-Level Nets

A specific, well-defined variant of high-level Petri nets are Algebraic High-Level nets (AHL nets for short). An AHL net is a combination of a place/transition net [Rei85] and an algebraic datatype specification *SPEC* describing operations used as arc inscriptions. Tokens are elements of a corresponding *SPEC*-algebra [EMC+98, EM85].

Definition 2.3.1 (*Type Graph for the AHL Net Language*)

The abstract type graph TG_A for the visual AHL net language of unmarked AHL nets is shown in Fig. 2.9

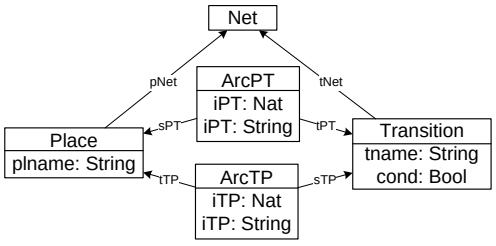


Figure 2.9: Abstract Syntax Type Graph TG_A for AHL Nets

Note that we allow polymorphic attributes (i.e. attributes of the same name but of different types), as in AHL nets tokens, and hence arc inscriptions may have different data types. We define the basic arc inscription types Nat and String, but an extension to more elaborate data types such as arrays or enumerations, is feasible by adding the respective attribute types to the arc inscription nodes. △

The concrete type graph for the VL of AHL nets look very similar to the concrete type graph for C/E nets (see Fig. 2.5). Firing conditions for transitions are written inside the transition rectangle, below the transition name. Moreover, the syntax rules for the VL of AHL nets are also similar to the syntax rules for C/E nets (see Fig. 2.7) and are not repeated here. For a complete VL definition for AHL nets, see [BEE01].

2.3.2 VL Definitions for Statecharts

The Statechart formalism proposed by Harel [Har87], mainly aims at modeling reactive systems as systems which are driven by events in a never-ending manner of reactions to external and internal stimuli. Modeling reactive systems with Statecharts means essentially to construct finite automata which are adapted to the necessities of the specification of reactive behavior. The states allow different reactions to equal stimuli, transitions change the states. The Statechart formalism extends this basic feature to additional concepts of hierarchy (*OR-states*) and parallelism (*AND-states*).

2.3.2.1 Statecharts with Nested OR-States

The abstract and concrete syntax of the VL for Statecharts with OR-states is given by the type graph $TG_{SC} = (TG_{SC_C}, TG_{SC_A})$ shown in Fig. 2.10. The abstract type graph TG_{SC_A} contains symbols for states and transitions Each state is attributed by its name and a boolean flag indicating whether it is an initial state or not, and each transition is attributed by the name of the event which triggers the transition. An edge of type super between two states allows to keep track of the nesting hierarchy (the source state of a super edge is substate of the target state).

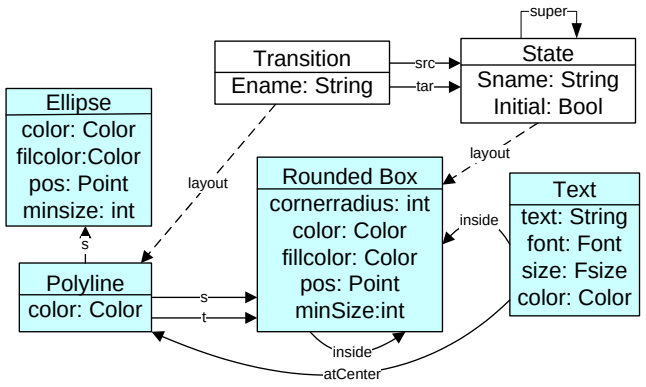


Figure 2.10: Type Graph for Statecharts with OR-States

The layout is given by TG_{SC_C} in Fig. 2.10. A state is visualized by a rounded box, where one property attribute defines the corner radius. The state name is displayed inside the box.

States are drawn inside their super states. A transition is shown by an arc connecting two states, and the name of the triggering event is displayed near the center of the arc.

An example for a small Statechart $M = (M_C, M_A)$ is shown in Fig. 2.11, where the abstract visual model M_A consists of the unfilled nodes and adjacent edges at the upper half of the figure (in fact, this is a part of the bigger radio clock Statechart serving as example for simulation in Section 3.1.2). At the right-hand side of Fig. 2.11, the Statechart is depicted using the shapes and properties defined by the concrete visual model M_C.

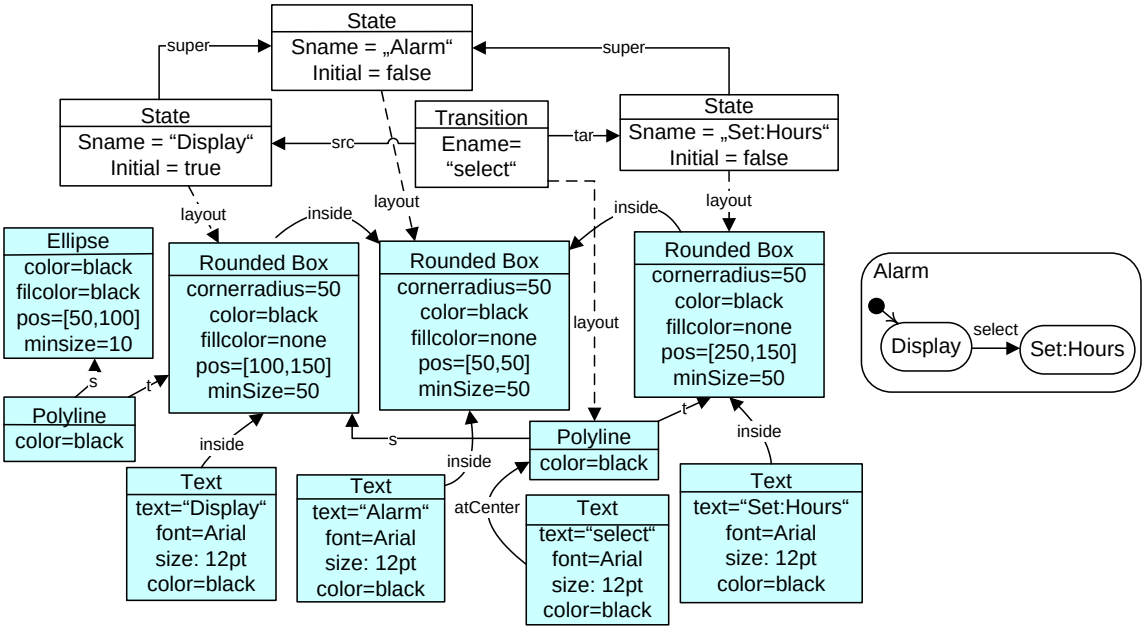


Figure 2.11: Statechart with OR-State as Visual Model

The design of the visual syntax grammar defining the visual language of Statecharts with OR-states has to ensure that the abstract syntax of all visual models of the language satisfies the following requirements

1. A transition always combines two states within the same hierarchy level;
2. There is at most one transition between two states s_1 and s_2 which is triggered by the same event ;
3. There is exactly one initial state in each hierarchy level;
4. There are no cycles in the hierarchy structure (i.e. a superstate of state s can never be a substate of state s or of any of its substates).

We give the abstract syntax grammar only, as we made sure by the definition of the visual language that a valid concrete visual model can be computed for each valid abstract visual

model. Especially, in the alphabet we define no upper bounds on the sizes of objects, and in the syntax grammar we ensure that there are no hierarchy cycles, such that we never have the situation that a symbol shape has to be drawn inside another symbol shape which in turn should be drawn inside the first one. All other layout constraints can be fulfilled by enlarging the corresponding shapes or by placing them further apart.

Fig. 2.12 shows the abstract syntax grammar for the visual language of Statecharts with OR-states. The start graph (not depicted in Fig. 2.12) contains only a start state named Start, which is at the outermost hierarchy level (i.e. it has no super state). All states to be added will be nested in this start state . In order to rename the start state (or any other state of the model), rule renameState(n) has to be applied with input parameter n being the new state name. Note that names for states and transitions need not be unique in this visual language specification. Rule insInit(n) inserts an initial state as substate of another state. As the NAC forbids the application of this rule if there is already an initial state at this hierarchy level, this rule fulfills a part of language requirement 3, namely that there is *at most* one initial state at each hierarchy level. The next rule, insSub(n), allows the insertion of a simple state as substate of another state, only if there is already an initial state at this level. Thus, rule insSub(n) fulfills the other part of language requirement 3, namely that there is *at least* one initial state at this level before adding more states. The fact that rules insSub(n) and insInit(n) allow to add states as substates of other states only, leads to the fulfillment of requirement 4, as thus the hierarchy structure is built as a tree and can never contain cycles. Rule insTrans(e) models the insertion of a transition between two states, its NAC forbidding the existence of another transition between them which is triggered by the same event (requirement 2). Moreover, a transition can be inserted only between two states which are substates of the same superstate (requirement 1).

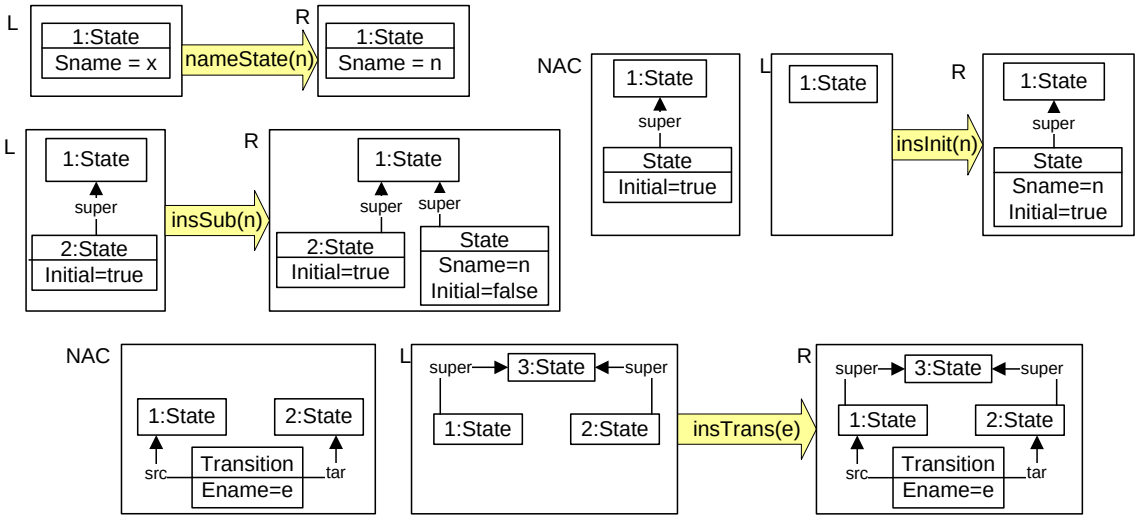


Figure 2.12: Abstract Syntax Grammar for Statecharts with OR-States

The sample Statechart in Fig. 2.11 is derived by applying the following syntax rules:
renameState(”Alarm”); insInit(”Display”); insSub(”Set:Hours”); insTrans(”select”);

In addition to the abstract syntax rules in Fig. 2.12, the syntax grammar contains concrete move rules for states, similar to the move rules for C/E nets in Example 2.2.14.

2.3.2.2 Statecharts with AND-States

The abstract and concrete syntax of the VL for Statecharts with AND-states is given by the type graph $TG_{SC} = (TG_{SC_C}, TG_{SC_A})$ shown in Fig. 3.21. At the abstract syntax level, we attribute states by their names and by boolean flags denoting their properties, i.e. a state is either an AND state or not (then it is a simple state), and its role can be initial Initial = true or not (Initial = false). A final state is modeled by an extra symbol type Final, because final states do not have components, they have no names, they cannot be initial and have no outgoing transitions. But they may have incoming transitions and they may be components of AND-states. An AND-state is linked to the component states of each of its parallel Statecharts. This is necessary to keep track of the hierarchy levels within AND-states. Each of the links to the initial states also holds the name of the parallel component. States are linked by transitions which are triggered by events.

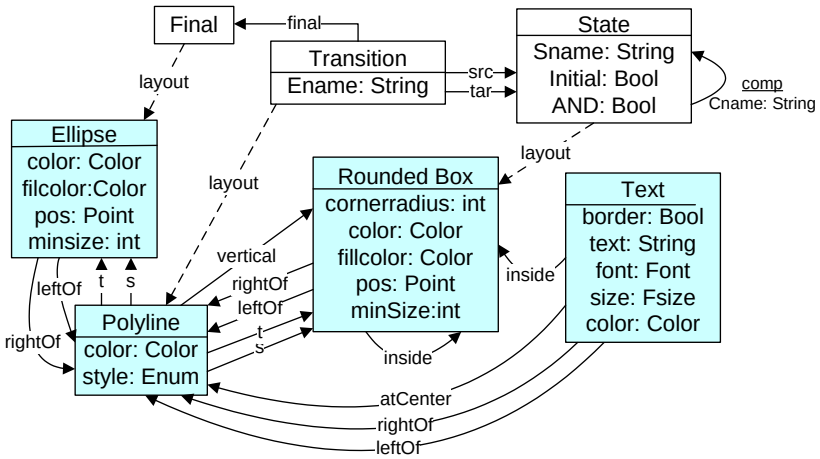


Figure 2.13: Type Graph for Statecharts with AND-States

The layout is given by TG_{SC_C} in Fig. 2.13. Like in Statecharts with OR-states, a state is visualized by a rounded box with the state name displayed inside, and a transition is shown by an arc connecting two states, where the name of the triggering event is displayed near the center of the arc. Components are drawn side by side inside their AND-states, where the Statecharts belonging to different components are separated by vertical, dashed lines.

An example for a small Statechart with an AND-state, $M = (M_C, M_A)$ is shown in Fig. 2.14, where the abstract visual model M_A consists of the unfilled nodes and adjacent

edges in the upper half of the figure (in fact, this is a part of the bigger ATM Statechart serving as example for simulation in Section 3.3.2). At the right-hand side of Fig. 2.14, the Statechart is depicted using the shapes and properties defined by the concrete visual model M_C (the complete syntax graph in Fig. 2.14).

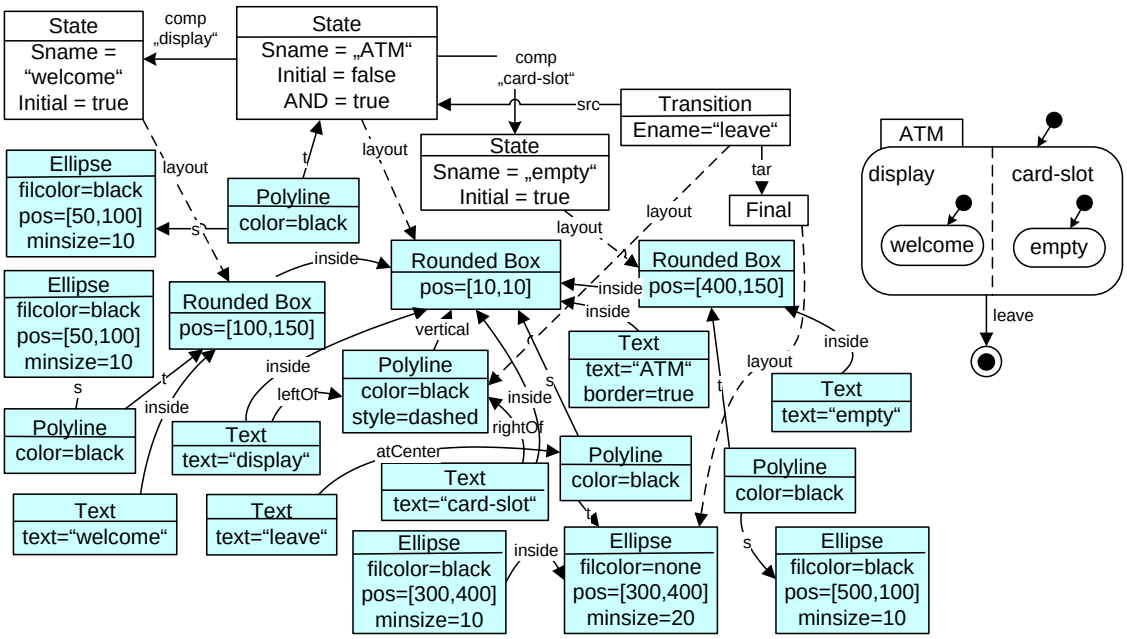


Figure 2.14: Statechart with AND-State as Visual Model

The design of the visual syntax grammar defining Statecharts with AND-states has to ensure (among others) the following requirements at the abstract syntax level:

1. An AND-state has at least two parallel components;
2. There is exactly one initial state in each component and one initial state outside of all components;
3. There are no cycles in the component structure (e.g. an AND-state s can never be a component of one of its own components).
4. A transition combines two states that belong either to the same component or to no component at all;
5. Final states do not occur isolated.

We show the abstract syntax grammar only, because we can argue analogously to the VL for Statecharts with OR-states (see Section 2.3.2.1) that a valid concrete visual model exists for each valid abstract visual model.

Fig. 2.15 shows the abstract syntax grammar for the visual language of Statecharts with AND-states. The start graph (not depicted in Fig. 2.15) contains only a start state named Start, which is an initial state but no AND-state. In order to rename the start state (or any other state of the model), rule renameState(n) is defined as in the syntax grammar for the OR-Statecharts VL in Fig. 2.12 (not depicted in Fig. 2.15). Rule newAND(l,r,initL,initR) transforms a simple state into an AND-state with two components, each containing one initial state. The input parameter defines the names for the left and right components (l and r) and for the respective initial states (initL and initR). This fulfills requirement 1. Additional components can be added to an AND-state using rule insComp(c,initC) which also generates the initial state of the new component. Rule insState(n) inserts a state outside of any AND-components. As the new state is not initial, there will always be only one initial state outside of AND-components (i.e. the start state). As rule insStateAND(n) also inserts only non-initial states within AND-components, requirement 2 is satisfied. Using rules insState, insStateAND and newAND to add states, leads to the fulfillment of requirement 3, as thus the component structure is built as a tree and can never contain cycles. Rule insTransAND(e) models the insertion of a transition between two states within the same component, and rule insTrans(e) adds a transition between two states outside of all AND-components. Thus, requirement 4 is satisfied. Final states are added only together with a transition the source of which is an existing state. This fulfills requirement 5.

The sample Statechart in Fig. 2.14 is derived by applying the following syntax rules, where the state in the rule insTransFinal is mapped to the state named ATM:

renameState(”ATM”); newAND(”display”,”card-slot”,”welcome”,”empty”); insTransFinal(”leave”);

At the concrete level, the rules in Fig. 2.15 have to be extended in order to insert the appropriate shapes like e.g. a dashed vertical line for each newly inserted component, and layout constraints, e.g. all states belonging to a component must be placed right to the dashed line separating the component from its left neighbour.

2.4 Related Work

Grammar-like formalisms for the definition of visual languages range from early approaches like array and web grammars [Ros76], and shape grammars [Gip75] to recent formalisms like picture layout grammars [Gol91b, Gol91a], positional grammars [CC91, CDLOT98], relational grammars [WW96, Wit91], constraint multiset grammars [Mar94], and several types of graph grammars [Roz97]. Non grammar-like formalisms include algebraic approaches [DÜ95] and logic-based approaches [Mey97, HM90, Wan94]. Most of these formalisms are described in [MM98], the first book about Visual Language Theory. Note that these formalisms are mainly motivated by applications involving parsing. Furthermore, most of the formalisms are rather concerned with visual programming languages than with

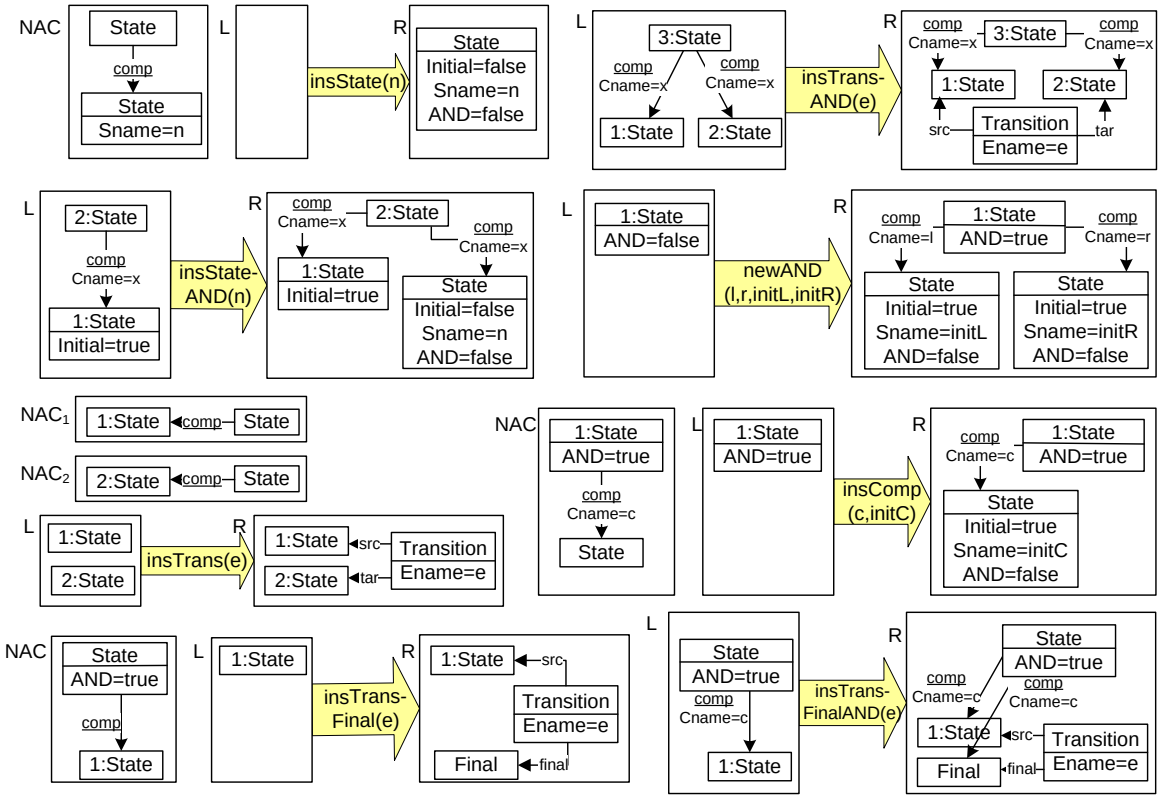


Figure 2.15: Abstract Syntax Grammar for Statecharts with AND-States

visual modeling languages. A comprehensive overview of visual programming languages, environments and parsing approaches is given in [BTMS99].

Graph grammars are used as a visual syntax definition formalism in graph-transformation based (meta) tools like PROGRES [SWZ99] and DIAGEN [Min02] which generate VL programming environments (with a main emphasis on VL editors and VL parsers) from visual syntax definitions. Graph-transformation based languages like PROGRES are a special brand of (visual) rule-based programming languages, which have a precisely defined syntax and semantics, and which offer thereby appropriate means for checking the correctness of constructed programs.

Both graph grammars and relational grammars [WW96] enforce a strict distinction between objects (nodes) and relations between objects (edges). Both approaches are, therefore, more or less equivalent to each other, especially in the case of hypergraph grammars, which use hyperedges as n-ary relationship objects. The differences between graph grammars on one hand and picture-layout grammars [Gol91b, Gol91a] or constraint multiset grammars [Mar94] on the other hand are more distinct. The latter two approaches enforce a language designer to represent all needed spatial or abstract relationships textually as constraints over attribute values of objects. Graph grammars use explicitly manipulated edges for any kind of relationships between objects and hence allow the visual definition

of all aspects of VL syntax, which is much more natural [BTMS99].

In recent years, a main direction of research has been to combine graph grammars as formal basis for VL definition and the MOF approach using UML diagrams for a semi-formal definition of the abstract syntax. For example, typed graph grammars have been extended by type graphs with inheritance and multiplicity constraints [BEdLT04]. Moreover, VL definitions based on typed graph grammars have proven a good basis for the formal definition of model transformations [EEEP06, TEG+05, EW05, EE05a, AK02], which is nowadays the key technology in the growing area of model driven development.

Chapter 3

Simulation of Visual Languages and Models

In Section 2.2, we considered the definition of visual languages in general. The techniques and concepts introduced are applied for the definition of all kinds of visual modeling languages, independent of the kind of system aspects shown in the corresponding models.

From now on we will concentrate on visual modeling languages for behavior modeling, which allow to describe how system components interact, and characterize the response to external system operations. Well-known modeling languages for behavioral aspects are Petri nets, Statecharts, automata, activity diagrams, sequence diagrams, collaboration diagrams etc. Models of such a visual behavior modeling language, called *visual behavior models*, represent system states over time during system runs, and are the basis for the *simulation* of system behavior.

Simulation is usually defined as "the process of designing a model of a real system and conducting experiments with this model for the purpose of understanding the behavior of the system and/or evaluating various strategies for the operation of the system" [PSS90], or, as defined more technically by the ISO/IEC standard on fundamental terms in information technology, simulation is "the representation of selected behavioural characteristics of one physical or abstract system by another system" [ISO93].

For visual behavior modeling languages such as Statecharts or Petri nets, usually simulation means to explore step by step (hand-triggered or automatically) the state space of a model. Simulation should be based on a step semantics of the modeling language, which is either given formally (as the definition of the firing behavior for Petri net transitions [Rei85] or the different approaches to define formal semantics for UML state machines / Statecharts [UML05]), or informally (e.g. in natural language, as for many domain-specific languages). Such a step semantics is defined for graph transformation systems in a formal way as the set of all transformations of a model, where the possible transformations are

specified by the graph transformation rules. We call such a graph transformation system for simulation a *simulation specification*. In order to allow automatic simulation, rules in a simulation specification (called simulation rules) may be structured to control their application order. This can be done either by using transformation units [KK99], simulation expressions [BEW02b], rule layers [BEW02a] or rule priorities [dL03]. We explain rule control structures along their usage in different examples.

Simulation specifications may be *universal* for the complete VL, i.e. the simulation rules can be used to simulate all possible models of the VL, or they may be *model-specific*, i.e. the simulation rules are designed for the simulation of one specific model and contain the model's features such as the names of places of a Petri net or states of a Statechart. Universal simulation specifications do not contain information about the models they are applied to, except that they all comply to a specific VL. As simulation specifications are the central concept to build animation specifications upon, we are more interested in model-specific approaches here, as domain-specific animations can only be defined if it is known (up to certain parameters) which state change in the model is realized by which simulation rule.

In this chapter we compare different approaches to define simulation specifications for different visual behavior modeling languages and visual behavior models, respectively. We discuss how the different notions for behavioral semantics are related. The basis for the sample simulation specifications are the visual language definitions given in Chapter 2 for Petri net and Statechart variants.

We distinguish between three kinds of simulation specifications which define behavior by typed graph transformation systems.

1. A *simulation specification for a visual language* VL (see Section 3.1) consists of universal simulation rules, i.e. the rules can be applied directly to the abstract syntax of all diagrams of the visual language VL. This defines a universal VL semantics, called *interpreter semantics*, a set of transformations for all possible VL diagrams. As example we present the interpreter semantics for Statecharts with nested OR-states.

2. A *simulation specification for a visual model* $G \in VL$ (see Section 3.2), consists of model-specific simulation rules which can be applied only to diagrams specific to G, i.e. diagrams corresponding to different markings of one Petri net or to different states of one specific Statechart. Given a visual behavior model G, it has to be translated ("compiled") into a simulation specification. By the notion *compiler semantics*, usually a more general mapping into an arbitrary semantic domain is meant. As in our case the semantic domain is always given by graph transformation systems (GTSs), we call the semantics of G, *GTS-compiler semantics*. The GTS-compiler semantics is defined by a model transformation translating a specific model G typed

over TG_1 into a simulation specification (a graph transformation system) typed over TG_2. The set of transformations defined by the simulation specification contains all possible transformations for one specific behavior model. An example is the model transformation from a specific Condition/Event Petri net into a corresponding simulation specification.

3. An *amalgamated simulation specification for a visual model* (see Section 3.3) combines the other two approaches by defining a general set of universal rule schemes which must be instantiated and amalgamated along a concrete behavior model resulting in a subset of all possible universal simulation rules. The advantage of this kind of simulation specification is that it can be defined using a finite number of rule schemes, even for a visual language VL, for which the simulation specification according to the first approach would have to contain an infinite number of universal simulation rules. For a concrete model G, the rule schemes are instantiated at all possible matches in G, resulting in a set of amalgamated rules. The set of transformations defined by these amalgamated rules is called *amalgamation semantics of* G. Examples for visual behavior modeling languages for which no finite simulation specifications can be given without using amalgamated rules, are all kinds of Petri nets and Statecharts with AND-states, which are discussed in detail as the illustrating example.

In Section 3.4 we discuss how the translation of simulation specifications with universal simulation rules or rule schemes, into model-specific simulation specifications. We show the semantical compatibility of the translation.

In the *Applications* section, we define the GTS-compiler semantics for Algebraic High-Level nets (AHL nets) [BEP02a, BEE01] and show that it is compatible to the original AHL net semantics, given in terms of firing sequences (Section 3.5.1). Furthermore, we define the amalgamation semantics for AHL nets, and show that the GTS-compiler semantics and the amalgamation semantics are semantically equivalent (Section 3.5.2). As a third application, a GTS-compiler semantics for the combined VL integrating selected UML diagram types (use cases, class diagrams, collaboration diagrams, Statecharts and object diagrams) is discussed [KGKK02] (Section 3.5.3).

Please note that in this chapter we consider only the abstract syntax of simulation rules. We assume that the underlying alphabet is defined in a way that there exists at least one concrete model for each derived abstract model.

All model-specific simulation specifications introduced in this chapter serve as bases for animation specifications introduced in Chapter 4.

3.1 Simulation Specifications for Visual Languages

Usually, a precondition for simulation of visual behavior modeling languages is a (slight) extension of the VL alphabet such that different execution states can be visualized. For example, state-oriented behavior descriptions such as automata, Statecharts and activity diagrams need a marker for the current state(s). Moreover, an event list may play a role for simulation. For Petri nets, tokens are used to mark the current system state. Having extended the VL alphabet, simulation rules operate directly on the (extended) abstract syntax of models.

3.1.1 Definition of Interpreter Semantics for Visual Languages

Definition 3.1.1 (*Simulation Alphabet*)
Let TG_A be the abstract alphabet of a visual behavior modeling language VL. We call TG_{sim} an abstract simulation alphabet extending TG_A, iff there is a type graph inclusion morphism $t : TG_A \rightarrow TG_{sim}$. △

In general, we assume that the visual behavior models to be simulated are syntactically correct, i.e. belong to the visual language VL. This approach assumes that the set of simulation rules typed over TG_{sim} can be splitted into two subsets (rule layers): the first subset of the simulation rules enriches the abstract model of the given visual language VL by special simulation information, e.g. a current-state marker or tokens, and thus allows to define the model's start state for the simulation run; the second subset of rules performs the simulation, e.g. by redistributing tokens or moving the current-state marker;

Definition 3.1.2 (*Simulation Specification for a Visual Language*)
Let TG_A be the abstract alphabet of the visual behavior modeling language VL, and TG_{sim} be the abstract simulation alphabet extending TG_A. A *simulation specification for* VL is given by a typed graph transformation system $SimSpec(VL) = (TG_{sim}, P_{sim})$. The rules in P_{sim} (called *universal simulation rules*) are typed over the type graph TG_{sim}. △

In general, universal simulation rules are applicable to all kinds of visual behavior models in VL. Thus, a *simulation* (a set of transformations using the simulation rules) contains transformations for all possible models, i.e. for the complete VL.

Definition 3.1.3 (*Simulation of a Visual Language*)
Let $SimSpec(VL) = (TG_{sim}, P_{sim})$ be a simulation specification for VL. The *simulation* $Sim(VL)$ (also called *interpreter semantics*) of the visual language VL consists of all graph transformations applying rules from P_{sim}:
$Sim(VL) = \{G_1 \overset{P_{sim}*}{\Longrightarrow} G_2 \mid G_1, G_2 \in VL,\ and\ G_1|_{TG_A} = G_2|_{TG_A}\}$

Each transformation of $Sim(VL)$ shows one specific simulation run, called *simulation scenario*. △

In order to define a semantics for a specific visual behavior model (e.g. a concrete Statechart), we have to restrict the transformations in $Sim(VL)$ to the ones which correspond to simulation rule applications to diagrams corresponding to states of this particular model. The diagrams representing model states differ from the model itself only in the extended symbols like the current-state marker or the current event queue.

Definition 3.1.4 (*Interpreter Semantics of Model G*)
Let G be a visual behavior model in VL. Let $SimSpec(VL) = (TG_{sim}, P_{sim})$ be the simulation specification for VL. Then *the interpreter semantics of* G, called $ISem(G)$, is the subset of transformations from $Sim(VL)$ where each diagram restricted to TG_A is isomorphic to G:
$ISem(G) = \{G_1 \overset{P_{sim}*}{\Longrightarrow} G_2 \mid G_1, G_2 \in VL,\ and\ G_1|_{TG_A} = G_2|_{TG_A} = G\}$

△

3.1.2 Interpreter Semantics for Statecharts with Nested OR-States

The main advantage of Statecharts is the common intuitive usage of the structuring concepts. The main disadvantage is caused by the different possibilities to interpret these structured models. Thus, numerous formal semantics are proposed in the literature, several of them based on graph transformation systems [BEdLT04, Var02, BEW02b, Kus01, GPP98]. In [BEdLT04], type graphs with inheritance are used to define the visual Statechart language and their simulation rules. Varró [Var02] uses model transition systems (a combination of graph transformation and metamodeling with explicit control structures). In [BEW02b], simulation expressions are used which allow the definition of parameterized and undividable simulation steps, each containing rule applications in a controlled, ordered way. In [Kus01, GPP98], the semantics (in [GPP98] this means the construction of a graph modeling a given Statechart in a flattened normal form without hierarchy) are defined by model transformation, i.e. the elements of a specific Statechart model are translated into a set of model-specific graph transformation rules. Kuske [Kus01] uses structured graph transformation (transformation units) to define rule application control in her approach.

In this section we give a purely graph-transformation-based semantics for the behavior of a simple Statechart variant with nested OR-states. (The corresponding VL for this Statechart variant is defined in Section 2.3.2.1). For rule application control we use a simple priority concept. We define priorities for rules which have to be taken into account after each rule application. If more than one rule are applicable at a time, then a rule of the layer with the highest priority must be chosen next. In our simple Statechart variant presented in this section, several concepts of the UML standard (such as transition guards, history

states, final states, deferred events, AND-states) have been omitted for the sake of clarity. An extension of the semantics to Statecharts with AND-states is presented in Section 3.3.2, given by a simulation specification with rule schemes.

The abstract syntax of the VL for Statechart simulation is given by the type graph TG_{SCsim} shown in Fig. 3.1. The basic Statechart VL (defined already in Section 2.3.2.1, consisting of symbols for states and transitions) is extended now by simulation symbols and links. The filled nodes Object (the object which points to the currently active state) and Event (the events in the queue which have to processed by the simulation run) are not included in the original Statechart VL but added for simulation purposes to be able to represent a Statechart configuration. The events in the event queue are linked by next edges. The last event is a special one (labelled as "none"), depicting the queue's end. A valid configuration requires exactly one basic state to be active. Then this basic state together with all its super states comprise a configuration.

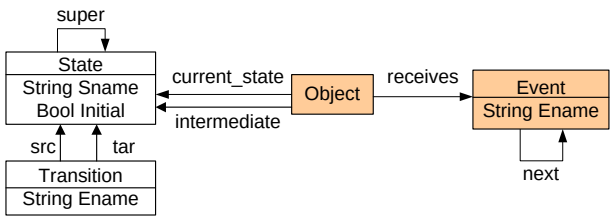


Figure 3.1: Type Graph for the Simulation of Statecharts with OR-States

Example: A Radio Clock Model

Fig. 3.2 shows a Statechart with nested OR-states modeling the behavior of a radio clock, which can show alternatively the time or the date or allows to set the alarm time. The changes between the modes (realized by pressing a *Mode* button on the clock) are modeled in the Statechart by transitions labeled with the event Mode. The nested state Alarm allows to change to modes for setting the hours and the minutes (realized by pressing a *Select* button on the clock) which is modeled by the transitions labeled with the event Select. The Set event increments the number of hours or minutes which are currently displayed.

Fig. 3.3 shows the abstract syntax of the radio clock model depicted in Fig.3.2 as graph typed over the type graph of the Statechart simulation language shown in Fig.3.1 before the beginning of the simulation (the subgraph with unfilled nodes only), and in its initial simulation state (the complete graph with the Object node and the event queue). This initial state is generated from the Statechart before the simulation by applying the initial simulation rules initial("mode"), addEvent("mode"), addEvent("select"), addEvent("set"), addEvent("mode") (see Fig. 3.4).

Note that we use a special shortcut notation to show the values of state and transition

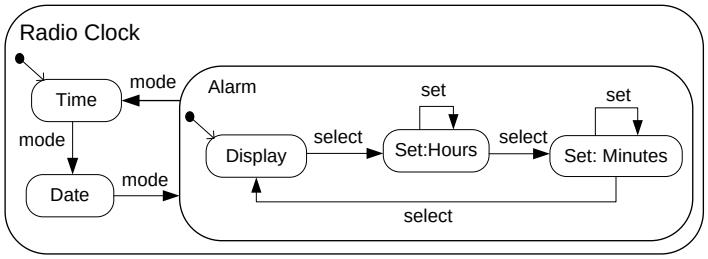


Figure 3.2: A Statechart Modeling the Behavior of a Radio Clock

attributes: We write names of states and transition-triggering events in quotation marks, and add the string init to an initial state (i.e. the value of the boolean attribute init is true for this state).

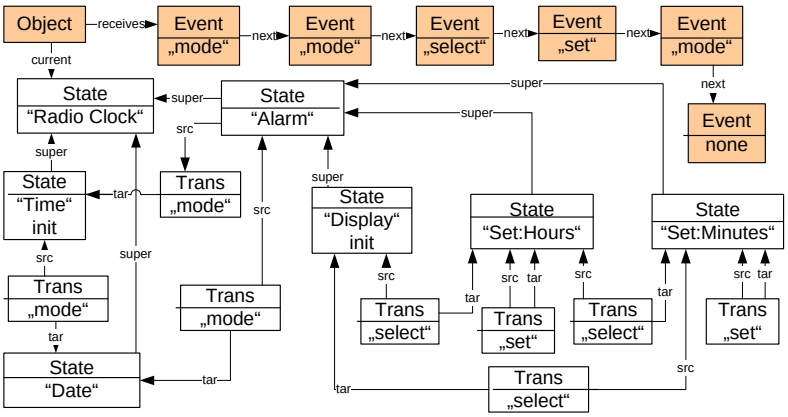


Figure 3.3: Abstract Syntax of the Radio Clock Statechart

The Simulation Specification for Statecharts with OR-states

The simulation specification of the visual language VL_{SC-OR} of Statecharts with OR-states is defined by $SimSpec(VL_{SC-OR}) = (TG_{SCsim}, P_{SCsim})$ where TG_{SCsim} is the type graph shown in Fig. 3.1, and P_{SCsim} is the set of simulation rules depicted in Figs. 3.4 and 3.5, and explained below.

To initialize the simulation, rule initial activates the root state by linking an Object symbol to it, together with an initial event queue containing one event. Moreover, rule addEvent allows to fill the event queue (see Fig. 3.4). In this way, the events that should be processed during a simulation run, can be defined in the beginning of the simulation. Alternatively, events also may be inserted at the end of the queue while a simulation is running.

The intended semantics for our Statecharts variant requires that if an OR-state (a composite state with substates) is reached, the initial state within the OR-state becomes the active one. Thus, before a transition is processed, the OR-state has to be entered. If the

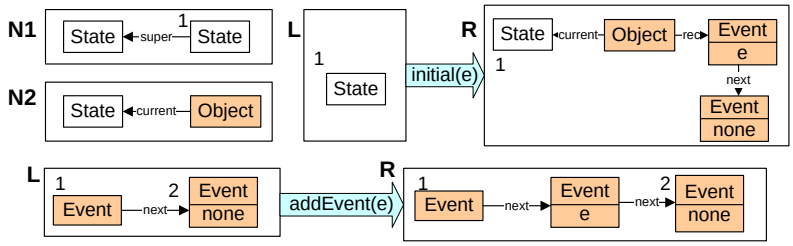


Figure 3.4: Initial Simulation Rules for Statecharts with OR-States

initial state inside the OR-state is again a composite one, the procedure has to be repeated until the object's current link points to the innermost simple state. This is modeled by rule down in Fig. 3.5.

A transition is processed if its pre-state is active and its triggering event is the same as the event which is received by the object (the first event in the queue). Afterwards, the state following the transition becomes active, the event of the processed transition is removed from the queue, and the previously active states (the pre-states of the transition) is not active anymore. For our simulator we use the object's current link as pointer to the current active state. The simulation rule trans models the relinking of the current link to the next active state and the updating of the event queue.

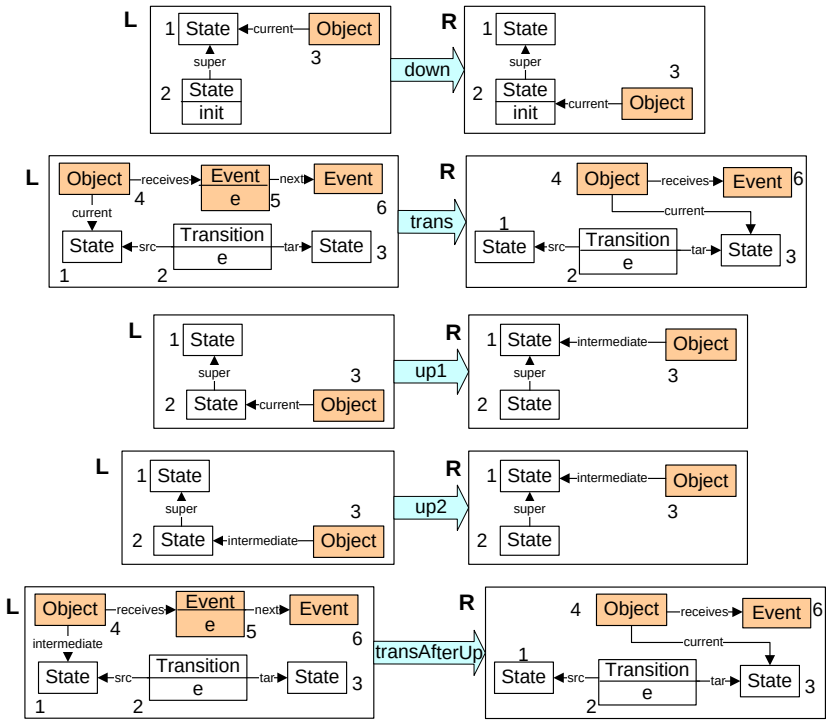


Figure 3.5: Further Simulation Rules for Statecharts with OR-States

Rule up1 models the fact that we can change the state due to transitions departing from

any of the super-states of the current state. Thus, this rule allows going up in the state hierarchy one level. As we do not want that rule down can be applied immediately after up has been applied, we replace the current link by an intermediate link which cannot be matched by rule down. Of course it must be possible to go up more than one step in the hierarchy, so we need another rule for the case that after going up one step, still no transition can be processed. This is realized by rule up2, which is similar to rule up1 except that it contains the intermediate pointer instead of the current pointer. In order to process a transition after going up in the hierarchy, we need a second transition rule, called transAfterUp, which equals rule trans except that it has an intermediate pointer instead of the current pointer in the left-hand side. The intermediate link is removed by the rule, and a current link is inserted again pointing to the next state.

The order and control of rule applications can be restricted by priorities, thus allowing an automatic simulation, where the next rule is chosen only if it has the highest priority of all applicable rules. Rule down gets the highest priority because it must be checked after each transition step if a complex OR-state has been reached which must be entered. At medium priority level are the rules transition and transitionAfterUp. This means, if down is not applicable in the current system state, then a transition should be processed (if possible). The lowest priority is for the rules up1 and up2, i.e. if no OR-state can be entered and no transition can be processed, then the active state changes to its super state. Thanks to the intermediate pointer, rule down cannot be applied immediately afterwards although it has the highest priority. Instead, rule transitionAfterUp is applied if possible. Afterwards, rule down could be applied again.

In the case that none of the rules in Fig. 3.5 is applicable any more, the simulation is finished, and the subset of final simulation rules is applied to remove the simulation symbols from the final model state diagram. The final simulation rules are not depicted. They equal the initial simulation rules in Fig. 3.4, without the NACs and with swapped left-hand and right-hand sides.

Fig. 3.6 shows some of the corresponding model-specific simulation rules which are the basis for the definition of the animation specification. For the RadioClock Statechart, the semantics is the same for both the set of model-independent simulation rules shown in Fig. 3.5 and the set of model-dependent simulation rules shown in Fig. 3.6.

Fig. 3.7 shows part of a scenario, according to the simulation specification $SimSpec$ (VL_{SC-OR}), applying the simulation rules to model state G_1, i.e. the radio clock Statechart in Fig. 3.3 (after the initial simulation rules have added the object node and its event queue, and before the final simulation rules delete them again). Due to space limitations, the transitions in Fig. 3.7 are drawn as arcs inscribed by the name of their triggering events instead of transition nodes connected with source and target edges to their adjacent states.

The complete scenario of which Fig. 3.7 shows the middle part, starts with G (the unfilled nodes of the radio clock Statechart in Fig. 3.3 and adjacent edges), and begins with

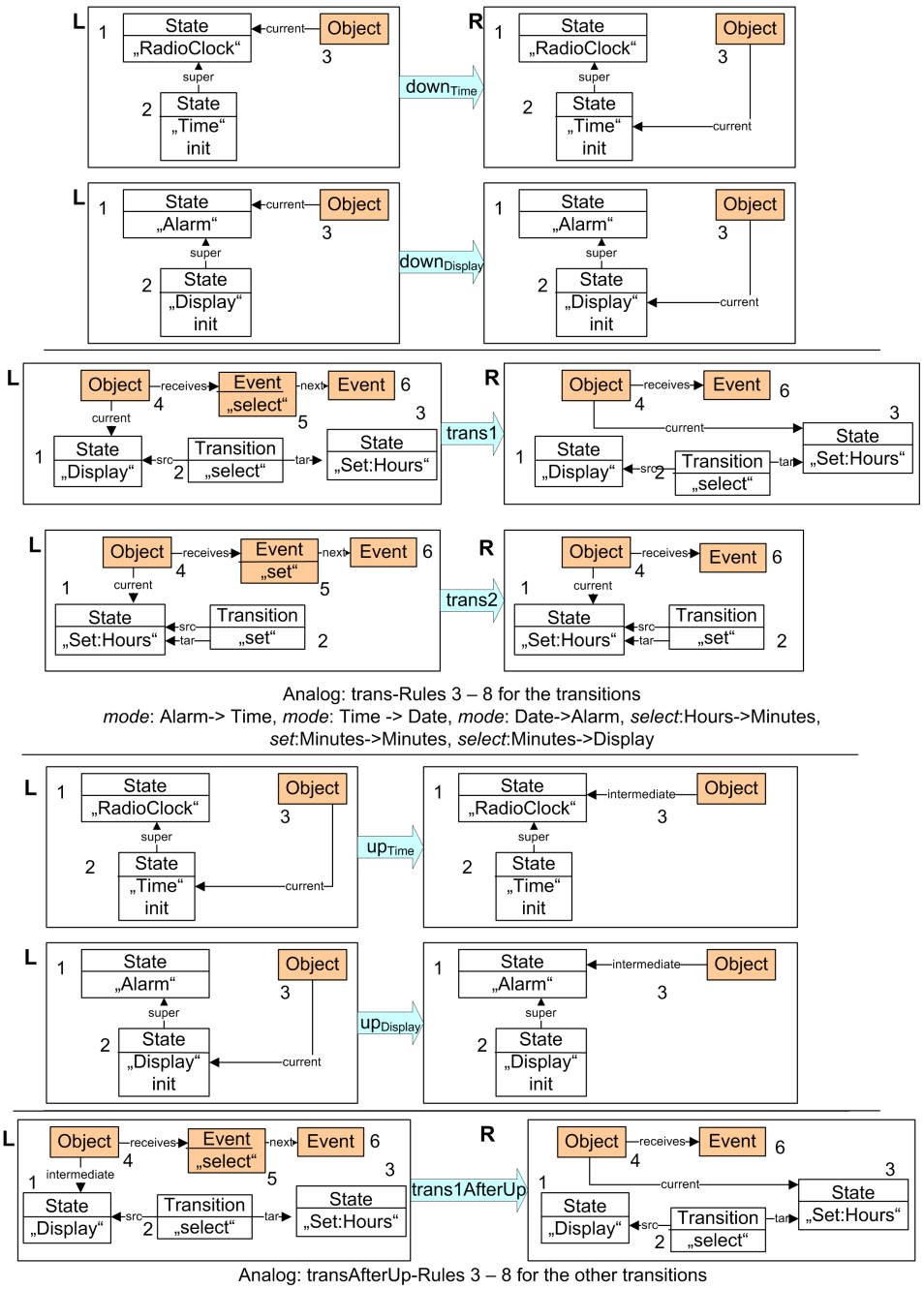


Figure 3.6: Model-Specific Simulation Rules for the RadioClock Statechart

the generation of the object pointing to the state ”Radio Clock” and the event queue, thus generating model state G_1. Then follows the partial scenario depicted in Fig. 3.7, and afterwards, the final simulation rules delete the simulation elements (the Object node, the Event node and the edges of types receives and current). Thus, the original model G is also the final diagram of the scenario.

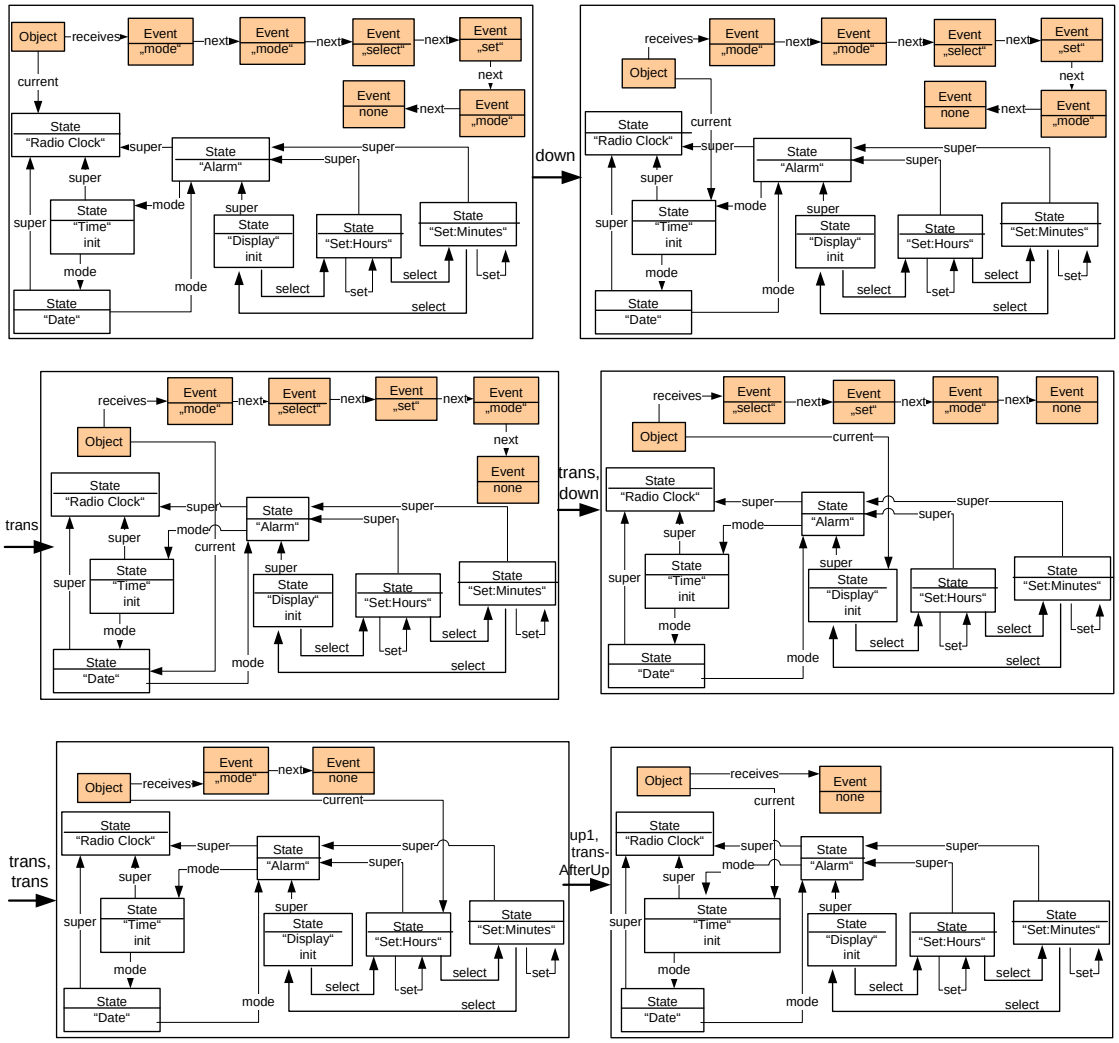


Figure 3.7: Part of a Scenario of the Radio Clock Model

The scenario is in $ISem(G)$, because the restriction to TG_A of each diagram in the scenario yields G (i.e. the white part in Fig. 3.7, which corresponds to G, is never changed in the scenario). Looking at the simulation rules in Fig. 3.4 and Fig. 3.5, we can argue that in general all scenarios starting at a model G are in $ISem(G)$ because none of the rules add or delete symbols typed over TG_{SC_A}, the abstract type graph of the basic Statechart VL (defined in Section 2.3.2.1), and all graph objects for simulation which are added by the rules, are deleted again in the end, using the final simulation rules.

3.2 Simulation Specifications for Visual Models

For some visual modeling languages, a simulation specification cannot be defined by finite, sequential graph transformation systems as described in Section 3.1. This makes it impossible to implement a graph transformation based simulation tool for this kind of VLs.

Especially for Petri nets, the number of rules modeling the firing of transitions in general would be infinite, as we would need separate rules for all combinations of all possible numbers of places in a transition's predomain and all possible numbers of places in its postdomain. For the variant of Place/Transition nets, we even have the additional problem to represent arbitrary numbers of black tokens on each place.

There are two solutions to the problem of representing indefinite many occurrences of graphical structures in rules: One solution is to use parallel instead of sequential graph transformation (see e.g. [Tae96, EHKP91, KW87, EK76]). The essence of parallel graph transformation is that (possibly indefinite) sets of rules which have a certain regularity, so-called rule schemes (see Fig. 3.8 for an example for a Petri net firing rule scheme), can be described by a finite set of rules modeling the elementary actions.

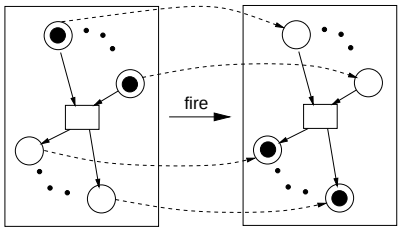


Figure 3.8: Rule Scheme Modeling Transition Firing in Petri Nets

For instance, when modeling the firing of a Petri net transition, the elementary actions would be the removal of a token from a place in the transition's predomain, and the addition of a token to a postdomain place. For the description of such rule schemes, the concept of amalgamating rules at subrules is used which is based on synchronization mechanisms for rules. This approach is treated in detail in Section 3.3.

The second solution to the problem is to define a general translation (compilation) of (finite) models to finite, sequential graph transformation systems (simulation specifications). This approach is the "classical" relationship of Petri nets and graph transformation systems in the literature. We discuss the techniques of translating models to simulation specifications for Petri nets and for Statecharts in this section.

In Petri nets, the elementary form of a step from one system state to the next is the firing of a transition. The straightforward technique for behavior simulation therefore is playing the token game. One of the first who discussed the relationship between graph grammars and Petri nets was Kreowski [Kre81]. He associated a graph rule to each transition, with

the rule being applicable if and only if the transition is allowed to fire. Tokens within a place were modeled as a bundle of new nodes connected to the node representing the place. This approach is able to handle both places with bounded capacity and places with unbounded capacity and can also be extended to individual tokens. Parisi-Presicce et al. [PPEM87] use a structured alphabet for labeling places with tokens. This alphabet has to allow changes of node labels in rule morphisms. In the case of high-level Petri nets, multiple and individual tokens can be represented using multisets as labels and the multiset inclusion as the structure of the alphabet. A further step is to allow arbitrary categories to label the places. For the high-level Petri net variant in this thesis, *algebraic high-level nets*, we use graph transformation systems where the tokens are modeled by data of arbitrary algebraic data types.

3.2.1 Model Transformation into Graph Transformation Systems

A sound definition of semantics for visual behavior models in a VL must relate the elements of each visual model in a unique way to the semantic domain elements that represent the model's meaning (i.e. the simulation specifications). Often, language designers explain the semantic mapping informally through examples and plain natural language. An adequate semantic mapping for a visual language is a *model transformation*, defined by a function from the language's syntax VL_1 to its semantic domain (i.e. another visual language VL_2, an adequate mathematical formalism, or a target textual language such as a programming language, etc.).

In this thesis, the semantic domain is again given by graph transformation systems. Note that in general, the so-called *denotational semantics* allows much more formalisms for the semantic domain, e.g. natural language, general-purpose formal languages such as logic and algebraic specification languages.

The semantic mapping has to be given as a general definition of how a visual behavior model G of VL_1 is translated ("compiled") into a simulation specification $SimSpec(G) = (TG_{sim}, P_{sim})$, a sentence of VL_2, the visual language of graph transformation systems. The simulation rules in $SimSpec(G)$ are model-specific, i.e. they can be applied only to TG_{sim}-typed diagrams which are specific to G, i.e. diagrams corresponding to different markings of one Petri net or to different states of one specific Statechart. Usually, TG_{sim} extends the abstract type graph of VL_1 by symbol and link types for simulation. The set of transformations defined by the simulation specification resulting from the model transformation yields the *GTS-compiler semantics* of the behavior model G.

The way to define a semantic mapping depends on the syntax of the modeling language and the choice of the semantic domain. In this thesis, both the syntax of the modeling language VL_1 and the syntax for the language VL_2 are defined by typed graph transformation systems. In this case, graph transformation works nicely to define the semantic mapping,

as well. The mapping language must include both VL_1 and VL_2, and the mapping rules define the translation from VL_1 diagrams to their simulation specification. The model G is typed over the abstract type graph TG_1 of the source language VL_1. Intermediate graphs of the model transformation are typed over a union of both type graphs (plus maybe additional auxiliary symbols) TG. The mapping definition ensures that the result is a graph typed over the type graph TG_{GTS} of the target language VL_2 of graph transformation systems, as shown in Fig. 3.9.

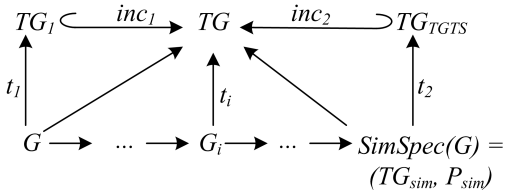


Figure 3.9: Type Graphs for Model Transformation to Graph Transformation Systems

Apart from graph transformation systems for the definition of model transformation, textual mathematical descriptions of the semantic mapping (e.g. by functions) work well, too, since the language designer can deal with all relevant elements within the mathematical framework, but it may be less intuitive for visual languages.

In this section we define as example the semantic mapping from Condition/Event nets into graph transformation systems in two ways: first, by giving mathematical functions which define in general how syntactical constructs from the modeling language (transitions and their pre- and postdomains) are translated to graph transformation rules; second, by specifying a graph transformation system for the model transformation. Here, the syntactical constructs from the modeling language (Petri net transitions and their pre- and postdomains) are contained in the left-hand sides of the transformation rules, and the right-hand sides add respective nodes and edges to the graph representing the set of graph transformation rules from the simulation specification, i.e. the semantic domain.

3.2.2 Definition of GTS-Compiler Semantics for Visual Models

Definition 3.2.1 (*Visual Language VL_{TGTS} for Typed Graph Transformation Systems*) The visual language VL_{TGTS} for typed graph transformation systems is defined by the abstract type graph TG_{TGTS} shown in Fig. 3.10. Each sentence of the language consists of an optional type graph and a set of graph transformation rules, each having a set of NACs, one left and one right-hand side. All NACs, rule sides and the type graph are graphs, i.e. they consist of (attributed) nodes and edges. Suitable constraints (given by multiplicities in Fig. 3.10) ensure e.g. that each rule has precisely one left- and one right hand side, and each edge has exactly one source and one target node.

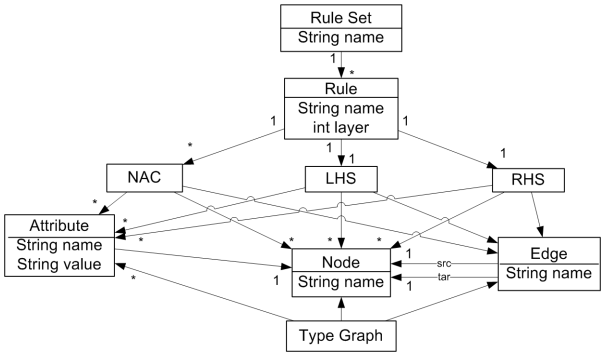


Figure 3.10: Type Graph TG_{TGTS} for Typed Graph Transformation Systems

△

Definition 3.2.2 (*Simulation Specification for Model G*)
Let TG_A be the abstract type graph of a visual behavior modeling language VL, and TG_{sim} be the extension of TG_A for simulation, i.e. $TG_A \subseteq TG_{sim}$. Let TG_{TGTS} be the type graph given in Fig. 3.10 defining the language VL_{TGTS} of typed graph transformation systems. A *simulation specification for* G is a mapping $SimSpec(G) : VL(TG_A) \rightarrow VL_{GTS})$ which maps each visual behavior model $G \in VL$ to a typed graph transformation system $SimSpec_{VL_S} = (VL_S, P_S)$, called *simulation specification of* G, where VL_S is a visual language typed over TG_{sim}, such that VL_S is closed under simulation steps, i.e. $G_S \in VL_{sim}$ and $G_S \overset{p_S}{\Longrightarrow} H_S$ via $p_S \in P_{sim}$ implies $H_S \in VL_{sim}$, and for all $G_S \in VL_S$ we have $G_S|_{TG_A} = G$.

The rules in P_{sim}, typed over TG_{sim}, are called *simulation rules for* G. △

Definition 3.2.3 (*Simulation and GTS-Compiler Semantics of Visual Models*)
Let $SimSpec_{VL_S} = (VL_S, P_{sim})$ be the simulation specification of $G \in VL$. The *simulation* $Sim(G)$ of the visual model G consists of all graph transformations applying rules from P_{sim}.

$$Sim(G) = \{G_1 \overset{P_{sim}}{\Longrightarrow} G_2 | G_1, G_2 \in VL_S \; and \; G_1|_{TG_A} = G_2|_{TG_A} = G\}$$

Each transformation $G_1 \overset{*}{\Longrightarrow} G_2$ of $Sim(G)$ shows one specific simulation run, called *simulation scenario*. The behavioral semantics of G, called *GTS-compiler semantics of* G, is defined by $CSem(G) = Sim(G)$. △

3.2.3 GTS-Compiler Semantics for Condition/Event Nets

In this section we present the definition of simulation specifications for Condition/Event nets (C/E nets). The simulation of further variants of Petri nets (Place/Transition nets and

Algebraic High-Level nets) is discussed in the application sections 3.5.1, and 3.5.2, and semantical compatibility of different approaches is shown.

In simulation specifications for Petri nets, we have to ensure that on the one hand, a transition in the net is enabled if and only if the corresponding rule is applicable to the visual sentence corresponding to the net, and, on the other hand, firing a transition in the net corresponds to a derivation step in the grammar and vice versa.

The token game then can be simulated by applying the rules of the simulation specification to a model G of the Petri net VL. The left-hand side defines the applicability condition, i.e. the transition's predomain with the minimal expected marking for the transition to be enabled. The rule removes the marking from the transition's predomain and adds the required tokens to the places in its postdomain.

Example: The Echo Algorithm as C/E Net

As an example for a visual behavior model G, we model the echo algorithm for a network of hosts (taken from [Wal95]) as C/E net. A number of hosts (three in our example) are connected by bidirectional message channels. One distinguished host (the boss) tests the connections between all hosts in the network. The algorithm works as follows: the boss sends a broadcast message to the other hosts in the network. Each host receiving a message sends out a message to every other host in the network except to to the sender of the initial message (his target group). After a host has received messages from all hosts in his target group, he sends a final message to the boss. Thus, after the boss got messages back from all hosts in the network, he knows that the connections between all host nodes are all right. Otherwise, one or more communication channels are not working.

Due to space limitations, we show the visual behavior model for the echo algorithm using its concrete layout in Fig.3.11. We model a network with a boss host B and two other hosts H1 and H2. Formally, the abstract model G is a diagram of the visual behavior modeling language for unmarked C/E nets, i.e. it is typed over the abstract VL alphabet TG_A for C/E nets (Fig. 2.5 in Example 2.2.3) and has been constructed by applying the abstract VL syntax rules for C/E nets (Fig. 2.7 in Example 2.2.14).

Simulation Alphabet for C/E Nets

Let us now define the alphabet TG_{CEsim} for the language of marked C/E nets, i.e. we extend the language of unmarked C/E nets by a node type for tokens Token and an edge type EdgeTK connecting each token to its respective place (see Fig. 3.12).

Now, the model transformation defining the semantics of G transforms the unmarked C/E net G to a graph transformation system $GTS_{CEsim} = (TG_{CEsim}, P_{CEsim}(G))$, where the rules in $P_{CEsim}(G)$ model the marking, unmarking and firing of transitions in G.

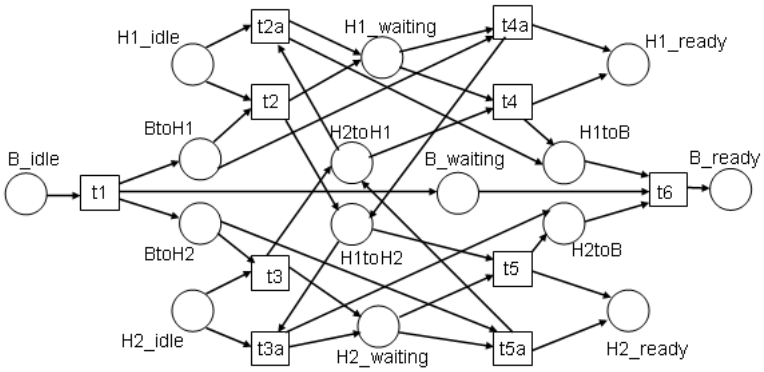


Figure 3.11: C/E net modeling the Echo Algorithm

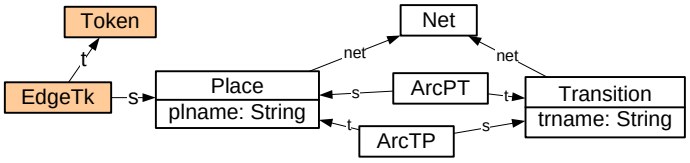


Figure 3.12: Simulation Alphabet for C/E Nets

Simulation Rules for the Echo Model

A subset of the simulation rules called *marking rules* are defined which are applicable to G (as they contain no token symbols in the left-hand side) and add the initial marking to the respective places. Another subset of $P_{CEsim}(G)$, called *firing rules*, correspond to the firing of transitions, and a third subset of rules, called *unmarking rules*, deletes all tokens after the firing rules cannot be applied anymore.

Note that the simulation rules do not contain transition symbols, as the effect of firing a transition (which is modeled by a rule) changes only the marking.

The simulation rules in $P_{CEsim}(G)$ are defined by the following model transformation algorithm, given in a set-theoretic way:

Marking Rules:
For each place symbol in the model we define a rule with negative application condition, $N \leftarrow L \leftarrow I \rightarrow R$ where N, I, L and R contain the place symbol, attributed by its place name. In N and R we add a token symbol, which is connected to the place symbol by an edge of type EdgeTk. The rule morphisms map the place symbol from I to the place symbol in L and R, respectively and the morphism $L \rightarrow N$ maps the place symbol in L to the place symbol in N.

Firing Rules:
For each transition in the model with the set of predomain places $Pre = \{p_1, .., p_n\}$ and the set of postdomain places $Post = \{o_1, .., o_m\}$, with $n, m \in I\!N$, we define a rule

$N_{1,..,m} \leftarrow L \leftarrow I \rightarrow R$, where I, L and R contain the place symbols for all places $p \in Pre \cup Post$, and the rule morphisms define the identities on all place symbols. The negative application conditions N_i, contain the place symbols corresponding to the places $o_i \in Post, i \in \{1, .., m\}$. The morphism from L to N_i consists of identic mappings of the place symbols corresponding to places $o_i \in Post$. To L we add a token symbol Token_j for each place symbol corresponding to $p_j \in Pre, j \in \{1, .., n\}$ and an EdgeTK edge connecting Token_j to the place symbol for p_j. To R we add a token symbol Token_i for each place symbol corresponding to $o_i \in Post$ and an EdgeTK edge connecting Token_i to the place symbol for o_i.

Unmarking Rules:
For each place symbol in the model we define a rule $L \leftarrow I \rightarrow R$ where L, I and R contain the place symbol, attributed by its place name. In L we add a token symbol, which is connected to the place symbol by an edge of type EdgeTk. The rule morphisms map the place symbol from I to the place symbol in L and R, respectively.

One sample marking rule, one firing rule, and one unmarking rule defined according to this algorithm are shown in Fig. 3.13. The complete set of simulation rules $P_{CEsim}(G)$ with G being our echo model in Fig. 3.11, consists of a marking and an unmarking rule for each place, and a firing rule for each transition of the net.

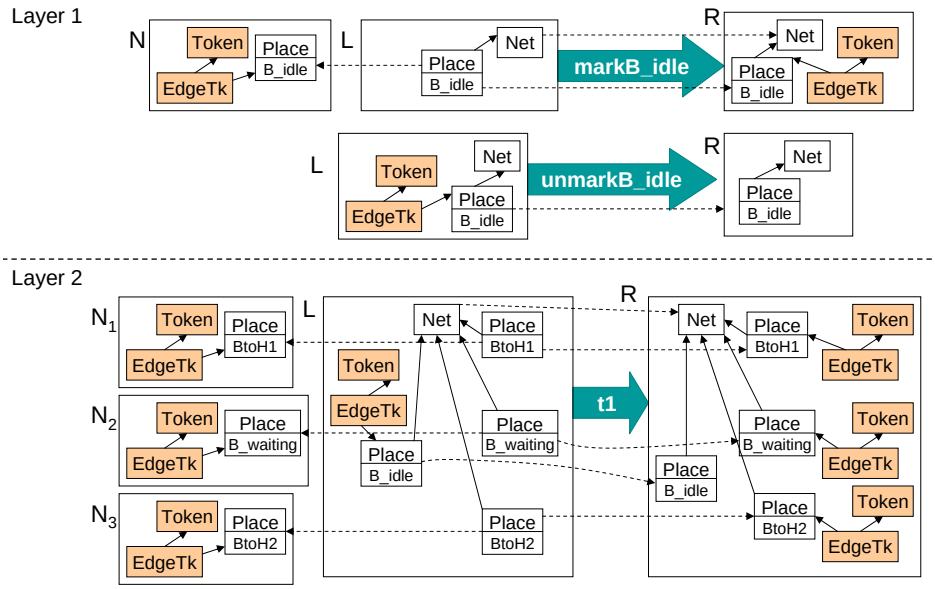


Figure 3.13: Three Simulation Rules for the Echo Model

Note that for C/E net firing rules (in contrast to P/T nets), the firing of a transition is forbidden if one or more post places are marked already. This is modeled by a set of NACs corresponding to marked post places for each firing rule.

Of course, the application of marking rules should be finished, thus having defined an

initial marking, before starting the application of firing rules leading to a simulation run. Analogously, the unmarking rules should be applied only after the simulation rules have been applied as long as possible.

The formal simulation specification for the echo model G thus is given by the typed attributed graph transformation system $SimSpec(G) = (TG_{CEsim}, P_{CESim}(G))$.

Simulation Rules for C/E Nets by Model Transformation GTS

The same model transformation can be defined as graph transformation system instead of the algorithm given above. Here, the target type graph TG_{GTS} (shown to the right in Fig. 3.14) defines the language of graph transformation systems, represented as typed graphs. The root types are RuleSet and TypeGraph which correspond to the structure of graph transformation systems. A RuleSet contains a set of Rules which have a left-hand side (LHS), a right-hand side (RHS), and (possibly) a set of negative application conditions (NAC). The rule graphs are described by Nodes, their Attributes and Edges. The different parts of a rule are presented in an integrated way, e.g. a node occurring in LHS and RHS is presented only once but with two edges both from the LHS and from the RHS pointing to it.

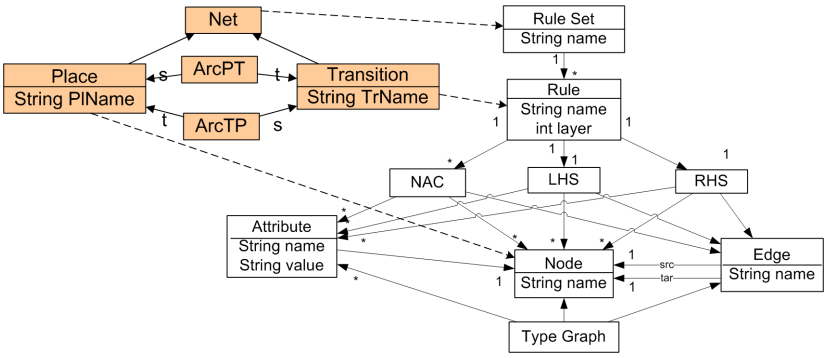


Figure 3.14: Type Graphs of Source VL (C/E nets) and Target VL (Graph Transformation Systems) for Model Transformation

The model transformation itself is given by a set of graph transformation rules (see Fig. 3.15 and 3.16) over the union of the type graphs for the source and the target VL (called model transformation rules). The rules relate structures from the source VL to graph objects in the target graph transformation system. Basically, the model transformation rules correspond to the steps described in the model transformation algorithm above: Rule init defines a RuleSet for a Net, rule mark/unmark adds a marking rule and an unmarking rule for each place node to the rule set (Fig. 3.15).

The firing rules are constructed by the model transformation rules in Fig. 3.16. Rule Trans adds a Rule to this RuleSet for each transition in the net. Rule PrePlace adds place

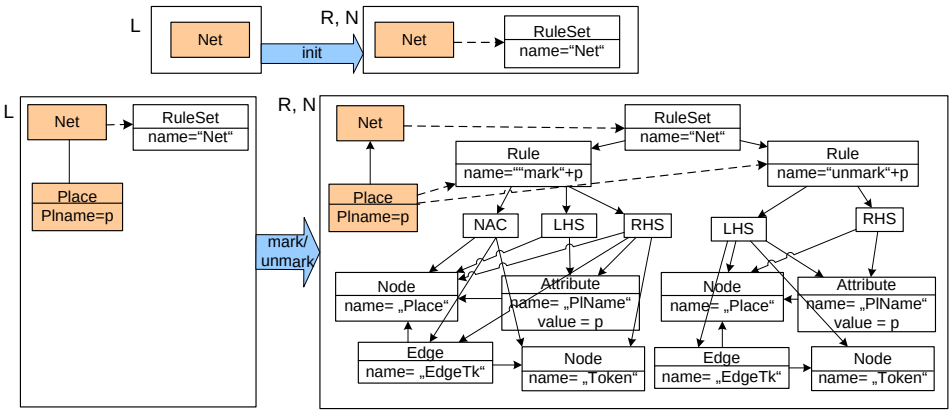


Figure 3.15: Model Transformation Rules Constructing Marking and Unmarking Rules for C/E Nets

symbol nodes for all places in the transition's predomain to both of its rule sides, and links tokens to these place symbol nodes in the left-hand rule side. Rule PostPlace adds place symbol nodes for all places in the transition's postdomain to both of its rule sides and links tokens to these place symbol nodes in the right-hand rule side. Moreover, the rule generates NACs each containing one marked place node which corresponds to a postdomain place.

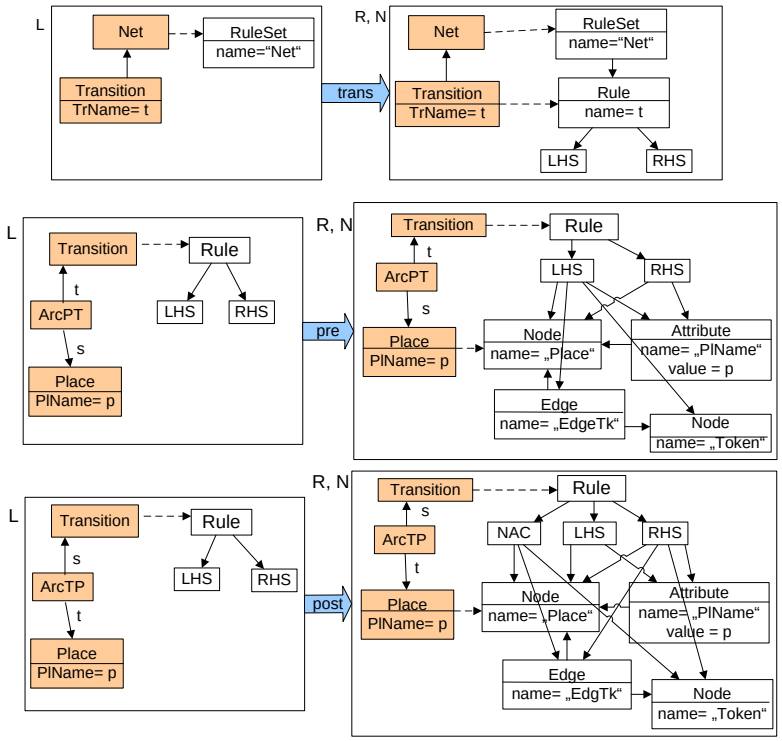


Figure 3.16: Model Transformation Rules Constructing Firing Rules for C/E Nets

Note that for all model transformation rules, the NAC is isomorphic to the right-hand

rule side. Thus, the rule can be applied at most once at a given match.

The first two model transformation steps of the transformation generating the firing rule for transition t1 from the abstract syntax of the Echo model G (see Fig. 3.11) are shown in Fig. 3.17. (For the sake of clarity, only the transition t1 and its environment is shown as source model G in Fig. 3.17).

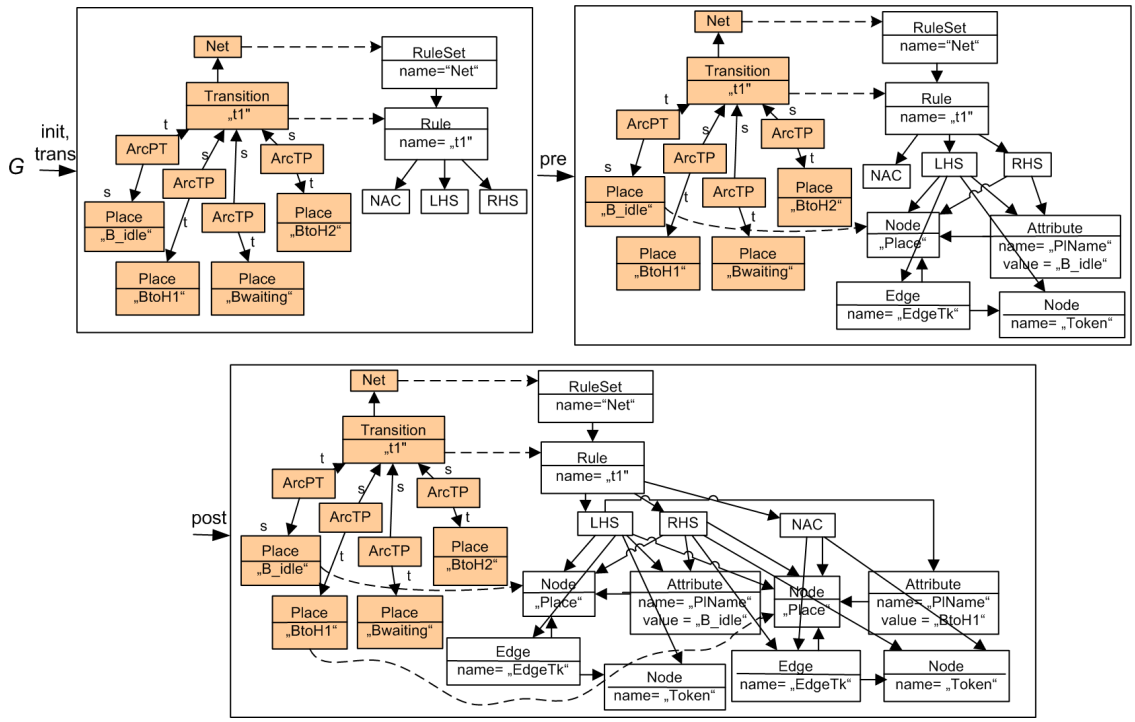


Figure 3.17: Model Transformation Steps Translating Transition t1 to a Graph Transformation Rule

After the model transformation rules have been applied to the model as long as possible, the resulting graph still includes elements of both the source and the target language and auxiliary symbols (the dashed reference arcs). The subgraph representing the graph transformation system is obtained by restricting the complete graph to the type graph of the target language. Then, the restricted graph can be translated into a rule set in a straightforward way, yielding the desired simulation specification for the model. For the Echo model, the rule set generated by the model transformation rules in Fig. 3.15 and 3.16 corresponds to the rule set generated according to the model transformation algorithm (conf. Fig. 3.13).

A Simulation Run of the Echo Model

For a simulation run starting with model G, the three left places Boss idle, H1 idle and H2 idle are marked by the respective marking rules to obtain the initial system state. Thus,

initially the boss is ready to send his messages and the two other hosts in the network (Host1 and Host2) are ready to receive the message and to react to it.

In order to have a compact visualization of the simulation run (which is defined by the simulation rules on the abstract syntax, only), let us assume an adequate concrete syntax definition in addition to the abstract simulation alphabet TG_{CEsim}, where tokens are visualized by black circles inside of place ellipses. Starting from the initial state Init (shown in Fig. 3.11), a scenario (i.e. a simulation run) of the Echo net behavior is depicted in Fig. 3.18. After having generated the initial marking by adding tokens to the places B1_idle, H1_idle and H2_idle, the boss sends his message to Host1 and Host2; Host1 forwards the message to Host2; Host2 himself forwards his own message from Boss to Host1; Host1 is ready now, as he has sent messages to every one else (to Host2) and he has received messages from everyone else (from Host2). As a last action, Host1 replies to Boss. The same situation applies to Host2. He is ready as well and as his last action he sends a message to Boss. The final event is that Boss gets in a "ready" state because he now has received replies from all hosts in the network. In this final state, no firing rules are applicable anymore. Thus, the unmarking rules delete all tokens from the model, resulting again in model G, such that another simulation run may be be started.

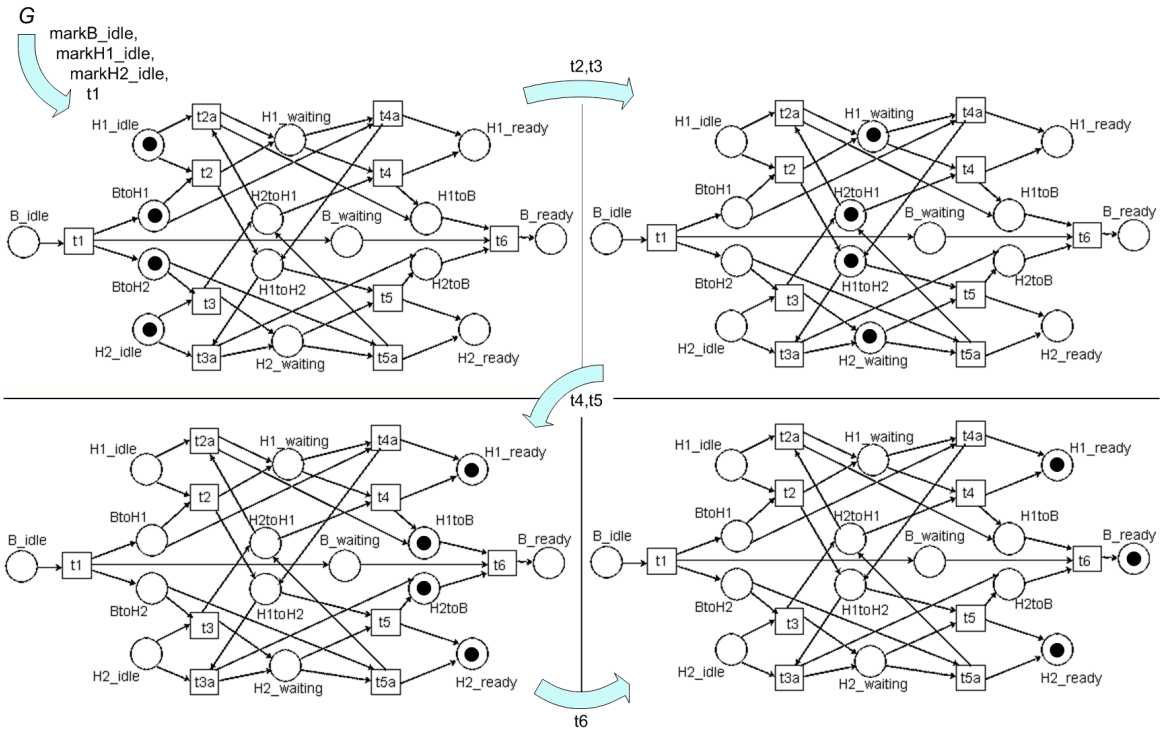


Figure 3.18: Simulation Run of the Echo Model

3.3 Amalgamated Simulation Specifications

Although graph transformation systems are an expressive, graphical and formal way to describe computations on graphs, they have limitations. For example, when describing the interpreter semantics of visual behavior modeling langauges, one often has the problem of modeling an arbitrary number of parallel actions. This can be modeled by a sequential application of rules, in which we have to explicitly encode an iteration on all the actions to be performed. Usually, this is not the most natural nor efficient way to express the semantics. Thus, it is necessary to have a more powerful means to express parallel actions.

In this section, we propose the use of parallel graph transformation [Tae96] to define the semantics of visual behavior modeling languages, for which no interpreter semantics can be given. The essence of parallel graph transformation is that (possibly infinite) sets of rules which have a certain regularity, so-called rule schemes, can be described by a finite set of rules modeling the elementary actions. As stated before, when modeling the firing of a Petri net transition, the elementary actions would be the removal of a token from a place in the transition's predomain and the addition of a token to a postdomain place. For the description of such rule schemes the concept of amalgamating rules at subrules is used. This concept (the *amalgamation approach*) is used in this section to describe the application of rule schemes in an unknown context.

The simplest type of parallel actions is that of *independent actions*. If they operate on different objects they can clearly be executed in parallel. If they overlap just in reading actions on common objects, the situation does not change essentially. In graph transformation, this is reflected by a *parallel rule* which is a disjoint union of rules. The overlapping part, i.e. the objects which occur in the match of more than one rule, is handled implicitly by the match of the parallel rule. As the application of a parallel rule can model the parallel execution of independent actions only, it is equivalent to the application of the original rules in either order.

If actions are not independent of each other, they can still be applied in parallel if they can be synchronized by subactions. If two actions contain the deletion or the creation of the same node, this operation can be encapsulated in a separate action which is a common subaction of the original ones. A common subaction is modeled by the application of a *subrule* of all original rules (called *elementary rules*). The application of rules synchronized by subrules is then performed by gluing the elementary rules at their subrules which leads to the corresponding *amalgamated rule* . The application of such a rule is called *amalgamated graph transformation*. . An example of an amalgamated graph transformation rule is given in Fig. 3.19. Here, two elementary rules *er1* and *er2* have the addition of a loop to a node in common. This common interface is modeled as a subrule. The amalgamated rule contains the common action and, additionally, all actions from the elementary rules that do not overlap. Dashed arrows in Fig. 3.19 indicate rule morphisms, the

vertical arrows are the embeddings of the subrule into the corresponding instances of the elementary rules, and the embeddings of the elementary rules into the amalgamated rule.

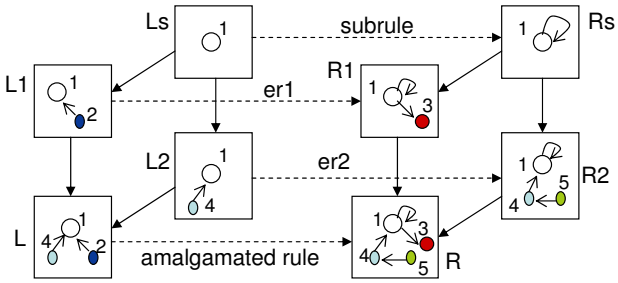


Figure 3.19: Construction of an amalgamated graph rule

Note that in Fig. 3.19 we have exactly one instance of each elementary rule. It is possible to have arbitrary many instances which are all glued at one subrule. The number of instances depends on the match of the rule scheme into a specific graph. Formally, the synchronization possibilities of actions are defined by an interaction scheme consisting of a set of elementary rules, a set of subrules and a set of subrule embeddings from the subrules into the elementary rules (see Def. 2.1.28).

Subrules with NACs can be embedded only into an elementary rule with at least the same NACs, no matter if the elementary rule contains additional NACs. The amalgamated rule collects the NACs of all rules. An interaction scheme can also be seen as a graph whose nodes are labelled by elementary rules and subrules. Edges are labelled by subrule embeddings.

In addition to the specification how elementary rules should be synchronized, we have to decide where and how often a set of elementary rules should be applied. The basic way to synchronize complex parallel operations is to require that a rule should be applied at *all different matches* it has in a given graph. This expresses massively parallel execution of actions. For the purpose of simulation in this thesis, we restrict the *covering of* G (the image of all different matches from copies of elementary rules in G) to all different elementary matches that overlap in one match of their common subrule. For other, more complex covering constructions see [Tae96].

We say, a graph H is *directly parallel derivable* from graph G by an interaction scheme if there is an amalgamated rule r constructed by gluing the elementary rules at one of their corresponding subrules, and if all matches from the elementary rules to G overlap in their subrule match.

A *parallel graph transformation system* $PGTS$ consists of a start graph and a set of interaction schemes. The language $L(PGTS)$ of a parallel graph transformation system is given by all graphs which are parallel derivable from the start graph by a finite number of direct parallel derivation steps applying amalgamated rules constructed from I.

Example 3.3.1 explains the principle of transition firing in P/T nets for transitions with an arbitrary number of input and output places and demonstrates how the context of an element (the environment of a transition) is modeled by synchronizing as many elementary rules as needed for the context according to an underlying interaction scheme.

Example 3.3.1 (*Firing of Transitions in P/T nets as PGTS*)
We model P/T nets [Rei85] as attributed graphs with two node types (for places and transitions), two edge types (for arcs from places to transitions and vice versa) and attributes of type Nat for the number of tokens and the arc weights. As start graph N of our PGTS, an arbitrary P/T net is allowed. The interaction scheme IS_{seq} (shown as graph on the left top of Fig. 3.20) describes sequential firing of arbitrary transitions. The subrule $trans$ glues together instances of the elementary rules get and put (see rules $trans$, get_1, get_2 and put_1 in Fig. 3.20). Thus, one amalgamated rule is constructed for each kind of transition in the P/T net. The subrule $trans$ ensures that all copies of elementary rules overlap in the same transition node. The elementary rule get removes the number of tokens from a predomain place corresponding to the arc weight, and the rule put generates the correct number of tokens on a postdomain place. In an amalgamated firing rule, there have to be as many copies of get and put as there are different matches to N, provided that they overlap in the same transition node. Thus, the pre- and postdomain of a transition will be completely covered by the corresponding amalgamated rule. Since the subrule $trans$ (and hence all elementary rules) delete the transition and reconstruct it, the dangling condition is not satisfied if not *all* places from the transition's pre- and postdomain are covered [Tae96]. Fig. 3.20 shows the construction of an amalgamated rule where the transition is matched to the upper transition of the P/T net N. There are two copies of get for the two places in the transition's predomain and one copy of put as there is only one place in the postdomain. Note that the variables for token numbers and arc weights are instantiated to different variables in the rule instances.

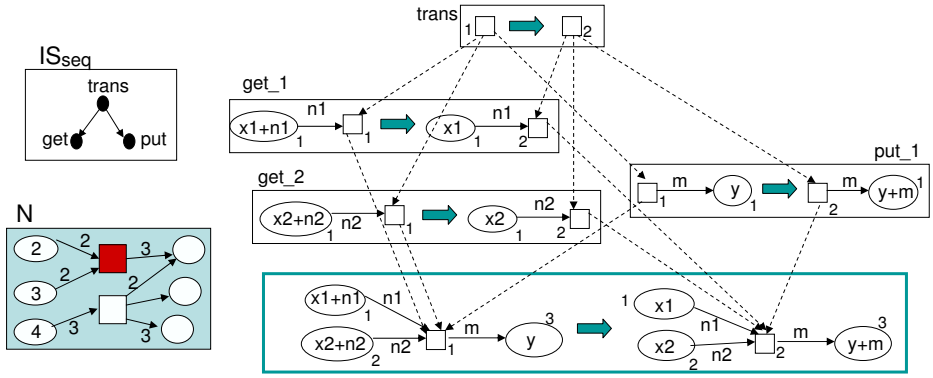


Figure 3.20: Firing of P/T Net Transition as Amalgamated Rule

△

Parallel firing of more than one transition can be similarly expressed by a parallel graph transformation system. The difference is that the elementary rules do not all have to be glued at a single subrule but can be glued at instances of different subrules. Here, we need two subrules (the transition-gluing one from example 3.3.1 and the empty rule) and require a slightly different covering construction (see [Tae96]).

3.3.1 Definition of Amalgamation Semantics for Visual Models

The formalization of the concepts for interaction schemes and rule amalgamation are given in detail in Section 2.1.4. These notions are used in this section for the formal definitions of simulation and behavioral semantics in the amalgamation approach.

Again, the simulation alphabet TG_{sim} extends the abstract VL alphabet TG_A over which model G is typed, by symbols indicating the current state of the model, e.g. tokens or pointers (see Def. 3.1.1).

In contrast to a simulation specification for a VL , where the set of simulation rules is defined directly over TG_{sim}, we here define first a so-called *virtual simulation specification* as parallel graph transformation system. This defines indirectly the simulation specification which is constructed as the set of all amalgamated rules over the virtual simulation specification.

A virtual simulation specification consists of interaction schemes and a start graph, the virtual model. This virtual model needs to supply all the information needed for the construction of all simulation rules as amalgamated rules. Hence, the underlying structure of the virtual model is the model G. Additionally, the virtual model models all states that are the precondition for all possible simulation steps to happen, i.e. that virtually enable all state changes. For Petri nets, this means that the virtual model has the same net structure as the model, and contains additionally the predomains of all transitions in form of "virtual" tokens, i.e. tokens corresponding to the arc inscriptions of the incoming arcs. In this way, each elementary rule can be matched to the virtual model as the left-hand side of each elementary rule contains token symbols corresponding to the arc inscription of an arc from a predomain place to the transition.

For Statecharts, in the virtual model, object nodes may be linked by a current pointer to each state. For a source state of a transition, the object node receives the same event by which the transition is triggered (and an arbitrary next event). If a state has more than one outgoing transition, then there are more than one events the object node receives. Thus, the Statechart elementary rules can all be matched to the virtual model, and amalgamated rules can be constructed for all possible transitions and for moving the current pointer up and down the state hierarchy.

Formally, a virtual model is constructed as union of the model G with all occurrences of the left-hand sides of all simulation rules and interaction schemes.

Definition 3.3.2 (*Virtual Model of G*)
Let G be a visual behavior model typed over TG_A. Let TG_{sim} be the type graph of the simulation alphabet extending TG_A, and I_{sim} the set of simulation rules and interaction schemes typed over TG_{sim}. Then the *virtual model* VM for G is constructed as follows: For each simulation rule and (in interaction schemes) for each elementary rule $L \leftarrow I \rightarrow R \in I_{sim}$, perform the following steps:

1. Initially, set $VM = M$.
2. For each rule or interaction scheme with elementary rules $r_i = (L_i \leftarrow I_i \rightarrow R_i)$, find all matches m from $L_i|_{TG_A}$ into the model VM,
3. at all found occurrences $m(L|_{TG})$ in VM add the graph objects from L which do not exist in $m(L|_{TG})$ so far (the simulation symbols), to $m(L|_{TG})$ in VM, thus extending the covering of m from $m(L|_{TG})$ to $m(L)$. For occurrences of different elementary rules which overlap at the match of their common subrule, the missing simulation symbols are added to VM only once to the elements of the match in the common subrule.

△

Remark: VM extends G by simulation symbols, i.e. $VM|_{TG_A} = M$. VM virtually enables all state transitions in the model, such that there are matches from all elementary rule instances into VM according to the instance interaction scheme I. □

Definition 3.3.3 (*Virtual Simulation Specification for a Model*)
A *virtual simulation specification* $VSimSpec(G)$ for a model G is a parallel graph transformation system $VSimSpec(G) = (TG_{sim}, I_{sim}, VM)$ consisting of the start graph VM (the virtual model of G), according to Def. 3.3.2, and a set of interaction schemes I_{sim}. △

Definition 3.3.4 (*Amalgamated Simulation Specification for Model G*)
Let $VSimSpec(G) = (TG_{sim}, I_{sim}, VM)$ be a virtual simulation specification for model G according to Def. 3.3.3. Then, an amalgamated simulation specification $ASimSpec(VM) = (TG_{sim}, P_{amalg})$ is given by the simulation alphabet TG_{sim} and the rule set P_{amalg} of all amalgamated rules that can be constructed from I_{sim} over VM. △

Definition 3.3.5 (*Simulation and Amalgamation Semantics of Model G*)
Let $ASimSpec(VM) = (TG_{sim}, P_{amalg})$ be an amalgamated simulation specification constructed over the virtual model VM of G. Then the simulation or amalgamation semantics of G, called $ASem(G)$, is the set of all graph transformations which can be derived by applying the simulation rules in P_{amalg}. Each transformation in $ASem(G)$ is called simulation run or scenario of G. △

Note that using the set of amalgamated simulation rules P_{amalg}, not all possible visual models of the VL can be simulated by the simulation rules in P_{amalg}. This is due to the fact that the amalgamated rules are constructed over a virtual model corresponding to one specific model G. Yet, the amalgamated simulation rules are not *model-specific*, as they contain no model-specific information like place or event names. The set of amalgamated simulation rules can be regarded as an incomplete set of *universal* simulation rules which include just the rules needed for a specific model, perhaps applicable also to other models, but not to all possible models.

3.3.2 Amalgamation Semantics for Statecharts with AND-States

In this section we extend the semantics for Statecharts with complex OR-States given in the interpreter approach in Section 3.1.2 by considering parallel AND-states. This could not be done in the interpreter approach where we have only sequential graph transformation systems, as there may be arbitrary many AND-states within one composite state. Such a situation can only be modeled by rule schemes instead of concrete rules, such that the number of occurrences in the left-hand side may be left open until the rule scheme is instanciated by a concrete model.

We give an amalgamation semantics for a simple Statechart variant with AND-states based on parallel graph transformation.

As we did already give a simulation specification for Statecharts with complex OR-states in Section 3.1.2, we do not repeat this approach here and restrict ourselves in this section to simple states and AND-states (except that we allow states within an AND-state to be AND-states again) in order to stress the advantages of parallel graph transformation systems. The simulation specification for Statecharts with AND-states could easily be united with the sequential simulation rules for Statecharts with complex OR-states from Section 3.1.2, thus yielding a simulation specification for Statecharts with AND-states *and* complex OR-states.

The basic Statechart language (defined already in Section 2.3.2.2, consisting of symbols for states and transitions) is extended now to a simulation language. Fig. 3.21 shows the abstract syntax of the type graph for the simulation language for Statecharts with AND-states. For simulation, an Object is needed which is linked to the currently active states by the cs relationship (for current state). The Object receives events, modeled by an event queue linked to it by the relationships receives and next; the last event is a special one (labelled as "none"), depicting the queue's end.

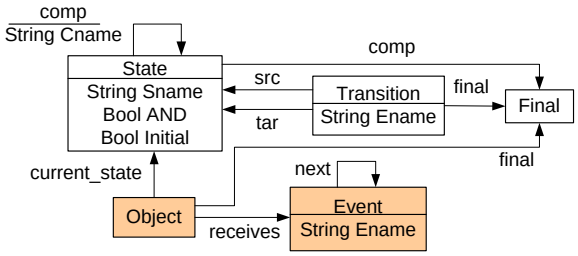


Figure 3.21: Type Graph TG_{SCsim} for Statecharts with AND-States

Example: An ATM Model

Fig. 3.22 shows a Statechart with an AND-state. We model an ATM (automated teller machine) where the user can insert a bank card and, after the input of the correct PIN, can draw a specified amount of cash from her or his bank account. The display component of the AND-state shows what is being displayed on screen, and, simultaneously, the card-slot component models whether the card slot is holding a bank card or not. The enter event triggers the transition before the AND-state to enter the AND-state. The card-sensed event happens if the sensor has sensed a user's bank card being put into the card slot. This event triggers two transitions in parallel. The next events (pin-input, pin-ok and amount-input) are local to the display Statechart. The end event again triggers two transitions if the current state is any but the welcome state for the display component and holding for the card-slot component. Then, the final states are reached and the AND-state can be left if the leave event happens.

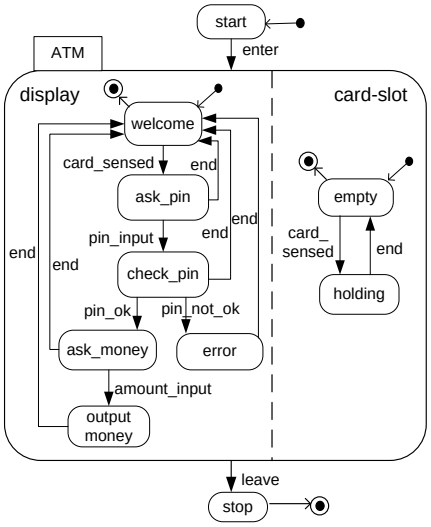


Figure 3.22: A Statechart Modeling an ATM

Fig. 3.23 shows the abstract syntax of the ATM model depicted in Fig. 3.22 as graph

typed over the type graph shown in Fig. 3.21. The initial state where we want to start the simulation is the start state before the AND-state is entered. Hence, the Current object points to the start state and is linked to an initial event queue consisting so far of the single event enter (the event needed to enter the AND-state) followed by the special event denoting the end of the queue. During the simulation, a rule can be applied which adds events to the event queue such that the queue holds the events that should be processed during the simulation. Note that we use a special shortcut notation to show the values of state attributes: We write a names which are not empty in quotation marks and put the name of a boolean attribute (AND, Initial) if its value is true.

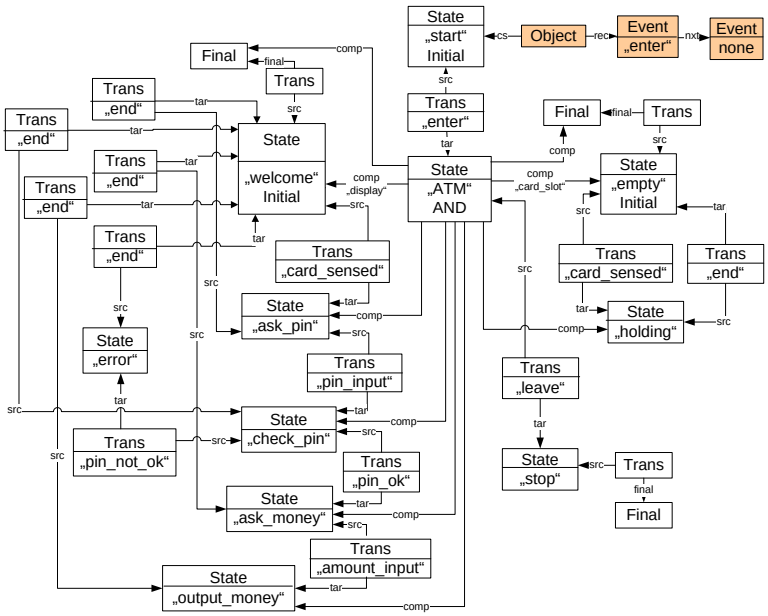


Figure 3.23: Abstract Syntax of the ATM Statechart in the start state

The Virtual Simulation Specification for the ATM Model

The Virtual Simulation Specification consists of the start graph (the virtual model) and a set of interaction schemes for simulation. We first consider the interaction schemes and then construct the virtual model according to the algorithm given in Def. 3.3.2.

Interaction Schemes of the Virtual Simulation Specification

In our Statecharts variant, every component Statechart belonging to an AND-state has exactly one initial state and at least one final state (such syntactical constraints must be guaranteed by the syntax grammar of the VL). The intended semantics for our Statecharts requires that if an AND-state is reached, the active states within the components are the

initial ones. A transition is processed if its pre-state is active and its triggering event is the same as the event which is received by the Object (the first event in the queue). Afterwards, the state(s) following the transition become(s) active, the event of the processed transition is removed from the queue, and the previously active state(s) (the pre-state(s) of the transition) is/are not active anymore. An active final state can be left again if a state connected to the final state by a final edge, has another outgoing transition whose triggering event equals the first event in the queue. Thus, final states can be reached more than once before an AND-state is left. More than one transition are processed simultaneously if they belong to different components of the same AND-state, if their pre-states are all active and if they are all triggered by the same event which is received by the Object. All component Statecharts belonging to the same AND-state must have reached a final state before the AND-state can be left and the transition from the AND-state to the next state can be processed.

For our simulator we use the Object not only as object which receives the next event (and is linked to the event queue) but also as pointer to the current active states. Thus, our simulation rules model the relinking of the Object to the next active states and the updating of the event queue. The parallel graph transformation system for the simulation of Statecharts with parallel states consists of three interaction schemes and three simple rules.

Rule addEvent(e) allows to add a new event of name e into the event queue. In this way, the events that should be processed during a simulation run, can be defined in the beginning of the simulation. Moreover, events also can be inserted at the end of the queue while a simulation is running.

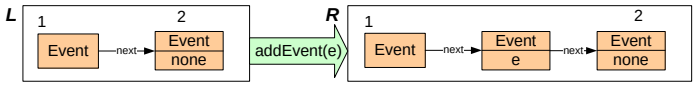


Figure 3.24: Rule addEvent(e) to insert Event e into the Event Queue

Fig. 3.25 shows the interaction scheme enterAND which moves the Object's current-state pointer along a transition that connects a state to an AND state. In this case, the pointer has not only to point to the AND state after the rule application, but also to all initial states of all parallel component Statecharts of the AND state. Hence, the amalgamated rule consists of as many copies of the elementary rule enterComponent as there are component Statecharts in the AND state (provided that each component has exactly one initial state which has to be ensured by a suitable syntax grammar).

Vice versa, when an AND state is left, the pointer has to be removed from all inner states of the AND state. This step is realized by the interaction scheme leaveAND in Fig. 3.26. The fact that the inner states all have to be Final is modeled by the NAC which says that the scheme is only applicable if there are no simple inner states. The elementary rule models how all inner links from the Current pointer to the component's final states are removed.

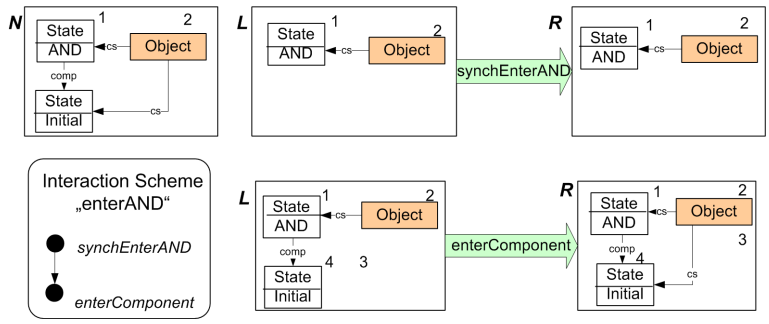


Figure 3.25: Interaction Scheme enterAND

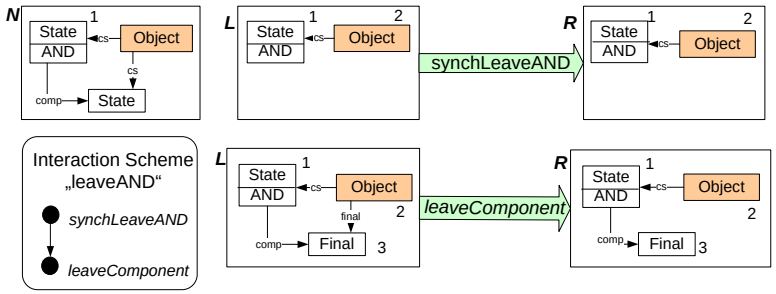


Figure 3.26: Interaction Scheme leaveAND

A simultaneous transition is modeled by the interaction scheme SimultaneousTrans in Fig. 3.27. In this scheme, an arbitrary number of transitions in different components of an AND-state are processed if they are triggered by the same event. In our ATM example this can happen at different points of the simulation: When the AND-state is entered and the event card-sensed is happening, then the two first transitions of the two parallel component Statecharts are processed simultaneously. Similarly, at any state of the display the user can abort the transaction: the end event triggers the return of the display component Statechart to the state welcome and the return of the card-slot component Statechart to the state empty.

The SimultaneousTrans interaction scheme is a good example for a concise way to model simultaneous transitions which are triggered by a single event. This would be quite difficult to model using simple rules. Note that this scheme is applicable also for sequential transition processing within an AND-state. Then there is only one copy of the elementary rule.

In order to enter and to leave final states, two simple rules are necessary. If the simulation is not hand-triggered, a suitable control mechanism has to take care that these two rules are not applied directly after each other in a never-ending loop (see the example for simulating Statecharts with OR-states in Section 3.1, where intermediate edge types are introduced to prevent this situation).

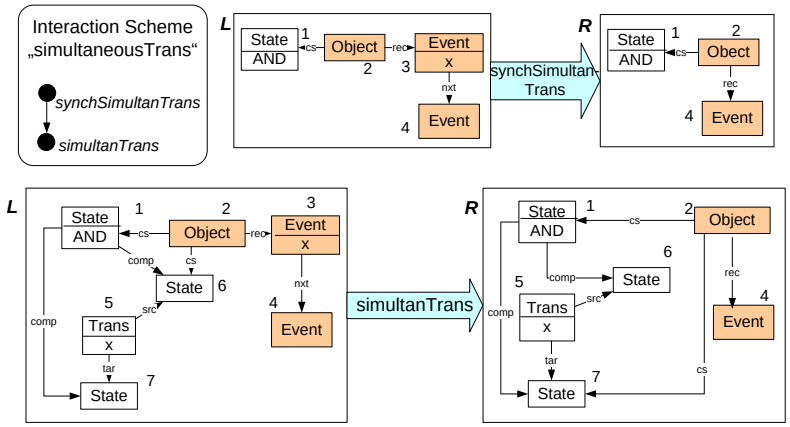


Figure 3.27: Interaction Scheme SimultaneousTrans

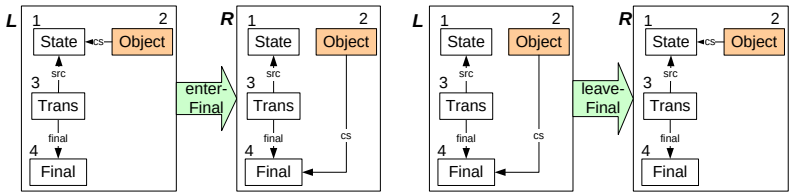


Figure 3.28: Rules for Entering and Leaving Final States

The next simple rule works similar to the SimultaneousTrans scheme but processes sequential transitions outside of AND-states. This rule is applied if the rule scheme for simultaneous transition is not applicable in the current system state. Rule sequentialTransition processes a transition in the active state which is triggered by the current event.

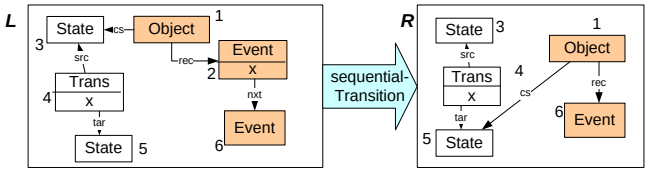


Figure 3.29: Rule SequentialTransition

The order and control of rule and scheme applications for automatic simulation can be given by priorities. The scheme enterAND and the rule enterFinal have the highest priority (1) because it must be checked after each transition step if an AND-state or a final state has been reached which must be entered. At priority level (2) is the rule scheme leaveAND. This means, if the current AND-state has been entered and all final states within its components have been reached, the AND-state is finished and may be left. Priority (3) have the rule scheme SimultaneousTrans and the rule leaveFinal. This means, if enterAND enterFinal and leaveAND are not applicable in the current system state, then a simultaneous transition or a single transition within the current AND-state may be processed (if possible),

or a final state is left which may lead to a transition within the same component being processed immediately afterwards. Note that in this way it is possible to have more than one cycle in one or more components of the AND-state even if the inner final states have been reached in between. The lowest priority (4) has the rule SequentialTransition because a sequential transition step outside of an AND-state must be processed only in the case that there are no simultaneous transitions which can be processed in one parallel step.

Virtual Model of the Virtual Simulation Specification

Initially, the virtual model VM is the model G, i.e. the abstract syntax graph of the model without the simulation symbols (Fig. 3.23 without the colored nodes). We now extend VM by matching the restrictions of the left-hand sides of the simulation rules and interaction schemes to the type graph of the modeling VL, $L|_{TG}$, to VM and by adding the missing simulation symbols to VM until each match $m : L \rightarrow VM$ is complete.

Trying to match the simple rule addEvent(e) (Fig. 3.24), we find that there is no match from L_{TG} to VM. In this case, nothing is added to VM, and we try the next rule or rule scheme, the interaction scheme SC-Scheme-enterAnd. Here, we find two different matches from the left-hand side of the elementary rule enterComponent into VM. Both matches map the AND-state to the state called ATM. In the first match, the initial state from the rule is mapped to the state called welcome, and in the second match to the state called empty. The left-hand side of the rule contains an additional Object node which is linked to the AND-state. Thus, an Object node is generated in VM. As both matches are glued at the AND-state and the Object node linked to it, only one Object node is generated.

Next, we consider the interaction scheme leaveAND (Fig. 3.26). We find two matches from the left-hand side of the elementary rule leaveComponent into VM. Again, both matches map the AND-state to the ATM state. In the first match m_1, the Final state is mapped to the Final state connected to the welcome state, in the other match m_2 to the Final state connected to the empty state. As both matches are glued at the AND-state and at the Object node linked to it, one Object node linked to the AND-state is generated in VM, and two links of type final which link the Object node to the two Final states in $m_1(L)$ and $m_2(L)$.

Considering the interaction scheme Simultaneous Trans (Fig. 3.27), we find a lot of matches from the elementary rule simultanTrans into VM, namely into all subgraphs which represent the environement of a transition. As all rule instances are glued over subrule synchSimultanTrans at their common AND-state, and an Object linked to it, again only one Object node is generated in VM and linked to the ATM AND-state. Moreover, for each transition in one of the matches, an Event node attributed with the same name as the transition, is generated and linked to the Object node. Another Event node is linked to it via a next edge. If two or more transitions are triggered by the same event, than only one

new Event node is generated, as all instances of the same event are glued via the subrule. Moreover, the Object node is linked by a current-state edge to all of the transitions' source states.

The next two rules enterFinal and leaveFinal lead to the addition of Object nodes linked to all states which are connected to a Final state, and to the addition of Object nodes linked to all Final states. Finally, the rule SequentialTransition has a similar effect as the rule scheme Simultaneous Trans: Object nodes are linked to all source states of all transitions in VM (this time also transitions outside of AND-states are matched), and Event nodes attributed by the names of the transitions' triggering events are linked to the Object node. The Event nodes themselves are linked by next links to unattributed Event nodes. Note that if nodes without attribute values are added to VM, then these attribute values are represented by variables of the respective attribute types.

Fig. 3.30 shows the resulting virtual model for the ATM model in Fig. 3.23. It can well be seen how all source states of "end"-triggered transitions in the left AND-component are combined by pointers from the respective object nodes with the source state holding of the "end"-triggered transition in the right AND-component, thus enabling the pairwise parallel processing of transitions triggered by the same event.

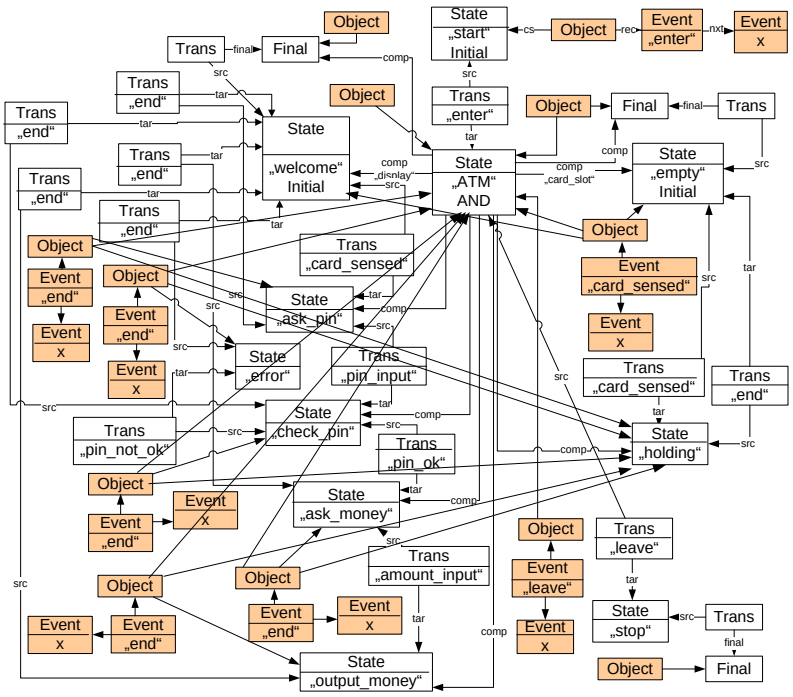


Figure 3.30: Virtual Model of the ATM Statechart

The Simulation Specification for the ATM Model

The type graph for the simulation specification is TG_{SCsim} (see Fig. 3.21), the type graph of the simulation alphabet. The set of amalgamated simulation rules comprising the simulation specification is constructed by amalgamating elementary rules at their subrules according to the interaction schemes of the virtual simulation specification, where the matches are defined over the virtual model, the start graph of the virtual simulation specification.

Fig. 3.31 shows two of the simulation rules constructed in this way. Rule enterAND is obtained as two matches from the elementary rule enterComponent (Fig. 3.25) into the virtual model are found, which differ only in the respective initial states. Hence, two instances of rule enterComponent are generated and amalgamated at their common nodes according to subrule synchEnterAND. Analogously, Rule leaveAND is constructed by amalgamating two instances of the elementary rule leaveComponent (Fig. 3.26) matched to the two final states within the AND-state in the virtual model, at their common subrule synchLeaveAND.

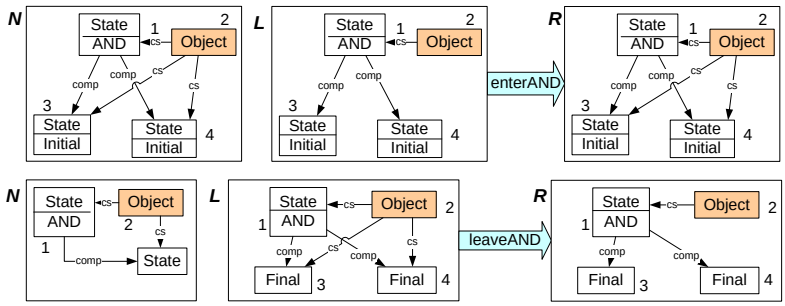


Figure 3.31: Simulation Rules for Entering and Leaving AND-states

Fig. 3.32 shows a simulation rule for processing two transitions simultaneously. The rule is constructed by amalgamating two instances of the elementary rule simultanTrans from Fig. 3.27 matched to two states which are enabled by the same event in parallel.

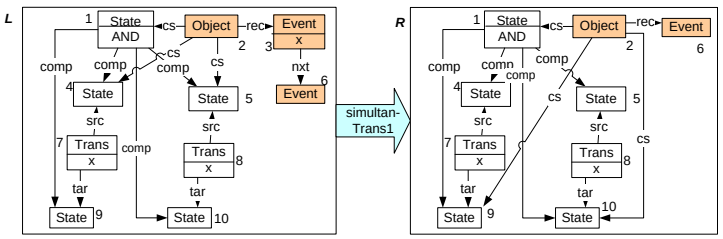


Figure 3.32: Simulation Rule for Processing Simultaneous Transitions

From the same interaction scheme, isomorphic amalgamated rules are obtained for all pairs of transitions enabled in parallel by an “end” event. Simple rules like rule sequentialTransition which are not interaction schemes, are formally “amalgamated” over the empty

subrule, which means that they remain unchanged and are simply taken over into the set of simulation rules.

The priorities defined for the interaction schemes are passed over to all of their amalgamated rules. Thus, applying the simulation rules to the model in Fig. 3.23 in a way that respects the rule priorities, we get valid simulation runs for the ATM model.

3.4 From Universal to Model-Specific Simulation Specifications

The advantage of using interaction schemes in simulation specifications is that interaction schemes can be defined independently of a concrete model (no model transformation has to be defined). Moreover, using rule schemes, finite simulation specifications may be defined for models where simulation specifications for the complete VL would contain an infinite set of simulation rules (e.g. for the VLs of Petri nets or Statecharts with AND-states). A slight disadvantage of the use of rule schemes (for the purpose of serving as a basis for animation) is that the amalgamated rules in the simulation specification are still not completely model-specific, e.g. they do not contain model-specific information like place or event names.

Hence, in this section we show how to translate simulation specifications containing universal simulation rules according to Def. 3.1.2 and Def. 3.3.4 to model-specific simulation specifications, according to Def. 3.2.2 .

For this translation we need again the virtual model of a model G. (Def. 3.3.2). By the help of this virtual model, all universal simulation rules can be instantiated by finding all possible matches into the virtual model.

We argue that the resulting model-specific set of simulation rules defines the same behavioral semantics as the original set of universal simulation rules.

Definition 3.4.1 (*Translating Universal to Model-Specific Simulation Rules*)
Let G be a model, typed over TG_A, the abstract alphabet of the visual language VL, and let VM be its virtual model according to Def. 3.3.2, typed over TG_{sim} which extends TG_A. Let $SimSpec(VL) = (TG_{sim}, P_{sim})$ be a simulation specification for VL, according to Def. 3.1.2, and $ASimSpec(VM) = (TG_{sim}, P_{amalg})$ an amalgamated simulation specification for G according to Def. 3.3.4.

A *model-specific simulation specification* for G constructed from $SimSpec(VL)$ $[ASimSpec(VM)]$ is given by $SimSpec_{VL_S} = (TG_{sim}, P_{sim})$, with the simulation type graph TG_{sim} and a set of model-specific simulation rules P_{sim}. The rules in P_{sim} are constructed in two steps: First, each universal simulation rule $p \in P_{sim}[P_{amalg}]$ is applied to the virtual model VM at each of its matches $m_i : L \rightarrow VM$. Each application of r at match m_i leads

to a span $VM \leftarrow VM_I \rightarrow VM'$. Second, this span is restricted to the codomain of match m_i. The result is the span $VM|_{codom(m_i)} \leftarrow VM_I|_{codom(i)} \rightarrow VM'|_{codom(m_i^*)}$, called *rule instance* of r. The set P_{sim} contains all rule instances which can be constructed from the universal simulation rules at all possible matches to VM.

The behavioral semantics $CSem(G)$, is the GTS-compiler semantics defined by the model-specific simulation specification $SimSpec_{VL_S}$ according to Def. 3.2.3, i.e. $CSem(G)$ is the set of all graph transformations which can be derived by applying the model-specific simulation rules in P_{sim}. △

Theorem 3.4.2 (*Semantical Compatibility of* $ISem(G)$, $ASem(G)$ *and* $CSem(G)$)
Let $SimSpec(VL) = (VL_{sim}, P_{sim})$ be a simulation specification for VL defining the semantics $ISem(G)$ for a model $G \in VL$. Let $ASimSpec(VM) = (TG_{sim}, P_{amalg})$ be an amalgamated simulation specification for G, defining the semantics $ASem(G)$, and $SimSpec(G)$ the model-specific simulation specification for G constructed from $SimSpec(VL)$ $[ASimSpec(VM)]$ according to Def. 3.4.1, defining the semantics $CSem(G)$.

Then, $ISem(G)$, $ASem(G)$ and $CSem(G)$ are equivalent. △

Proof: A simulation rule instance of rule $r \in P_{sim}[P_{amalg}]$, constructed at match m can be applied only at match m to G, and leads obviously to the same derived graph G' as the application of the universal rule r at match m. Thus, whenever a universal simulation rule r is applicable at a match m to a graph corresponding to a state of the model G, then the respective rule instance is also applicable and leads to the same derived graph G'. Hence, the behavioral semantics $ISem(G)[ASem(G)]$, defined as the set of transformations of G over universal simulation rules, is equivalent to the behavioral semantics $CSem(G)$, defined for the model-specific simulation specification $SimSpec(G)$, where the model-specific simulation rules in $P_{sim}(G)$ are directly given as rule instances of the universal simulation rules. □

As example, we present the translation from a simulation specification with universal simulation rules for the Statechart VL to a model-specific simulation specification for the Statechart example modeling the radio clock (see Section 3.1.2).

3.4.0.1 From the Simulation Specification for Statecharts with OR-states to a Model-Specific Simulation Specification for the Radio Clock Statechart

The simulation type graph TG_{sim} for the radio clock Statechart in Section 3.1.2 is the type graph for Statecharts with OR-states, enriched by simulation symbols (see Fig. 3.1).

For the example of the radio clock Statechart, the model G corresponds to the simulation model in Fig. 3.3 without the colored nodes and their adjacent arcs. We now construct the virtual model VM from G finding matches from the universal simulation rules in Fig. 3.4

and 3.5. Initially, VM equals G. The initial simulation rules initial and addEvent(e) do not add symbols to VM. Rule initial does not contain simulation symbols in the left-hand side, and for rule addEvent(e) no match can be found from $L|_{TG}$ into VM, as $L|_{TG}$ is empty.

For rule down, two matches can be found, as our radio clock Statechart contains two initial states. This results in the addition of two Object nodes to VM which are linked by current edges to the state called Time, and to the state called Display, respectively. The matches found for rule trans result in the addition of an Object node for each transition. The Object node is linked to the transition's source state and receives an event of the name of the event triggering the transition. The received event is linked by a next edge to another new event (with a variable for its name, as this is not fixed in the rule).

The rules up1 and up2 can be matched to all states which have a super state. Two Object nodes are linked to all the substates, one by a current link and one by an intermediate link.

Finally, rule transAfterUp matches to the same occurences as rule trans. For rule transAfterUp, the added Object nodes are linked to the transition's source state by an intermediate edge. Events are linked to the [Object like for rule trans.

We do not draw the virtual model for the radio Statechart here, as the construction has been shown already in detail for the ATM Statechart in Section 3.3.2, (see Fig. 3.30) and the extension of the model in Fig. 3.3 works analogously here.

The set of model-specific simulation rules comprising the simulation specification is the set of all rule instances of the universal simulation rules for all possible matches into the virtual model VM.

Fig. 3.33 shows the two model-specific rules down1 and down2 which are all rule instances of rule down as there are exactly two matches from the left-hand side of rule down into the virtual model VM.

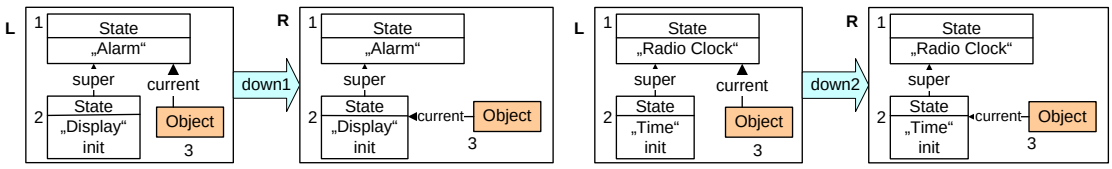


Figure 3.33: Model-Specific Simulation Rules for Entering an OR-state

Model-specific simulation rules for processing transitions are obtained as various matches from the universal rules trans and transAfterUp into the virtual model are found. Two of these rule instances are depicted in Fig. 3.34.

Moreover, for each state with a super state we get one instance of rule up1, and one instance of rule up2 (not depicted).

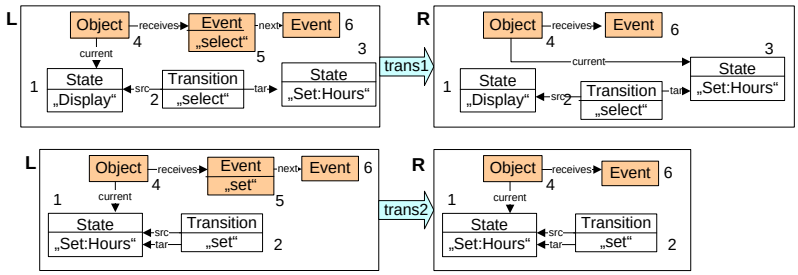


Figure 3.34: Some Model-Specific Simulation Rules for Processing Transitions

3.5 Applications

3.5.1 GTS-Compiler Semantics for Algebraic High-Level Nets

In this section, we review the definition of AHL nets and their behavior, and present our running example, the specification of the well-known *Dining Philosophers* as AHL net.

The version of AHL-nets defined in this section corresponds to [Ehr04b]. Places are typed, that is, the data elements on these places and the terms occurring in the inscriptions of the attached arcs are required to be of a specified sort. This typing reduces the marking graph considerably.

Definition 3.5.1 (*Algebraic High-Level Net*)
An *algebraic high-level net* (*AHL-net*) $N = (SPEC, P, T, pre, post, cond, type, A)$ consists of an algebraic specification $SPEC = (S, OP, E; X)$, sets P and T of places and transitions respectively, pre- and post-domain functions $pre, post : T \to (T_{OP}(X) \otimes P)^{\oplus}$ assigning to each transition $t \in T$ the pre- and post-domains $pre(t)$ and $post(t)$ (see first Remark), respectively, a firing condition function $cond : T \to \mathcal{P}_{fin}(EQNS(S, OP, X))$ assigning to each transition $t \in T$ a finite set $cond(t)$ of equations over the signature (S, OP) with variables X, a type function $type : P \to S$ assigning to each place $p \in P$ a sort $type(p) \in S$, and an (S, OP, E)-algebra A (see [EM85]). △

Remarks:

1. Denoting by $T_{OP}(X)$, or more precisely $T_{(S,OP)}(X)$ the set of terms with variables X over the signature (S, OP) (see [EM85]), and by $M^{\oplus}$ the free commutative monoid over a set M, the set of all type-consistent arc inscriptions $T_{OP}(X) \otimes P$ is defined by $T_{OP}(X) \otimes P = \{(term, p) | term \in T_{OP}(X)_{type(p)}, p \in P\}$.

   Thus, $pre(t)$ (and similar $post(t)$) is of the form $pre(t) = \sum_{i=1}^{n}(term_i, p_i) \quad (n \geq 0)$ with $p_i \in P, term_i \in T_{OP}(X)_{type(p_i)}$. This means, $\{p_1, \ldots, p_n\}$ is the pre-domain of t with arc-inscription $term_i$ for the arc from p_i to t if the $p_1, \ldots, p_n$ are pairwise

distinct (unary case) and arc-inscription $term_{i1} \oplus \cdots \oplus term_{ik}$ for $p_{i1} = \cdots = p_{ik}$ (multi case). In our sample AHL net (see Example 3.5.3) we have the multi case.

2. As places are typed, a marking m is an element $m \in (A \otimes P)^{\oplus}$ with $A \otimes P = \{(a,p) | a \in A_{type(p)}, p \in P\}$.

□

Enabling and firing of transitions are defined as follows.

Definition 3.5.2 (*Firing Behavior of AHL Nets*)
Given an AHL-net as above and a transition $t \in T$, $Var(t)$ denotes the set of variables occurring in $pre(t), post(t)$, and $cond(t)$. An assignment $asg_A : Var(t) \to A$ is called consistent if the equations $cond(t)$ are satisfied in A under asg_A.

The marking $pre_A(t, asg_A)$ – and similarly $post_A(t, asg_A)$ – is defined for $pre(t) = \sum_{i=1}^{n}(term_i, p_i)$ by $pre_A(t, asg_A) = \sum_{i=1}^{n}(\overline{asg_A}(term_i), p_i)$, where $\overline{asg_A} : T_{OP}(Var(t)) \to A$ is the extension of the assignment asg_A to an evaluation of terms (see [EM85]).

A transition $t \in T$ is enabled under a consistent assignment $asg_A : Var(t) \to A$ and marking $m \in (A \otimes P)^{\oplus}$, if $pre_A(t, asg_A) \leq m$. In this case, the successor marking m' is defined by $m' = m \ominus pre_A(t, asg_A) \oplus post_A(t, asg_A)$ and gives raise to a *firing step* $m[t, asg_A\rangle m'$.

△

Example 3.5.3 (*The Dining Philosophers as AHL Net*)

As example we show the AHL net for *The Dining Philosophers* in Fig. 3.35 (see [Rei85, PER95] for the corresponding place/transition net). We identify the five philosophers as well as their chopsticks by numbers. Fig. 3.35 (a) shows the initial situation where all philosophers are thinking and all chopsticks are lying on the table. Fig. 3.35 (b) shows the AHL net with the corresponding initial marking. For this marking, the transition take is enabled as a thinking philosopher and his left and right hand side chopsticks are available. The firing of transition take with the variable binding $p = 2$, for example, removes token 2 from place thinking and adds it to place eating, whereas tokens 2 and 3 are removed from place table, as the chopstick computing operation (p mod 5) +1 is evaluated to 3.

As datatype specification we take a basic specification for all AHL nets $SPEC_{BASIS}$ consisting of the union of specifications NAT for natural numbers, $BOOL$ for boolean operations, and $STRING$ for strings. The tokens on all places are elements of a corresponding $SPEC_{BASIS}$-algebra, i.e. natural numbers in our example. The arcs are inscribed each by one variable or term from $T_{OP}(X)$ denoting computation operations to be

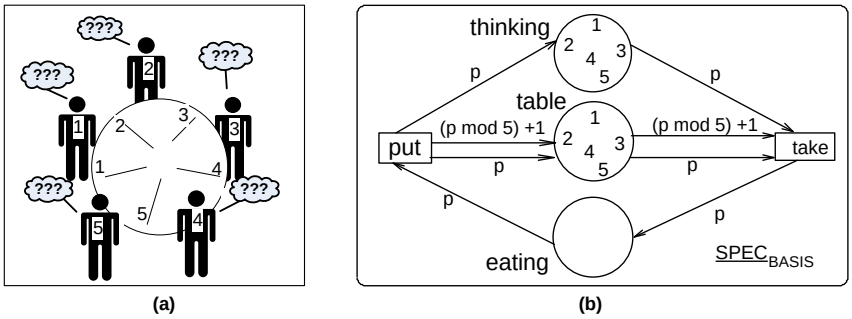


Figure 3.35: The *Dining Philosophers* (a) modeled as AHL Net (b)

executed on token values. Note that we allow more than one arc from a place to a transition (and vice versa), such that the set of terms $pre(t)(p)$ may contain more than one term.

△

Translating AHL Nets into Typed Attributed Graph Transformation Systems

The translation of AHL nets to attributed graph transformation systems generalizes that of P/T nets into graph transformation systems as proposed in the literature [CM95, Kre81] and reviews in a slightly modified form the concepts and results in [BEP02a].

An AHL net N, is translated to an attributed graph transformation system $SimSpec(N) = (TG_{AHL}, P_{AHL})$ (the simulation specification of N), where $\mathsf{TG}_{AHLsimu}$ is the type graph for marked AHL nets (see Fig. 3.36), extending the type graph for unmarked AHL nets (see Def. 2.3.1 in Section 2.3.1.1).

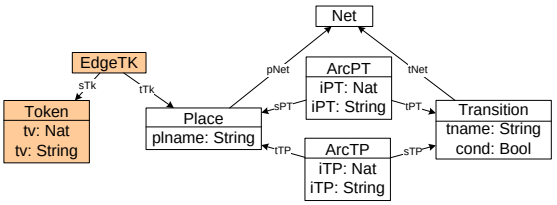


Figure 3.36: Simulation Alphabet TG_{AHL} for AHL Nets

The set P_{AHL} contains the simulation rules p_t, one for each transition $t \in T$. The left- and right-hand sides of each simulation rule L and R contain the transition's pre- and postdomain, and the rule application condition corresponds to the firing condition of t. Moreover, we define a translation from the initial marking of N to a start graph G (the visual model) typed over TG_{AHL}.

In a first step, we define the translation of a marked AHL net into an attributed graph which serves as start graph for the simulation (Def. 3.5.4). In the second step, we define

the translation of the AHL net firing behavior, given by the transitions and their environments, into a simulation specification according to the compiler approach (Def. 3.5.6). On the basis of this formalization of a model transformation from AHL nets to graph transformation systems we prove the semantical compatibility of AHL nets and their translation (Theorem 3.5.10).

Definition 3.5.4 (*Translation of a Marked AHL Net to an Attributed Graph*)
Given an AHL net $N = (SPEC, P, T, pre, post, cond, A)$ with marking $m \in (A \times P)^{\oplus}$. The translation Tr of (N, m) is given by the function $Tr : (AHLnet, (A \times P)^{\oplus}) \rightarrow \mathbf{Graphs_{TG_{AHL}}}$ from the set of pairs of AHL nets plus markings to the set of graphs typed over TG_{AHL} (Fig. 3.36) with

$$\begin{aligned} Tr(N,m) = G = (&G_{Net}, G_{Place}, G_{Trans}, G_{Token}, G_{EdgeTk}, G_{ArcPT}, G_{ArcTP},\\ &op_{sPT}, op_{tPT}, op_{sTP}, op_{tTP}, op_{sTk}, op_{tTk},\\ &attr_{tv}, attr_{iPT}, attr_{iTP}, attr_{cond}), \qquad \text{where} \end{aligned}$$

$G_{DSIG} = T_{OP}(X) \uplus A$ (disjoint union of the term algebra with variables over TG_{AHL} and A),

$G_{Net} = \{n\}$ (one net node),

$G_{Place} = P$ (the place nodes), $G_{Trans} = T$ (the transition nodes),

$G_{Token} = \{tk | tk = (a, p, i) \in \widetilde{m}\}$. The multiset $m \in (A \times P)^{\oplus}$ is given by the set $\widetilde{m} = \{(a, p, i) \in A \times P \times I\!N \mid 0 < i \leq m(a, p)\}$, where multiple occurrences of the same element in m are numbered by i in $\widetilde{m}$,

$G_{EdgeTk} = \{e_{tk} | tk \in G_{Token}\}$,

$G_{ArcPT} = \{arcPT | arcPT = (term, p, i) \in PreSet\}$,

$G_{ArcTP} = \{arcTP | arcTP = (term, p, i) \in PostSet\}$, where the multisets of terms in arc inscriptions are given by the sets $PreSet = \cup_{t \in T} PreSet_t$ and $PostSet = \cup_{t \in T} PostSet_t$ where $PreSet_t = \{(term, p, i) | pre(t)(term, p) \geq i > 0\}$ corresponds to $pre(t)$ and, analogously, $PostSet_t$ to $post(t)$.

$pNet : G_{Place} \rightarrow G_{Net}$ with $pNet(p) = n$,

$tNet : G_{Transition} \rightarrow G_{Net}$ with $tNet(t) = n$,

$op_{sPT} : G_{ArcPT} \rightarrow G_{Place}$ with $op_{sPT}(term, p, i) = p \; \forall (term, p, i) \in G_{ArcPT}$,

$op_{tPT} : G_{ArcPT} \rightarrow G_{Trans}$ with $op_{tPT}(term, p, i) = t$, if $(term, p, i) \in PreSet_t$, $\forall (term, p, i) \in G_{ArcPT}$,

$op_{sTP} : G_{ArcTP} \rightarrow G_{Place}, op_{tTP} : G_{ArcTP} \rightarrow G_{Trans}$: analogously,

$op_{sTk} : G_{EdgeTk} \rightarrow G_{Token}$ with $op_{sTk}(e_{(a,p,i)}) = (a, p, i)\ \forall e_{(a,p,i)} \in G_{EdgeTk}$,

$op_{tTk} : G_{EdgeTk} \rightarrow G_{Place}$ with $op_{tTk}(e_{(a,p,i)}) = p\ \forall e_{(a,p,i)} \in G_{EdgeTk}$,

$attr_{tv} : G_{Token} \rightarrow I\!N$ with $attr_{tv}((a, p, i)) = a\ \forall (a, p, i) \in G_{TV}$,

$attr_{iPT} : G_{ArcPT} \rightarrow T_{OP}(X)$ with $attr_{iPT}((term, p, i)) = term\ \forall (term, p, i) \in G_{ArcPT}$, $attr_{iTP} : G_{ArcTP} \rightarrow T_{OP}(X)$: analogously,

$attr_{cond} : G_{Trans} \rightarrow \mathcal{P}_{fin}(EQNS(X))$ with $attr_{cond}(t) = cond(t)\ \forall t \in G_{Trans}$

The typed attributed graph $Tr(N, m)$ is the simulation model corresponding to AHL net N with initial marking m.

△

Example 3.5.5 (*AHL net* Dining Philosophers *translated to an attributed graph*)
Fig. 3.37 shows the attributed graph resulting from the translation of the initially marked AHL net presented in Fig. 3.35 (b).

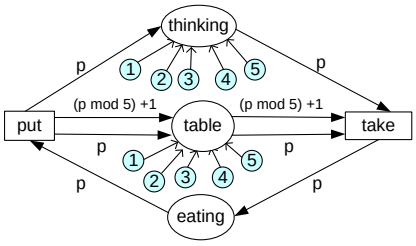


Figure 3.37: Translation of *Dining Philosophers* Net with Initial Marking

We visualize Place nodes as ellipses, Transition nodes as rectangles, and Token nodes as coloured circles containing the token value attributes. Token nodes are connected to their places by EdgeTk arcs. ArcPT and ArcTP symbols are drawn as edges which are attributed by the arc inscription terms. In this example we have no firing conditions. △

In addition to the statical structure of the AHL net and the net marking, we now define the translation of a net's firing behavior into a set of graph rules P_{sim}, the simulation rules. Each simulation rule encorporates the firing behavior of one transition: the left-hand side contains its predomain, the right-hand side its postdomain. A possible firing condition $cond(t)$ is translated to the attribute condition of the simulation rule for transition t.

Definition 3.5.6 (*Simulation Specification for AHL Nets*)
Let $N = (SPEC, P, T, pre, post, cond)$ be an AHL net. We translate the firing behavior of N to a set of simulation rules $P_{sim} = \{p_t = (L_t \overset{l_t}{\leftarrow} I_t \overset{r_t}{\rightarrow} R_t) | t \in T\}$ where for each transition $t \in T$ the rule components L_t, I_t and R_t are attributed graphs over TG_{AHL} (Def. 2.3.1), defined as follows:

The interface I_t contains only nodes of sort Place (the environment of transition t) and no operations. All sorts and operations in L_t and R_t are empty, except Place, Token, EdgeTk and the adjacent operations:

- $L_{Place} = I_{Place} = R_{Place} = \{p | p \in pre(t) \cup post(t)\}$
- $L_{Token}[R_{Token}] = \{tk | tk = (term, p, i) \in PreSet_t[PostSet_t]\}$
- $L_{EdgeTk} = \{e_{tk} | tk \in L_{Token}\}$,

- $op^L_{sTK} : L_{EdgeTk} \rightarrow L_{Token}$ with $op^L_{sTk}(e_{(term,p,i)}) = (term, p, i)$,
- $op^L_{tTK} : L_{EdgeTk} \rightarrow L_{Place}$ with $op^L_{tTk}(e_{(term,p,i)}) = p$,
- $attr^L_{tv} : L_{Token} \rightarrow T_{OP}(X)$ with $attr_{tv}((term, p, i)) = term$
  (analogously for $R_{EdgeTk}, op^R_{sTK}, op^R_{tTK}$ and $attr^R_{tv}$)

The rule morphisms $L_t \overset{l_t}{\leftarrow} I_t$ and $I_t \overset{r_t}{\rightarrow} R_t$ are given by (p_{Place}, p_{Trans}, p_{Token}, p_{ArcPT}, p_{ArcTP}, p_{EdgeTk}) = ($id_{Place}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset$). Then, p_t is the conditional rule corresponding to the firing behavior of transition t. The typed attributed graph transformation system (TG_{AHL}, P_{sim}) is called the *simulation specification* $SimSpec(N)$ for AHL net N. △

Remarks: Both L_t and R_t contain only the places of the transition's environment and tokens connected to these places, where the tokens are attributed by terms of $T_{OP}(X)$. The difference between L_t and R_t is that L_t corresponds to $pre(t)$ whereas R_t corresponds to $post(t)$. The Token symbols are not in the interface I_t as the rule models the deletion of tokens from the predomain (L_t) and the addition of tokens to the postdomain (R_t).

□

Example 3.5.7 (*Simulation Specification for the* Dining Philosophers)
Let N be our AHL net as shown in Fig.3.35 (b). The simulation specification for our AHL net is given by $SimSpec(N) = (\mathsf{TG}_{AHL}, P_{simDiPhi})$ with $P_{simDiPhi}$ being the set of two simulation rules constructed according to Def. 3.5.6. These simulation rules are shown in Fig. 3.38. Note that place nodes are preserved by the rule mapping (equal numbers for an object in L and R means that this object is contained in the interface I), and token nodes are deleted (predomain tokens) or generated (postdomain tokens).

△

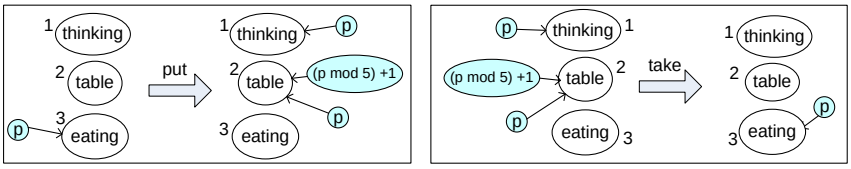


Figure 3.38: Simulation Rules for the AHL net *Dining Philosophers*

The behavior of a model is simulated by applying the simulation rules from P_{sim} to the simulation model $Tr(N, m)$ and to the sequentially derived graphs which correspond to different markings of N.

For the theorem stating the semantical compatibility of an AHL net and its simulation specification (Theorem 3.5.10), we need to translate an attributed graph $Tr(N, m)$, which is the translation of the marked AHL net (N, m), back to a marked AHL net N_{back}. In Lemma 3.5.9 we show that the marking of N_{back} is the same as the marking of the original net N.

Definition 3.5.8 (*Backward Translation of Graph $Tr(N, m)$ to a Marked AHL Net*) Given an AHL net $N = (P, T, pre, post, SPEC, cond, A)$ with marking $m \in (A \times P)^{\oplus}$ and its translation, the attributed graph $Tr(N, m)$, constructed as defined in Def. 3.5.4. Then the backward translation $Back$ of $Tr(N, m)$ is the marked AHL net N_{back} with:

$P_{back} = G_{Place}, T_{back} = G_{Trans}$,

$\forall t \in T : pre_{back}(t) = \sum_{j=1}^{|G_{ArcPT}|}(term_i, p_i)$ with $(term_j, p_j, i) \in G_{ArcPT}$,

$\forall t \in T : post_{back}(t) = \sum_{j=1}^{|G_{ArcPT}|}(term_i, p_i)$ with $(term_j, p_j, i) \in G_{ArcTP}$,

$\forall t \in T : cond_{back}(t) = attr_{cond}(t)$,

$m_{back} = \sum_{tk \in G_{Token}}(attr_{tv}(op_{sTk}(e_{tk}), op_{tTk}(e_{tk})))$

△

Lemma 3.5.9 (*Compatibility of Markings of N and $Back(Tr(N, m))$*)

Let $m \in (A \times P)^{\oplus}$ be a marking of an AHL net N, $G = Tr(N, m)$ be the translation of N with marking m according to Def. 3.5.4, and $Back(G)$ the backward translation as defined in Def. 3.5.8. Then, the marking of $Back(G)$ is the same as the marking of N, i.e. $m_{back} = m$. △

Proof:

$$\begin{aligned} m_{back} &= \sum_{e_{tk} \in G_{EdgeTk}} (attr_{tv}(op_{sTk}(e_{tk})), op_{tTk}(e_{tk})) \\ &= \sum_{e_{(a,p,i)} \in G_{EdgeTk}} (a,p) = \sum_{(a,p,i) \in G_{Token}} (a,p) = m \end{aligned}$$

□

Theorem 3.5.10 (*Semantical Compatibility of AHL net N and its Simulation Specification*)

The semantics of an AHL net N with initial marking m_{init} and the semantics of the simulation specification $SimSpec(N)$ are compatible, denoted by $Sem_{AHL}(N, m_{init}) \cong Sem_{AGT}(Tr(N, m_{init}), SimSpec(N))$, where the semantics of an AHL net is given by a set of firing sequences (firing steps), and the semantics of an attributed graph transformation system together with a start graph by a set of transformation sequences starting from the start graph $Tr(N, m_{init})$. Semantical compatibility means that for each firing step $m[t, asg\rangle m'$ and for $G = Tr(N, m)$ there is a transformation step $d : G \stackrel{p_t}{\Longrightarrow} H$ where p_t is the simulation rule corresponding to transition t, such that the marking m' is the same as the marking of the backward translated AHL net $Back(H)$. △

Proof: We show that

1. For each firing step $m[t, asg\rangle m'$ and for $G = Tr(N, m)$ there is a transformation step $d : G \stackrel{p_t}{\Longrightarrow} H$ where p_t is the simulation rule corresponding to transition t, such that the marking m' is the same as the marking of the backward translated AHL net $Back(H)$, i.e. the diagram to the right commutes.

$$\begin{array}{ccc} (N,m) & \stackrel{[t>}{\Longrightarrow} & (N,m') \\ {\scriptstyle Tr}\Big\Vert & = & \Big\Vert{\scriptstyle Tr} \\ G & \underset{p_t}{\Longrightarrow} & H \end{array}$$

2. Each firing sequence $\sigma \in Sem_{AHL}(N, m_{init})$ corresponds to a transformation sequence $\sigma' \in Sem_{AGT}(Tr^{AGT}(N, m_{init}))$. This means, for all firing sequences $\sigma_i = (m_i[t_i, asg_i\rangle m_i') \wedge 1 \leq i \leq n$ such that $m'_{i-1} = m_i$, we have $Tr(N, m'_{i-1}) = Tr(N, m_i)$ in the corresponding transformation sequence $Tr(N, m_i) \stackrel{r_{t_i}}{\Longrightarrow} Tr(N, m_i')$, for $1 \leq i \leq n$.

ad (1)

Let $m[t, asg\rangle m'$ be a firing step, $G = Tr(N, m)$ the translation of N, m to an attributed graph and $p_t : L_t \to R_t$ the simulation rule for transition t as defined in Def. 3.5.6.

We define the match $m_{asg} : L_t \to G$ as follows: m_{asg} is the identity on the sorts $G_{Place}, G_{Transition}, G_{ArcTP}$ and G_{ArcPT}. For tokens in G_{Token} we define $m_{asg}((term, p, i)) = (\overline{asg}_A(term), p, i)$ and for token edges in G_{EdgeTk} we define $m_{asg}(e_{(term,p,i)}) = e_{(\overline{asg}_A(term),p,i)}$. $m_{asg_A} : T_{OP}(X) \to T_{OP}(X) \cup A$ maps each term in $T_{OP}(X)$ to its extended assignment in A: $m_{asg_A}(term) = \overline{asg}_A(term)$.

We show that the match m_{asg} satisfies the graph morphism properties for the token edges $e_{(term,p,i)} \in G_{EdgeTk}$:
$m_{asg}(op_{sTk}(e_{(term,p,i)}))$ = $m_{asg}((term,p,i))$ = $(\overline{asg}_A(term),p,i)$ =
$op_{sTk}(e_{(\overline{asg}_A(term),p,i)})$ = $op_{sTk}(m_{asg}(e_{(term,p,i)}))$.
Analogously, $m_{asg}(op_{tTk}(e_{(term,p,i)}))$ = $op_{tTk}(m_{asg}(e_{(term,p,i)}))$.

The transformation step $d : G \stackrel{p_t}{\Longrightarrow} H$ is constructed as follows: Using rule p_t and match m_{asg}, we obtain for $G = Tr(N,m)$ a transformation step as double pushout of $p_t : L_t \stackrel{l_t}{\longleftarrow} I_t \stackrel{r_t}{\longrightarrow} R_t$ and m_{asg} in $\mathbf{Graphs_{TG_{AHL}}}$ due to Def. 2.1.10. The gluing object I_t just contains all places of L_t as these are the only nodes preserved by the rule. As the m_{asg} is injective, the pushout object H is constructed by $H \widetilde{=} G - m_{asg}(L_t) + m_{asg}(l_t(I_t)) + m^*_{asg}(R_t) - m^*_{asg}(r_t(I_t))$. As $D \widetilde{=} G - m_{asg}(L_t) + m_{asg}(l_t(I_t))$ and $g : D \to G$, $h : D \to H$ are inclusions, we have $H \widetilde{=} G - m_{asg}(L_t) + m^*_{asg}(R)$.

Places are preserved by the rule, i.e. if $p \in G$ then $p \in H$. The transition and its adjacent arcs of type $ArcPT$ and $ArcTP$ are deleted and generated again. This means, if $t \in G$ then $t \in H$ and if $e_{ArcTP}, e_{ArcPT} \in G$ then $e_{ArcTP}, e_{ArcPT} \in H$. Tokens $tk \in G_{Token}$ which are in the match of m_{asg} are deleted. All other tokens are preserved, i.e. they are in D. Tokens in R_{Token} are generated, i.e. if $tk \in R_{Token}$ then $m^*_{asg}(tk) \in H_{Token}$. For token values of $tk \in m^*_{asg}(R_{Token})$ we have $attr^H_{tv}(tk) = m^*_{asg}(attr^R_{tv}(tk))$ and for the token edges $e_{tk} \in m^*_{asg}(R_{EdgeTk})$ we have $attr^H_{tv}(op^H_{sTk}(e_{tk})) = m^*_{asg}(attr^R_{tv}(op_{sTk}(e_{tk})))$.

We show now that $H = Tr(N,m')$. We know that $H = Tr(N)$ for all sorts except Token, EdgeTk and the adjacent operations, because rule p_t preserves all places and deletes and generates the transition and arcs of type $ArcPT$ and $ArcTP$. Concerning the tokens in H, we have to show that m_{back}, the marking of $Back(H)$, equals m', the resulting marking after the firing step.

$$
\begin{aligned}
m_{back} &= \sum_{e_{tk} \in H_{EdgeTk}} (attr^H_{tv}(op^H_{sTk}(e_{tk})), op^H_{tTk}(e_{tk})) \\
&\quad \text{(due to the definition of } m_{back} \text{ in Def. 3.5.8)} \\
&= \sum_{e_{tk} \in G_{EdgeTk}} (attr^G_{tv}(op^G_{sTk}(e_{tk})), op^G_{tTk}(e_{tk})) \\
&\quad - \sum_{e_{tk} \in m_{asg}(L)} (attr^L_{tv}(op^L_{sTk}(e_{tk})), op^L_{tTk}(e_{tk})) \\
&\quad + \sum_{e_{tk} \in m^*_{asg}(R)} (attr^R_{tv}(op^R_{sTk}(e_{tk})), op^R_{tTk}(e_{tk})) \\
&\quad \text{(due to definition of } H \text{ as pushout)} \\
&= m - \sum_{e_{(\overline{asg}_A(term),p,i)} \in m_{asg}(L)} (\overline{asg}_A(term, p)) \\
&\quad + \sum_{e_{(\overline{asg}_A(term),p,i)} \in m^*_{asg}(R)} (\overline{asg}_A(term, p)) \\
&\quad \text{(due to def. of } m \text{ as marking of } Back(G), \text{ and of } attr_{tv}, op_{sTk} \text{ and } op_{tTk}) \\
&= m - \sum_{(term,p,i) \in PreSet_t} (\overline{asg}_A(term), p) \\
&\quad + \sum_{(term,p,i) \in PostSet_t} (\overline{asg}_A(term), p) \\
&\quad \text{(due to def. of } L_{EdgeTk}, R_{EdgeTk} \text{ in Def. 3.5.6)} \\
&= m - \sum_{j=1}^{n} (\overline{asg}_A(term_i), p_i) \text{with } (term_j, p_j, i) \in PreSet_t, n = |PreSet_t| \\
&\quad + \sum_{j=1}^{n} (\overline{asg}_A(term_i), p_i) \text{with } (term_j, p_j, i) \in PostSet_t, n = |PostSet_t| \\
&= m \ominus pre_A(t, asg_A) \oplus post_A(t, asg_A) \text{ (due to def. of } pre_A, post_A \text{ in Def. 3.5.2)} \\
&= m' \quad \text{(due to definition of } m' \text{ in Def. 3.5.2)}
\end{aligned}
$$

ad (2)
We prove the correspondence of firing sequences and transformation sequences by structural induction over the length of the sequences.

Induction Anchor
For the length of 0, there is no firing sequence. We only have to show that for a marking m of N and the translation $G = Tr(N, m)$ the marking of the backward translation $B = Back(Tr(N, m))$ equals m. We have shown this in Lemma 3.5.9.

For the length of 1, we have one firing step $m[t, asg\rangle m'$ and we have to show that for

$G = Tr(N, m)$ there is a transformation step $d : G \stackrel{p_t}{\Longrightarrow} H$ where p_t is the simulation rule corresponding to transition t, s.t. the marking m' corresponds to the marking of the backward translated AHL net $Back(H)$. We have proven this in part 1 (ad (1)) of this proof.

Induction Step
We have to show that if there is a correspondence of firing sequences and transformation sequences of length n, then there is also a correspondence of sequences of length $n + 1$. We know that the n'th firing step from the firing sequence of length n, $m_n[t_n, asg_n\rangle m'_n$ corresponds to a transformation $Tr(N, m_n) \stackrel{r_{t_n}}{\Longrightarrow} Tr(N, m'_n)$. For the firing step m_{n+1} $[t_{n+1}, asg_{n+1}\rangle m_{n+1}$ with $m'_n = m_{n+1}$ the corresponding transformation step is defined by $Tr(N, m_{n+1}) \stackrel{r_{t_{n+1}}}{\Longrightarrow} Tr(N, m'_{n+1})$.

We have to show that the marking of $Back(Tr(N, m_{n+1}))$ equals the marking of $Back$ $(Tr(N, m_{n'}))$. By the definition of backward translation we know that the marking of $Back(Tr(N, m_{n+1}))$ equals m_{n+1} and the marking of $Back(Tr(N, m'_n))$ equals m'_n. As we know that $m'_n = m_{n+1}$, the transformation sequence is now of length $n+1$ and because the marking of $Back(Tr(N, m'_{n+1}))$ equals m'_{n+1}, it has the desired property.

□

3.5.2 Amalgamation Semantics for Algebraic High-Level Nets

In this section, we define AHL net behavior by parallel graph transformation (amalgamation approach) and compare this approach to the compiler approach presented in Section 3.5.1.

For the construction of the covering construction for simulation rules we need a virtual model to define the set of matches MA from all subrules and rules in the interaction scheme (see Def. 2.1.30). This graph needs to supply all the information we need for the simulation rule construction. The virtual model contains the predomains of all transitions in form of virtual tokens, i.e. tokens being the terms in $PreSet$ corresponding to the ArcPT inscriptions, and the information about the postdomains in form of ArcTP inscriptions. As we use only "virtual" tokens, we call this graph V *virtual AHL net model*. The amalgamation construction over V then yields amalgamated rules containing the transitions and the adjacent arcs. Thus we apply a restriction functor after the amalgamation and show that the result is equivalent to the sequential simulation rules. Note that so far we do not consider firing conditions in the amalgamation, i.e. the correspondence result (Theorem 3.5.18) holds only for AHL nets without firing conditions like the Dining Philosophers.

Definition 3.5.11 (*Virtual AHL Net Model*)
Let N be an AHL net, and $Tr(N, m)$ the corresponding attributed graph (acc. to Def. 3.5.4).

The virtual AHL net model V corresponds to $Tr(N, m)$, but is marked by terms $(term, p, i) \in PreSet$ (which virtually enables all transitions):

$V = Tr(N, m)$ for all sorts except Token, EdgeTk and the adjacent arc operations:

$V_{Token} = \{tk | tk = (term, p, i) \in PreSet\}$, $V_{EdgeTk} = \{e_{tk} | tk \in V_{Token}\}$, and the operations op_{sTK}, op_{tTK}, and $attr_{tv}$ are defined as the corresponding operations for the simulation rule sides in Def. 3.5.6.

△

Example 3.5.12 (*Virtual AHL Net model for the* Dining Philosophers)
The bottom graph in Fig. 3.40 shows the virtually marked AHL net graph V_{DIPHI} for our sample AHL net modeling the Dining Philosophers. Note that the marking of the virtual AHL net model denotes the union of predomains of all transitions and has nothing to do with a specific marking as e.g. shown in Fig. 3.37.

△

Next, we define an interaction scheme for AHL nets according to Def. 2.1.28.

Definition 3.5.13 (*Interaction Scheme for AHL Nets*)
The interaction scheme IS_{AHL} consists of two subrules glueTrans and gluePlace, two extending rules get and put, and four subrule embeddings t_1 : glueTrans → get,t_2 :glueTrans→put, t_3 : gluePlace→ get and t_4 : gluePlace → put.

Fig. 3.39 shows the interaction scheme IS_{AHL}, i.e. the definitions of the subrules, the extending rules, and the four embeddings. For each rule, the algebra is the term algebra $T_{OP}(Y)$ where Y is the set of variables depicted at graph objects in Fig. 3.39. The interaction scheme IS_{AHL} is local, as e.g. subrule glueTrans is source of embeddings to both extending rules get and put.

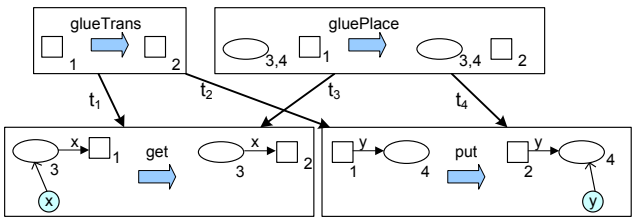


Figure 3.39: Interaction Scheme for AHL nets

△

Example 3.5.14 (*Partial Covering for the AHL net* Dining Philosophers)

Given interaction scheme IS_{AHL} as defined in Def. 3.5.13. An instance interaction scheme IIS_{take} is shown in the upper part of Fig. 3.40. (Note that the detailed presentation on the left does not include all gluePlace copies.) Then, $COV_{take} = (IIS_{take}, MA_{take})$ is a partial covering with MA_{take} being a set of matches from IS_{AHL} into V_{DIPHI} as shown at the bottom of Fig. 3.40. The matches in MA_{take} are indicated in Fig.3.40 by a fat arc inscribed by MA_{take} and given precisely by node numbers. All matches from all extending rules of all subrule embeddings in IIS_{take} commute with the matches of the subrules.

COV_{take} is local as IIS_{take} is local (the subrule $glueTrans$ is embedded in all extending rules) and because the matches from the left-hand sides of all three extending rule instances of get into $G_{AHL_{DIPHI}}$ are non-isomorphic. COV_{take} is additionally fully synchronized, because for each pair of extending rules we find a subrule s.t. the matches of their left-hand sides into $G_{AHL_{DIPHI}}$ overlap only in the match of this common subrule.

△

Note that, if a graph G and an interaction scheme are given and the covering is characterized (as e.g. for AHL nets the covering must be local, and fully synchronized), then the set of all partial coverings, i.e. the instance interaction schemes and the set of matches MA from all rules and subrules from the instance interaction scheme into G can be computed automatically.

For the covering construction for the AHL net *Dining Philosophers* this means that we can find two basic partial coverings – one for transition take in V_{DIPHI} (as shown in Fig. 3.40), and the other one for transition put. In the second case, a different instance interaction scheme is computed with three instances of rule put and one instance of rule get. From one instance of the subrule gluePlace there are embeddings into two of the put instances, and from one instance of the subrule glueTrans there are embeddings into all get and put instances.

A desired property of our AHL net covering construction is that it can be computed deterministically in the sense that the rules resulting from the amalgamation are unique. This property will be shown in Proposition 3.5.17.

Example 3.5.15 (*Amalgamated Rule for the AHL net* Dining Philosophers)

Let $COV_{take} = (IIS_{take}, MA_{take})$ be the partial covering construction as defined in Def. 3.5.14. The LHS (RHS) of the amalgamated rule p_{take} for this partial covering is constructed according to Def. 2.1.31 by gluing the instances of the LHS (RHS) of get and put along the objects of the LHS (RHS) of their common subrules.

In the center of Fig. 3.40, the construction of the amalgamated rule $p_{amalg_{take}}$ from

COV_{take} is shown. The embeddings of rules and subrules into the amalgamated rule are indicated by dashed arrows and given precisely by numbers.

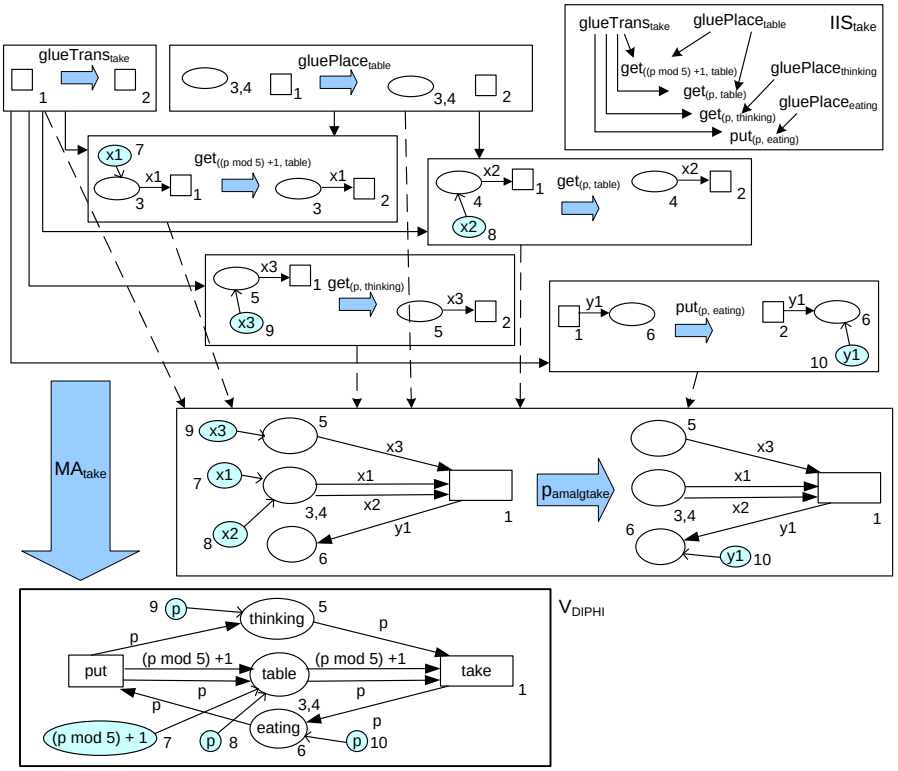


Figure 3.40: Covering Construction COV_{take} and Amalgamated Rule $p_{amalg_{take}}$

△

The result of the amalgamation, $p_{amalg_{take}}$, is a rule corresponding to the simulation rule for transition take with two slight differences. The variables $x_1, .., x_3$ and y_1 used in the amalgamated rule have to be replaced by the right terms from $T_{OP}(X)$, and the transition and arcs must not appear in the simulation rule. The rewriting step for the variables is given by the matches in MA_{take}, where x_1 is matched to (p mod 5) + 1, and x_2, x_3 and y_1 are matched to p. The transition and arcs disappear by applying a functor restricting an ASSIG algebra such that the sorts $Trans$, $ArcPT$ and $ArcTP$ and the adjacent arc operations are empty.

The general construction of a partial covering for a transition $t \in T$ is the basis for the correspondence proof in Theorem 3.5.18.

Construction 3.5.16 (*Partial Coverings for Amalgamated Rules modeling AHL Net Behavior*)

Let V be the virtual AHL net model for net N defined in Def. 3.5.11. Let $COV_t =$

(IIS_t, MA_t) be the partial covering for a transition $t \in V_{Trans}$ with IIS_t being an instance interaction scheme of IS_{AHL} as defined in Def. 3.5.13 and MA_t the set of matches from IIS_t into V. IIS_t and MA_t are defined as follows:

- *Extending rule instances:* For each edge $arcPT \in V_{ArcPT}$ there is one instance get_{arcPT} of the extending rule get. For each edge $arcTP \in V_{ArcTP}$ there is one instance put_{arcTP} of the extending rule put.
- *Subrule instances:* There is one instance of subrule glueTrans for transition t which is embedded into all get and put instances as defined in Def. 3.5.13. For each place $p \in N_{Env_t}$ there is one gluePlace instance, called $\mathsf{gluePlace}_p$, which is embedded into all those extending rule instances get_{arcPT} with $op_{sPT}(arcPT) = p$ similar as in Def. 3.5.13. Analogously, $\mathsf{gluePlace}_p$ is embedded into all those extending rule instances put_{arcTP} with $op_{tTP}(arcTP) = p$.
- *Matches in MA_t:* The transitions of all rules and subrules in IIS_t are mapped to $t \in V_{Trans}$. The place nodes from get instances are mapped to place nodes in $pre(t)$ such that the arc inscription and the token value are mapped to the same term and the mappings overlap only in the matches of their subrules in IIS_t. Place nodes from put instances are mapped to place nodes in $post(t)$ such that the mappings overlap only in the matches of their subrules.

△

Proposition 3.5.17 (*Existence and Uniqueness of Partial Covering COV_t*)
Let V be the virtual AHL net model for net N defined in Def. 3.5.11. For each transition $t \in V_{Trans}$ a local, fully synchronized partial covering $COV_t = (IIS_t, MA_t)$ constructed as in Construction 3.5.16, exists and is unique. △

Proof: We show that

1. there is at least one partial covering COV_t which is local and fully synchronized (due to the instance of glueTrans in IIS_t).
2. COV_t is unique by assuming that there are two different partial coverings $COV1_t$ and $COV2_t$ and by showing that they are equal.

ad (1)
Taking an arbitrary transition $t \in G_{Transition}$ we show that there exists exactly one partial covering $COV_t = (IIS_t, MA_t)$.

There is at least one partial covering.
According to Def. 3.5.16 there is one instance of subproduction glueTrans in IIS_t. Since

its left and right-hand sides contain only a transition node each, there exists a match of glueTrans to G. For all extending production instances in IIS_t we know that their matches overlap in the match of glueTrans. Thus, a partial covering COV_t exists (independent of the number of extending productions).

The partial covering is local and fully synchronized.
COV_t is local, since glueTrans is a subproduction of all instances of get and put. Moreover, each two instances of get match to two different $arcPT_1, arcPT_2 \in G_{ArcPT}$ with $op_{tPT}(arcPT_1) = op_{tPT}(arcPT_2) = t$ and each two instances of put match to two different $arcTP_1, arcTP_2 \in G_{ArcTP}$ with $op_{sTP}(arcTP_1) = op_{sTP}(arcTP_2) = t$.

COV_t is fully synchronized, since for each two instances i_1, i_2 of get or put one of the following cases holds:

- Both instance matches overlap only in transition node t. Then, their matches overlap only in the match of glueTrans.
- Both instance matches overlap also in their place node p. Then, there is an instance of gluePlace such that its match is the intersection of the matches of i_1 and i_2.

ad (2)
The partial covering is unique.
Assume we have two different partial coverings $COV1_t$ and $COV2_t$. Since both are local and fully synchronized, they have to differ in the set of instances of get, put, or gluePlace. Due to Def. 3.5.16 in IIS_t, we have one instance get_{arcPT} of get for all $arcPT \in G_{ArcPT}$ with $op_{tPT}(arcPT) = t$, one instance of put_{arcTP} of put for all $arcTP \in G_{ArcTP}$ with $op_{sTP}(arcTP) = t$, and one instance of gluePlace_p of gluePlace for all $p \in G_P$. There are no conflicts between any two instances of get and put, since we have shown in the first part that COV_t is fully synchronized, independent of the number of instances of get and put. In the case that both instance matches overlap in more than transition t, this is only true, if there is an instance of gluePlace such that its match is the intersection of both instance matches. This holds, since there is an instance gluePlace_p for all $p \in N_{Env_t}$.

□

On the basis of the unique construction of the amalgamated rule $p_{amalg} : L_{amalg_t} \rightarrow R_{amalg_t}$ using the virtual AHL net model V as host graph (step (1) in Fig. 3.41), we get the match $m_{cov} : L_{amalg_t} \rightarrow V$ by gluing the matches in MA_t along the matches of the subrules (step (2) in Fig. 3.41). Then we apply the amalgamated rule p_{amalg} at match m_{cov} to V (step (3) in Fig. 3.41). The resulting span $V \leftarrow V_I \rightarrow V'$ can be interpreted as rule again. This rule still contains all AHL net places, arcs and the transitions due to V being constructed once for the complete AHL net N. So we now restrict $V \leftarrow V_I \rightarrow V'$ to the elements of the environment of transition t. This transformation step is depicted as step (4) in Fig. 3.41. The result is the span $V|_{codom(m_{cov})} \leftarrow V_I|_{codom(i)} \rightarrow V'|_{codom(m^*_{cov})}$ which

looks similar to our sequential simulation rule p_t with the difference that it still contains the transition and the adjacent arcs. Thus, in a last step (step (5) in Fig. 3.41) we apply a functor which forgets the transition and its adjacent arcs.

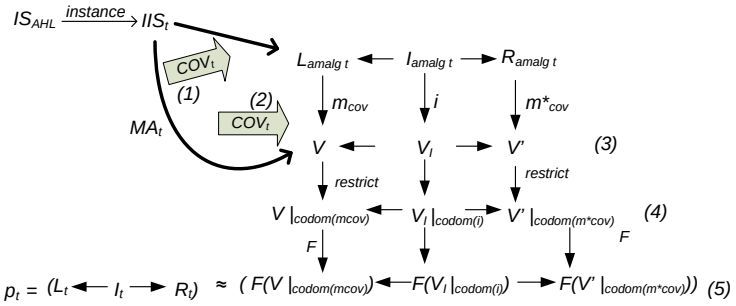


Figure 3.41: Correspondence of Amalgamated Rules and Simulation Rules for AHL Nets

Theorem 3.5.18 now formally states that the rule resulting from this functor application is isomorphic to the sequential simulation rule p_t as defined in Def. 3.5.6.

Theorem 3.5.18 (*Correspondence of Amalgamated Rules and Simulation Rules for AHL Nets*)

Let N be an AHL net and $m \in M^{\oplus}$ its initial marking. Let $SimSpec(N)$ be the simulation specification of N with the set of simulation rules $P_{sim} = \{\hat{p}_t : L_t \to R_t | t \in T\}$.
Let V be the virtual AHL net model for N acc. to Def. 3.5.11.

Then for each transition $t \in T$ the following holds: Given $COV_t = (IIS_t, MA_t)$, the partial covering for transition t constructed as in Constr. 3.5.16, and $p_{amalg_t} : L_{amalg_t} \to R_{amalg_t}$, the amalgamated rule for COV_t. Let m_{cov} be the match from L_{amalg_t} to V, with m_{cov} being the gluing of MA_t, and let $V \overset{p_{amalg_t}, m_{cov}}{\longrightarrow} V'$ be the transformation step. Performing an epi-mono-factorization of the corresponding rule embedding (m_{cov}, i, m^*_{cov}) leads to a new rule $p_{codom} = (codom(m_{cov}) \leftarrow codom(i) \rightarrow codom(m^*_{cov}))$. Let F be a functor that forgets transition and arcs, i.e. the sorts $Trans, ArcPT, ArcTP$ and all adjacent operations are empty. Then, $F(p_{codom}) \overset{\sim}{=} p_t$.

△

Proof: We construct p_{amalg_t} and $m_{cov} : L_{amalg_t} \to V$ and show that

1. the transformation step $V \overset{p_{amalg_t}, m_{cov}}{\longrightarrow} V'$ restricted to the codomain of rule embedding (m_{cov}, i, m^*_{cov}) corresponds to p_t except that it still contains the transition and adjacent arcs.

2. $F((p_{codom})$ is isomorphic to p_t.

ad (1)
Given partial covering COV_t for a transition $t \in T$, the amalgamated rule $p_{\hat{amalg}_t} = ((p_{amalg_t} : L_{amalg_t} \overset{l_{amalg_t}}{\leftarrow} I_{amalg_t} \overset{r_{amalg_t}}{\rightarrow} R_{amalg_t}), \emptyset, Y)$ has to be constructed first. Due to the fact that colimit constructions are unique up to isomorphisms, the construction of p_{amalg_t} is also unique to isomorphism.

First, we have to show that L_{amalg_t} is isomorphic to $Tr(N_{Env_t})$ for all sorts except Token, EdgeTk and the adjacent operations as well as operations arc_{iPT} and arc_{iTP}. We have one transition node, since COV_t is local, i.e. subproduction glueTrans is embedded into all extending production instances of get and put. Thus, in the colimit construction all transition nodes are glued to one. For all places p in $PreSet(t)$, we know that there is at least one token required from p. Thus, there is at least one instance of get with a place node mapped to p. If there are more such instances of get, they are glued by an instance of gluePlace, since for each place p in N_{Env_t} there is a production gluePlace. Similarly for all places p in $PostSet(t)$, at least one token is put to p after firing t. Thus, there is at least one instance of put with a place node mapped to p. If there are more such instances of put, they are again glued by an instance of gluePlace. All place nodes glued occur only once in L_{amalg_t}. Edges between place and transition nodes as well as tokens and their place-assigning edges in L_{amalg_t} are not glued, i.e. they occur in L_{amalg_t} as often as instances of productions get and put are in COV_t. Compare Def. 3.5.16 for the number of instances. Since there is an instance of get in COV_t for each $(term, p, i) \in PreSet_t$ and an instance of put in COV_t for each $(term, p, i) \in PostSet_t$, we get for sorts Token, EdgeTk and the adjacent operations as well as operations arc_{iPT} and arc_{iTP}:

- $L_{amalg_{t\,Token}} = \{tk | tk = (term, p, i) \in PreSet_t\}$
- $L_{amalg_{t\,EdgeTk}} = \{e_{tk} | tk \in L_{amalg_{t\,Token}}\}$,

- $op_{sTK} : L_{amalg_{t\,EdgeTk}} \rightarrow L_{amalg_{t\,Token}}$ with $op_{sTk}(e_{(term,p,i)}) = (term, p, i)$,
- $op_{tTK} : L_{amalg_{t\,EdgeTk}} \rightarrow L_{amalg_{t\,Place}}$ with $op_{tTk}(e_{(term,p,i)}) = p$,

- $attr_{tv} : L_{amalg_{t\,Token}} \rightarrow \mathbb{N}$ with $attr_{tv}((term, p, i)) = x_{(term,i)}$
- $attr_{iPT} : L_{amalg_{t\,ArcPT}} \rightarrow \mathbb{N}$ with $attr_{iPT}((term, p, i)) = x_{(term,i)}$
- $attr_{iTP} : L_{amalg_{t\,ArcTP}} \rightarrow \mathbb{N}$ with $attr_{iTP}((term, p, i)) = y_{(term,i)}$

I_{amalg_t} just contains all places of L_{amalg_t}. These are the only nodes preserved by the productions in IIS_t.

Since there is an instance of put in COV_t for each $(term, p, i) \in PostSet_t$, R_{amalg_t} is constructed similarly to L_{amalg_t} except for sort Token and operation attr_{tv}.

- $R_{amalg_{t_{Token}}} = \{tk | tk = (term, p, i) \in PostSet_t\}$
- $attr_{tv} : R_{amalg_{t_{Token}}} \rightarrow I\!N$ with $attr_{tv}((term, p, i)) = y_{(term,i)}$

l_{amalg_t} and r_{amalg_t} are the obvious embeddings. They result from gluing the embeddings of interfaces in productions of IIS_t.

Variable set Y to sort Nat consists of variables $\{x_i\} \cup \{y_i\}$ for $i, j \in T_{OP}(X) \times I\!N$.

The match $m_{cov} : L_{amalg_t} \rightarrow V$ is given by identical mappings on the places, transitions and arcs. For the tokens, we have $m_{cov_{DSIG}} : Y \rightarrow T_{OP}(X)$ assigning each variable in Y a term in $T_{Nat}(X)$ with $m_{Nat}(y_{(term,i)}) = term$ if $(term, p, i) \in PreSet_t$. Otherwise, $y_{(term,i)}$ is mapped to some variable in X not used in V. The codomain of m_{cov} is the subgraph of V containing the places, transitions and arcs of N_{Env_t} and all tokens from V connected to place nodes from N_{Env_t}.

In Def. 3.5.6, we defined $L_t = Tr(N_{Env_t})$ for all sorts except Token, EdgeTk and the adjacent operations. Thus, $codom(m_{cov}$ and L_t represent the same net structure (except for arc inscriptions). Moreover, there is a token in $codom(m_{cov_{Token}}$ for each $tk = (term, p, i) \in PreSet_t$, since for each $(term, p, i) \in PreSet_t$ there is an arc $arcPT \in V_{ArcPT}$, and thus one instance get_{arcPT}, due to Def. 3.5.16.

Applying p_{amalg_t} at match m_{cov} to $codom(m_{cov})$, we must show that the result graph is isomorphic to $codom(m^*_{cov})$. As stated above, L_{amalg_t} and R_{amalg_t} differ only in their token sets and adjacent edges and attributes. The same is true for $codom(m_{cov})$ and $codom(m^*_{cov})$ because here the application of p_{amalg_t} is reflected. The only difference are the token attributes which are terms in $T_{OP}(X)$ here. It is obvious that $m_{cov_{DSIG}} = m^*_{cov_{DSIG}}$.

The only difference between $codom(m_{cov})$ and L_t [$codom(m^*_{cov})$ and R_t] is that L_t [R_t] contains no arc inscriptions.

ad (2)
Applying the restriction functor F to the rule $p_{codom} : codom(m_{cov}) \rightarrow codom(m^*_{cov})$, we have to show that $F(p_{codom})$ is isomorphic to p_t. As the arc inscription operations are empty for $F(codom(m_{cov})$, the restricted rule $F(p_{codom})$ equals p_{codom} without arc inscriptions, and hence is isomorphic to p_t.

□

3.5.3 GTS-Compiler Semantics for Integrated UML Models

In this section we discuss the GTS-compiler semantics for models integrating selected UML diagram types (use cases, class diagrams, collaboration diagrams, Statecharts and object diagrams). The approach has been developed at the University of Bremen [KGKK02], but fits nicely in the simulation framework discussed in this chapter. We just sketch the

automatic translation of UML models to model-specific simulation specifications using a running example, because the resulting simulation specification is taken as basis for an animation specification in Chapter 4.

Note that the semantical compatibility of source model semantics and simulation specification cannot be proven formally for UML models, as the semantics of the integrated UML model (the source model of the model transformation) is formally defined just by this model transformation to a simulation specification.

In general, UML models are composed of several UML diagrams such as use-case, class, object, state, collaboration, sequence, and activity diagrams (see [BRJ98, UML04a]). The interplay of UML diagrams – as used in our integration approach – is depicted in Fig. 3.42. An integrated UML model consists of one class diagram and one top-level use-case diagram containing the use cases of the simulation which are refined by collaboration diagrams. For each class, there can be a state diagram describing in which order the operations of the class can be executed. Object diagrams which instantiate the class diagram model possible states of the system.

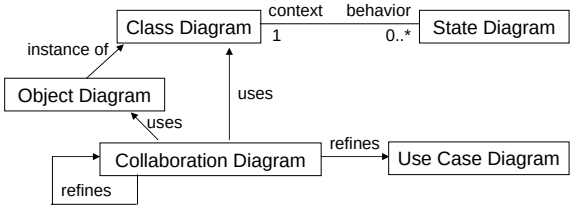


Figure 3.42: Central UML modeling concepts and their interplay

A Sample UML Model

To illustrate the connection between the different UML modeling concepts, we sketch a sample UML model of a client-server system which consists of a set of drive-through restaurants each of which has a (possibly empty) queue of hungry clients that are served one after the other. There may also exist some more idle people who are not yet visiting a drive-through.

The user of the drive-through simulation may select idle people to be hungry and drive-throughs to start serving clients. Hence, the *use-case diagram* of our simulation system consists of two use cases, namely callClientToEat and startDriveThrough. Both are refined by a set of collaboration diagrams, one of which is presented below.

The *class diagram* is depicted in Fig. 3.43. It consists of the four classes Client, DriveThrough, Meal, and Order. Clients may be associated with a drive-through via a Visit-association (in this case they are hungry). Every client in a queue is linked with his/her

successor provided that he/she is not the last person in a queue. This is reflected in the association Queue. The first and the last client are also linked to the drive-through they are visiting. Clients may submit orders and eat meals that are served by drive-throughs. Drive-throughs and clients can perform a series of operations the most important of which are shown in the figure. The names of the meals and the orders are given by attributes.

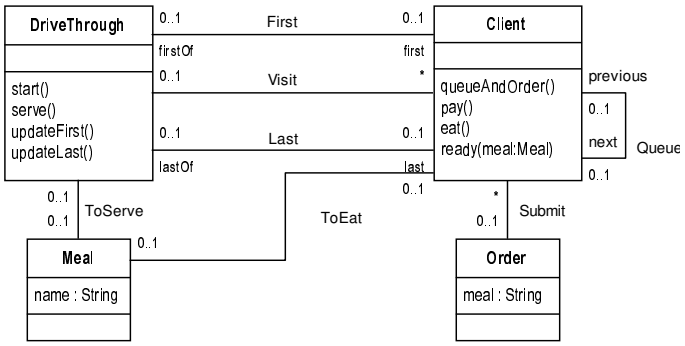


Figure 3.43: Class Diagram for the Drive-Through Model

Instances of class diagrams are *object diagrams* consisting of a set of objects for every class, and links for every association so that the multiplicity requirements of the associations are satisfied.

Fig. 3.44 presents the *state diagram* for the class Client. Initially, a client is in the state idle. In this state the operation queueAndOrder can be executed which also changes the state of the client from idle to waiting. In the state waiting the client can pay and change its current state to hasPaid. After paying the client can eat and be idle again. Operations which do not occur in the state diagram can be executed in every state. (By the collaboration diagrams of the model it is guaranteed that the operation ready is executed only between the operations pay and eat.)

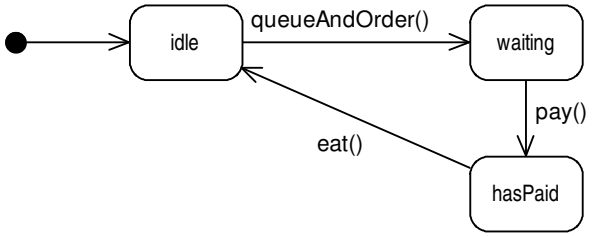


Figure 3.44: State Diagram for Class Client

Operations of classes can be described with *collaboration diagrams*. The collaboration diagram in Fig. 3.45 specifies the operation serve in which a drive-through d serves a meal m ordered by the first client c of its queue. First, the operation creates the meal-object m (1.1). Second, it inserts a ToServe-link between d and m (1.2). Third, the attribute name of

m is set to the meal-attribute of the order submitted by c (1.3). Fourth, d sends the message ready(meal) to c (1.4) with the effect that the Submit-link attached to c is deleted (1.4.1) and a ToEat-link between c and m is created (1.4.2). Finally, the ToServe-link between d and m is deleted (1.5).

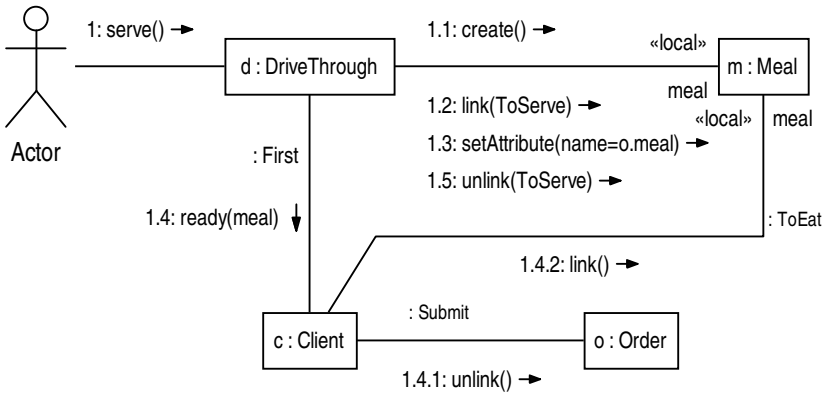


Figure 3.45: Collaboration Diagram for serve()

It is worth noting that the main operations modeled in the collaboration diagram are the creation and deletion of links or objects or the sending of messages. Graph transformation systems can model such operations in a straightforward way.

In the next section we sketch how UML models can be automatically translated into behavior specifications in order to obtain a behavioral semantics of UML models allowing to execute the behavior of a UML model by applying the rules of the behavior specification.

Integrated UML Semantics in the Compiler Approach

Up to now, the approach to generate an integrated semantics of a part of UML takes into account use-case diagrams, class and object diagrams, state diagrams, and sequence and collaboration diagrams. The model transformation of all these diagrams yields an attributed graph transformation system which consists mainly of a set of simulation rules (the behavior specification) and a graph representing the initial system state. The compiler semantics of the system then consists of all system states that can be reached from the initial state via the iterated application of simulation rules. For the automatic translation of UML models to graph transformation systems the prototypic tool UGT (*UML to Graph Transformation*) has recently been developed [Sch05]. UGT reads a UML model specification from a given text file and automatically generates the graph transformation rules according to the model transformation algorithm given in [ZHG04a, ZHG04b]. The initial graph is computed from the input model plus an object diagram specified by the modeler as initial.

As stated before, the integrated semantics of a UML model is given by a behavior specification modeling the modification of system states via the application of simulation rules.

A system state is a kind of extended object diagram representing the objects that are alive in the system state.

Fig. 3.46 shows an excerpt of a system state of our drive-through model. It consists of two clients (o1 and o2), together with their current states, a drive-through (o3), and an order (o4). Both clients are visiting the drive-through, o1 being the first and o2 being the second and last client in the queue. The order has been submitted by the first client.

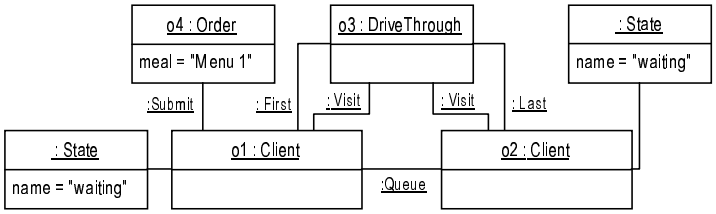


Figure 3.46: A System State of the Drive-Through Model

Technically, the classes, their attributes and operations, the associations, the initial states of the objects as well as a node for every use case are also included in every system state. For reasons of space limitations and clarity they are omitted here.

An initial state of a UML model consists of all objects and links (together with the corresponding classes, associations, initial states, etc.) that are alive in the beginning of the system simulation. (The state graph in Fig. 3.46 is not an initial state because the clients are not in their initial state idle.)

Each operation of a class and the use cases are translated into a set of graph transformation rules according to the specifications given by the collaboration and state diagrams. Technically, the control flow of operation execution is modeled by means of process nodes. These nodes fall into two main categories: complex and atomic ones. The complex process nodes represent operations that are composed of suboperations as specified in the collaboration diagrams. Atomic process nodes represent predefined, basic operations which specify the creation of objects, the insertion of links between them, the setting of their attribute values, and the like.

When a state diagram is specified for an object, the execution of an operation may not be defined to take place in a certain object state. In this case, a corresponding process node is nonetheless added to the system state, but it can never get active. In this way, an operation call will be ignored by our approach if the called object is in a “forbidden” state. Currently, our approach supports basic state diagrams without advanced features like nested states.

Should the model be incomplete (e.g. a class operation without a specification in a collaboration diagram), no rules are generated for the incomplete part. Thus the system execution will get stuck, due to the lack of adequate rules (in case of a not specified operation there will be no rules to properly execute and terminate it, thus succeeding operations may

never be executed).

Fig. 3.47 presents an example of a graph transformation rule. Both sides of the rule contain three common nodes representing an object of class DriveThrough, an object of class Client and an object of class Order. The application of the rule to a graph inserts an additional node (an object of class Meal) of the right-hand side into the graph it is applied to.

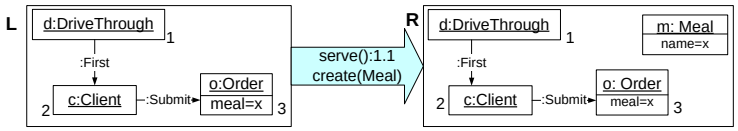


Figure 3.47: Rule for the Operation serve():1.1 create(Meal)

This rule is obtained automatically from the collaboration diagram in Fig. 3.45. Roughly speaking, as left-hand side all objects and links from the collaboration diagram are taken that are existing before the execution of the operation create. As specified in the collaboration diagram the execution adds a meal-object. This rule realizes the effect of operation 1.1 of the collaboration diagram in Fig. 3.45. There the execution of operation 1.1 create is specified to create a new Meal object.

The rule presented in Fig. 3.47 summarizes the effect of a set of four more detailed graph transformation rules that are applied in a special order. These detailed rules would be applied to the complete system state graph, of which Fig. 3.46 shows an excerpt.

Another example for a rule corresponding to the operation 1.4 in the collaboration diagram for serve() in Fig. 3.45 is the rule shown in Fig. 3.48. As specified by the operations 1.4.1 and 1.4.2 of the collaboration diagram, the Submit link between a Client object and an Order object is deleted, and a ToEat link between the Client object and the Meal object is created under the condition that the name attribute of the Meal object corresponds to the meal attribute of the Order object. Again, this is a summary of a set of more detailed rules realizing the unlinking and linking in several steps, taking into account also the link ends.

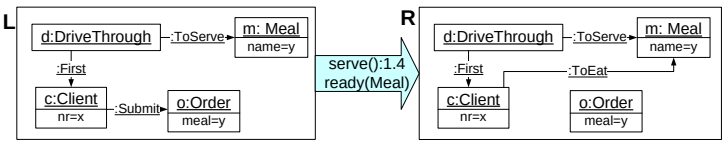


Figure 3.48: Rule for the Operation serve():1.4 ready(Meal)

Summarizing, for every operation of a class specified in a collaboration diagram we can automatically generate a set of graph transformation rules the application order of which is determined by way of so-called process nodes. Furthermore, the state diagrams determine in which object states the operations can be executed. Finally, all parts of the class diagram

and the use cases are reflected in the system states. Hence, the graph transformation system is obtained from the use-case, class, state, and collaboration diagrams given for the model.

In Chapter4, we continue the DriveThrough example by giving a complete simulation specification and by describing how system states and their transformations can be enriched to obtain an application oriented animation view.

3.6 Related Work

Like in the case of text-based modeling languages, there are in principle the two possibilities of *operational* and *denotational* semantics also for visual models.

In contrast to the textual case, the semantics of visual models is not yet worked out systematically. But one encounters a number of tentative proposals in the literature. Many of them point in the direction that the transformation of graphs, diagrams, or other kinds of states or configurations may play a similar central role to define an operational semantics as term rewriting in the traditional case of textual models. But also for the definition of denotational semantics, graph transformations can be used to define the mapping of a model into a semantic domain. In this section we give a short overview of the semantic potentials of graph transformation as given in the survey [KHK06], and relate approaches to defining semantics by graph transformation to the simulation approaches given in this chapter.

In our approach, we define the operational semantics of simulation specifications (i.e. graph transformation systmes) as the set of all transformations induced by the rule set P. Similar semantics definitions in the literature cover basically two variants of semantics:

(1) *Derivation Graph*: $Graph(P) = (\mathcal{C}, \overset{\mathcal{P}}{\Longrightarrow})$
where $\overset{P}{\Longrightarrow}$ is the union of the binary relations on model states for each rule: $\overset{P}{\Longrightarrow} = \bigcup_{p \in P} \overset{p}{\Longrightarrow}$. The derivation graph is a graph with the model states as nodes and transformation steps as edges.

(2) *Derivation Relation* (see e.g. [Roz97]): $\overset{P\,*}{\Longrightarrow} \subseteq \mathcal{G} \times \mathcal{G}$,
where $\overset{P\,*}{\Longrightarrow}$ is the reflexive and transitive closure of $\overset{P}{\Longrightarrow}$.

Control conditions may be defined which enhance graph transformation systems given by pure set of rules, e.g. by the distinction of some initial and terminal states, or by regulating the process of rule applications by rule layers, rule priorities or other evaluation strategies. Control conditions are used in visual graph transformation languages like PROGRES [SWZ99] and AToM^3 [dLVA04].

In our approach to simulation we use rule layers and rule priorities. Moreover, as graph transformation systems for simulation are usually not terminating, we will allow to use

iterative control structures like If-Then-Else or While as structuring principle for simulation and animation rules to define simulation scenarios in GENGED (Section 5.3). This principle is also used in transformation units [Kus00].

Interpreter Semantics Typically, the operations of an interpreter are specified using the abstract syntax of the visual modeling language (or a slight extension of it). There are three variants of interpreter semantics:

(i) The simulation rules are used for all models of a modeling VL. In this case, we have an abstract interpreter for the complete VL, and call the simulation rules *universal*. In this chapter, we used this variant of interpreter semantics in Section 3.1. Universal simualation rules are also used in [EEdL$^+$05, Var02, dLV02, EHHS00, CHM00], to define the interpreter semantics of Statecharts.

(ii) The simulation rules are *model-specific* in the sense that they define the semantics of one distinct model only. Model-specific simulation rules are used e.g. in [HHS04, KGKK02, Kus01, TE00].

(iii) Amalgamation semantics are a mixture of variants (i) and (ii). The semantics are defined by *universal* rule schemes for visual languages like Petri nets, where graph transformation systems in their usual, sequential form are not expressive enough. The universal rule schemes are instantiated for a given model to *model-specific* simulation rules. We use this variant of interpreter semantics in Section 3.3.1. This approach is used also in [dLETE04] to simulate timed-transition Petri nets.

Compiler Semantics The idea of compiler semantics is to translate (compile) a model of the source language into a model of a target language. The executability of the target language may be described by means of an interpreter semantics, e.g. given by a set of graph transformation rules working on the graph-based representations of the states of the model (see, e.g., [Kre93, Kus01, MSP96]). In this thesis, we speak of *GTS-Compiler semantics* if the target language is the VL of type graph transformation systems (Section 3.2.2). Examples for compiler semantics definitions with a target VL different from graph transformation systems are given e.g. in [TEG$^+$05, CHK04, dLV02].

Semantics defined as a mapping from the abstract syntax into some semantic domain is referred to as denotational. According to this definition, the compiler semantics is denotational as well, (in case of GTS-compiler semantics with a semantic domain that is itself operational). If possible, the mapping of models into the semantic domain should be defined separately for each element of the abstract syntax so that the meaning of the complete entity can be assembled from the meaning of its elements. This compositionality principle is typical for denotational semantics of programming languages, and it is the basis

for modular verification, analysis, and evolution of visual models (see, e.g., [EGHK02]). Assuming that the abstract syntax of visual models is represented by graphs, one faces the problem of describing a model transformation from graphs to graphs (if the semantic domain happens to have a diagrammatic syntax, like with Petri nets) or from graphs to text (if the semantic domain is algebraic or logic-based, like a process calculus). For both variants, different forms of graph transformation rules can be found in the literature (see, e.g., [Bar97, EKGH01, EGHK02]). In Section 3.2.3, the GTS compiler semantics for Condition/Event nets has been defined by a model transformation using graph transformation rules.

Chapter 4

From Simulation to Animation

Despite the benefits of graph-transformation based simulation, for validation purposes, simulation of model behavior within the means of the formal visual modeling language is not always adequate. System states are visualized in simulation runs as graphs which may become rather complex. This is sometimes due to the fact that the simulation specification is generated automatically, (like the graph transformation system generated from UML diagrams in [ZHG04b]) which necessarily involves auxiliary constructs and yields more complex rules than a hand-written specification would contain.

Therefore, in this chapter we extend the formal concept of behavior specifications to *animation specifications* which allow to define model-specific scenario animations in the layout of the application domain (cf. also [EB04]).

In contrast to simulation, *animation* is no standardized technical notion, and is generally not linked to a model or physical or abstract system. Animation is usually defined in the areas of web site design, movie industry or scientific visualization as "the creating of a timed sequence or series of graphic images or frames together to give the appearance of continuous movement" [DD04] or "a movie; a sequence of related images viewed in rapid succession to see and experience the apparent movement of objects." [Owe99, KK93].

It is worth noting that our notion of *animation* goes beyond the notion of *specification animation* in the literature. In the *formal methods* community, specification animation is understood as a way of creating an executable prototype that is generated from a formal specification [MSHB98, Gra01]. *Animation* in this thesis is also based on a simulation specification but differs from simulation in three respects:

1. Simulation presents the states of the system in the abstract, formal syntax of the modeling language (e.g. as object diagrams or graphs), whereas animation uses a domain-specific layout (an *animation view*), which is visually closer to the modeled system and hides the underlying formalism.

2. Simulation shows model state transitions as discrete steps, whereas animation shows a continuously changing scenario in a movie-like fashion.

3. Simulation shows complete states whereas animation may abstract from implementational details of the model that are not important to understand selected functional aspects of the system behavior.

We advocate that this way of animation simplifies the early detection of inconsistencies and possible missing requirements in the model which cannot always be found by simulation and analysis only.

In order to define animation based on simulation in a systematic way, we propose the extension of the underlying formal simulation specification by graph transformation rules [EE05b], which realize a consistent mapping from simulation to animation steps in the respective animation specification. Different animation views may reflect the behavior of different parts of the system. The transition from simulation to animation is realized on the basis of simulation specifications as defined in Chapter 3. The simulation rules are enriched in a systematic way with domain-specific symbols for animation, at the same time making sure that this process does not change the semantics of the model. Furthermore, animation operations are integrated with the enriched simulation rules defining continuous changes of animation symbol properties (such as position, color or size).

The most important properties of the construction of the animation specification are termination and confluence (crucial research topics in the area of model transformation in general), and the syntactical and semantical correctness of the resulting animation specification in relation to the original simulation specification.

Section 4.1 presents a conceptual overview of the different steps for the construction of an animation specification from a simulation specification (see also [EB02, EBE03, EB04]). Moreover, the desired properties of the construction and the resulting animation specification are introduced. In Section 4.2 we define and apply model transformation rules to realize a consistent extension from the simulation specification to an animation specification. As this involves the graph transformation-based change of simulation rules, a formal definition is given how to rewrite graph rules by non-deleting graph rules. An *animation view* (i.e. a set of domain-specific animation scenarios) for a simulation specification and a domain-specific visualization alphabet then is defined by restricting the transformations of the animation specification to the visualization alphabet. We illustrate each step of the animation view construction by our running example of the Echo Algorithm which was introduced in Example 3.2.3. Section 4.3 shows the termination of the animation specification construction, Section 4.4 deals with the syntactical and Section 4.5 with the semantical correctness of resulting animation specifications. In the *Applications* Section 4.7, three sample animation specifications are presented, namely for an AHL net (the well-known *Dining Philosophers* model [EE05b]), for a Statechart (the *Radio-Clock*

model), and for an integrated UML model (a client-server system modeling a drive-through restaurant [EHKZ05]). Section 4.8 compares the approach of animation presented in this thesis to related approaches realizing animation in the areas of modeling with UML diagrams or Petri nets.

4.1 General Overview

For the visualization of a model's behavior directly in a domain-oriented layout (the animation view), the system states have to be mapped onto graphical representations for real-world objects and values (represented by a so-called *visualization alphabet* TG_V of the application domain).

A behavior model is formalized by a typed graph transformation system $S = (TG_S, P_S)$ (a model-specific *simulation specification*, according to Section 3.4). As basis for the animation view definition, the simulation alphabet TG_S and the visualization alphabet TG_V are integrated yielding an *animation alphabet* $TG_A = TG_S +_{TG_I} TG_V$ (Def. 4.2.4 in Section 4.2.1). By the forward retyping functor $f^{>}_{TG_S} : \mathbf{Graphs_{TG_S}} \rightarrow \mathbf{Graphs_{TG_A}}$, the elements G_S of the simulation language VL_S are retyped over the animation alphabet TG_A. A state of the behavior model is given by a graph $G_S \in VL_S$. We can map each $G_S \in VL_S$ to a graph $G_A \in VL_A$ by applying a set of graph transformation rules Q as long as possible to G_S. This *simulation-to-animation model transformation*, short *S2AM transformation* $S2AM : VL_S \rightarrow VL_A$ (Def. 4.2.10), is defined by $S2AM = (VL_S, TG_A, Q)$.

S2AM transformation corresponds to the usual notion of model transformation from graphs of a source language VL_1 to graphs of a target language VL_2 as given in [EE05a]. Fig. 4.1 illustrates the *S2AM* transformation concept.

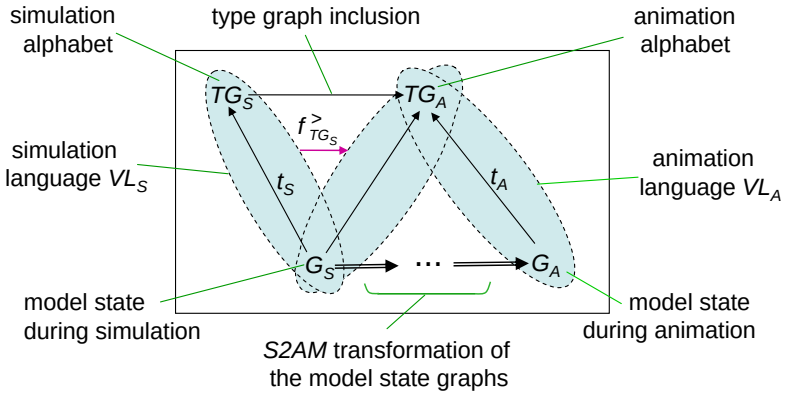


Figure 4.1: *S2AM* Transformation from Simulation to Animation Language

Our aim is to transform a simulation specification $S = (TG_S, P_S)$ into an animation specification $A = (TG_A, P_A)$. Hence, in addition to the concept of *S2AM transforma-*

tion, we need a construction which allows us to apply the transformation rules in Q to the simulation rules in P_S (see Def. 4.2.12 in Section 4.2.2). This construction is the basis for the definition of *simulation-to-animation rule transformation*, short $S2AR$ *transformation* $S2AR : P_S \rightarrow P_A$ (Def. 4.2.14), defined by $S2AR = (P_S, TG_A, Q)$. Fig. 4.2 illustrates the $S2AR$ transformation concept.

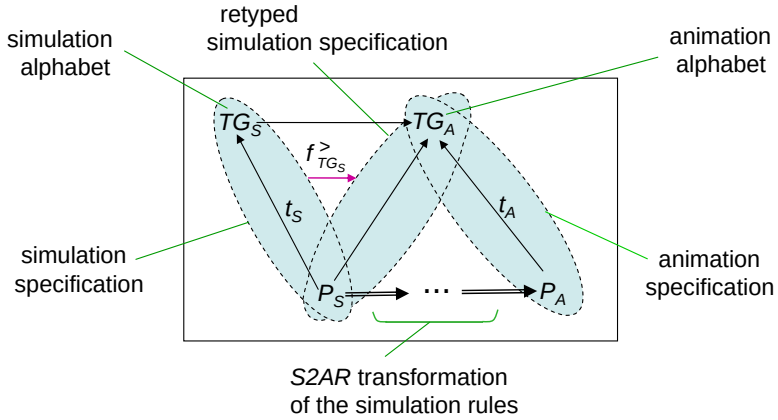
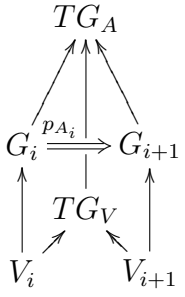


Figure 4.2: $S2AR$ Transformation from Simulation to Animation Specifications

Based on $S2AM$ and $S2AR$ transformation, an initial model state G_S is transformed to the corresponding state G_A in the animation view, and a (retyped) simulation specification $S = (TG_A, P_S)$ is transformed to a corresponding *animation specification* $A = (TG_A, P_A)$ (Def. 4.2.16). This combined transformation is called *simulation-to-animation model and rule transformation*, short $S2A$ transformation $S2A = (S2AM, S2AR)$. A transformation step $G_A \overset{p_A}{\Longrightarrow} H_A$ defined by the animation specification is called an *animation step*.

After the $S2A$ transformation has been completed, the set of animation scenarios in the *animation view* is defined by a restriction of each animation step in a transformation sequence over the animation specification A to the visualization alphabet TG_V (Def. 4.2.18 in Section 4.2.3). Such a restriction is shown in the diagram to the right where the graphs before and after each animation step $G_i \overset{p_{A_i}}{\Longrightarrow} G_{i+1}$ over A are restricted to graphs V_i and V_{i+1} in the animation view.



Please note that we do *not* restrict the animation rules to the visualization alphabet TG_V, since the resulting graph transformation system $V = (TG_V, P_V)$ would have a semantics allowing also transformations which cannot be constructed by restricting corresponding transformations defined by the animation specification. To get only transformations corresponding to $Trans(A)$, the semantics of the V would have to be restricted again to those transformations which can be constructed by restricting valid transformations from

$Trans(A)$. The same result is achieved directly in our procedure of restricting the transformation spans themselves to TG_V.

In the following we summarize the important properties of $S2A$ transformations which are shown in detail in Sections 4.3 - 4.6. Furthermore, we discuss limitations of our approach concerning the formal basis of our proofs.

1. **Formal Basis: Typed Graph Transformation Systems**
   Formally, simulation and animation specifications are defined as typed graph transformation systems (TGTS) and not as typed attributed graph transformation systems (TAGTS). In the formal correctness proofs, this enables us to use properties of TGTS embeddings, a notion relying on retyping functors between categories of graphs typed over different type graphs. These retyping functors are adjoint only for TGTS but not for TAGTS. As discussed in Section 2.1.3, the concept of retyping can be extended to node attributes by using attributes only as labels which are not changed during transformation. This kind of typed attributed graphs can be defined by ordinary typed graphs where potentially infinite sets of data values are considered as nodes. Hence, in our examples in Section 4.7, we use node attributes in this sense.

2. **Termination**
   The termination of $S2AM$ is shown using the general termination criteria from [EEPT06] for layered graph transformation systems. This termination result can be extended to show the termination of $S2AR$ transformation (Section 4.3).

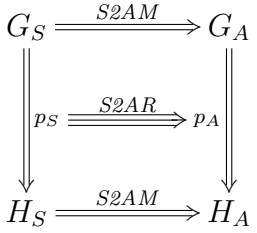
3. **Syntactical Correctness**
   Syntactical correctness of $S2AM$ means basically that all graphs G_A resulting from applying rules from Q to graphs G_S typed over TG_S, are typed over TG_A.

   A stronger notion of syntactical correctness would include functional behavior of the $S2A$ transformation. This means that in addition to the termination of $S2A$ transformation, local confluence has to be required in this case. Confluence of $S2A$ transformation is not shown in general in this thesis and would probably require at least local confluence of $S2AM$ transformation. For the specific examples in Section 4.7, we argue that the $S2A$ transformations are locally confluent.

   Provided we have syntactical correctness of $S2A$ transformation, then all possibly derivable graphs in the animation view are typed over TG_V due to the animation view construction: all graphs in transformations defined by the animation specification A are typed over TG_A and are restricted in the animation view construction to TG_V.

4. **Semantical Correctness and Completeness**
   Semantical correctness of $S2A$ transformation means that the transformation preserves the behavior defined by the original simulation specification.

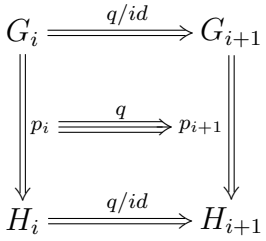
More precisely, we show that for each simulation step $G_S \stackrel{p_S}{\Longrightarrow} H_S$ and each $S2AR$ transformation $p_S \stackrel{Q\ !}{\Longrightarrow} p_A$ (where "!" denotes that the rules from Q are applied as long as possible), we have the $S2AM$ transformations $G_S \stackrel{Q\ !}{\Longrightarrow} G_A$ and $H_S \stackrel{Q\ !}{\Longrightarrow} H_A$, and the animation step $G_A \stackrel{p_A}{\Longrightarrow} H_A$. (see Theorem 4.5.11 in Section 4.5).



Note that the diagram for semantical correctness above is slightly different from the Mixed Semantical Confluence Diagram presented in [EE05a], as in our case the rules in P_A are not given a priori but constructed by the $S2AR$ transformation from the rules in P_S.

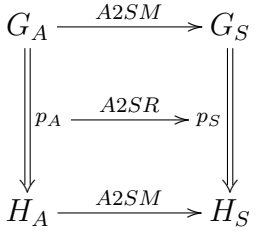
In order to show the semantical correctness stated above, *local semantical correctness* is shown first, which means the following: Let $p_i \stackrel{q}{\Longrightarrow} p_{i+1}$ be an intermediate $S2AR$ model transformation step which is part of the $S2AR$ transformation $p_S \stackrel{Q\ !}{\Longrightarrow} p_A$.

Then, for each graph transformation step $G_i \stackrel{p_i}{\Longrightarrow} H_i$, we have either graph transformation steps $G_i \stackrel{q}{\Longrightarrow} G_{i+1}$ or the identity $G_i \stackrel{id}{\Longrightarrow} G_{i+1}$, and we have either graph transformation steps $H_i \stackrel{q}{\Longrightarrow} H_{i+1}$ or the identity $H_i \stackrel{id}{\Longrightarrow} H_{i+1}$, such that there is the graph transformation step $G_{i+1} \stackrel{p_{i+1}}{\Longrightarrow} H_{i+1}$.



Local semantical correctness is shown in Theorem 4.5.7, and semantical correctness in Theorem 4.5.11.

To obtain semantical completeness of the $S2A$ transformation, we have to show that we have semantical correctness also in the backward direction.
Hence, in Section 4.6.1, we define an $A2S$ backward transformation $A2S : AnimSpec_{VL_A} \rightarrow SimSpec_{VL_S}$ given by $A2S = (A2SM : VL_A \rightarrow VL_S, A2SR : P_A \rightarrow P_S)$ which is *semantically correct* if for each animation step $G_A \stackrel{p_A}{\Longrightarrow} H_A$ with $G_A, H_A \in VL_A$ and $A2SM(G_A) = G_S$ and $A2SM(H_A) = H_S$, there is a corresponding simulation step $G_S \stackrel{p_S}{\Longrightarrow} H_S$ with $A2SM(H_A) = H_S$ (see Theorem 4.6.8).



We call an $S2A$ transformation *semantical equivalence* of $SimSpec_{VL_S}$ and $AnimSpec_{VL_A}$ if $S2A$ is semantically correct and complete, and we have in addition that $A2S \circ S2A = Id$ and $S2A \circ A2S = Id$, for the $A2S$ backward transformation $A2S$ of $S2A$. In Section 4.6.3, we give criteria for such a semantical equivalence.

The last step of the animation view definition only restricts the derived graphs in a transformation but does not change the used rules of the animation specification.

Hence, this step has no influence on the semantical correctness and completeness of the resulting model in the animation view.

4.2 Animation View Construction

In this section, we describe the construction of an animation specification from a simulation specification by $S2A$ transformation according to Figures 4.1 and 4.2 in more detail, using the simulation specification of the Echo model from Section 3.2.3 as running example.

In Section 4.2.1, the integration of a simulation alphabet TG_S and a domain-specific visualization alphabet TG_V (also called *animation view*) is defined, yielding an *animation alphabet* $TG_A = TG_S +_{TG_I} TG_V$ (Def. 4.2.4). In Section 4.2.2, the characteristics of $S2A$ *transformation rules* are discussed, which define how the symbols used in the simulation rules in P_S are extended by domain-specific visualization symbols typed over TG_V (Def. 4.2.8). We then give a definition to apply non-deleting rules to arbitrary rules (Def. 4.2.12) which is the formal basis for the application of $S2A$ transformation rules to simulation rules. The result of this $S2A$ rule transformation is the *animation specification* A. Section 4.2.3 defines animation scenarios in the animation view by restricting the transformations over the animation specification to the animation view, and introduces animation operations for defining continuous state changes.

4.2.1 Integration of Simulation and Visualization Alphabets

The model transformation rules adding domain-specific symbols to the simulation specification, have to be typed over an integrated alphabet including the simulation alphabet and a visualization alphabet (the *animation view*). Hence, as a first step towards animation, an animation view consisting of symbols and links representing the application domain, has to be defined. In order to be able to define the model transformation rules by a layered graph transformation system, it is important that the animation view (which is an alphabet and hence modeled by a type graph) does not contain cycles. Therefore, we distinguish different layers of symbol types in animation views where symbols of a higher layer are connected to symbols of lower layers only, i.e. there exist directed edges only in the direction from types of layer l to types of layer k with $k \leq l$.

Definition 4.2.1 (*Layered Alphabet*)
A *layered alphabet* (TG, tl) with $tl = (tl_V, tl_E)$ is a type graph $TG = (V, E, src, tar)$ together with two layering functions $tl_V : V \to \mathbb{N}$ and $tl_E : E \to \mathbb{N}$, mapping each symbol type node $v \in V$ to a type layer $tl_V(v)$ and each link type edge $e \in E$ to a type layer $tl_E(e)$, with $0 \leq tl_V(v), tl_E(e) \leq n$, and n is the number of type layers. $\triangle$

Definition 4.2.2 (*Interface Type Graph TG_I and Animation View A_V*)
Let TG_V be the type graph of an application domain (called *animation view type graph*), and $S = (TG_S, P_S)$ be a simulation specification. A type graph TG_I is called *interface type graph of TG_S and TG_V* if TG_I is the intersection of TG_V and TG_S such that there are two type graph inclusions $TG_V \overset{t_V}{\hookleftarrow} TG_I \overset{t_S}{\hookrightarrow} TG_S$.

Let $A_V = (TG_V, tl_V)$ with $tl_V = (tl_{V_V}, tl_{V_E})$ be a layered alphabet acc. to Def. 4.2.1, and TG_I the interface type graph of TG_S and TG_V.

A_V is called *animation view* of S, if the following layering conditions are satisfied:

1. the interface types in A_V have the type layer number 0: $tl_{V_V}(v) = 0$ and $tl_{V_E}(e) = 0$ for all symbols $v \in tl_{V_V}(TG_I)$ and links $e \in tl_{V_E}(TG_I)$.
2. a symbol type in A_V is connected by directed links e only to symbol types of the same layer or to symbol types of lower layers: $\forall e \in E_{TG_V} : tl_{V_V}(tar(e)) \leq tl_{V_V}(src(e))$
3. the type layer of a link type e is equal to the type layer of the source symbol type of the link type: $\forall e \in E_{TG_V} : tl_{V_E}(e) = tl_{V_V}(src(e))$

△

Please note that in the following we do not always distinguish between the notions "alphabet" and "type graph" if the layering function is not important.

Example 4.2.3 (*Animation View for the Echo Model*)
The idea to animate the model of the echo algorithm introduced in Section 3.2.3, is to show the states of the model as host network graph where the nodes correspond to the hosts and the edges correspond to busy communication channels between them. We only show a channel edge if a message is sent from a sending host (the source node of the channel edge) to a receiving host (the target node of the edge). A host who has received all messages from his target group is marked ready by a red frame around his host node. The algorithm terminates successfully when the boss node is marked ready. By assigning animation view symbols only to those states of the net which concern the sending of a message, the animation of the system behavior cuts out details that have to be present in the Petri net but make it rather difficult to validate the behavioral requirements.

The animation view $AV_{Echo} = (TG_V, tl_V)$, shown on the right of Fig. 4.3, represents a network of hosts. It contains the *interface type* Net, which is also present in the simulation alphabet TG_S for C/E nets, shown on the left of Fig. 4.3. A Host symbol type is attributed by its name and depicted at the concrete syntax level as rectangle with text inside. A Message from one Host to another one is visualized as arc between the two Hosts. The ready flag is a marker for a Host which has finished sending messages and is depicted by

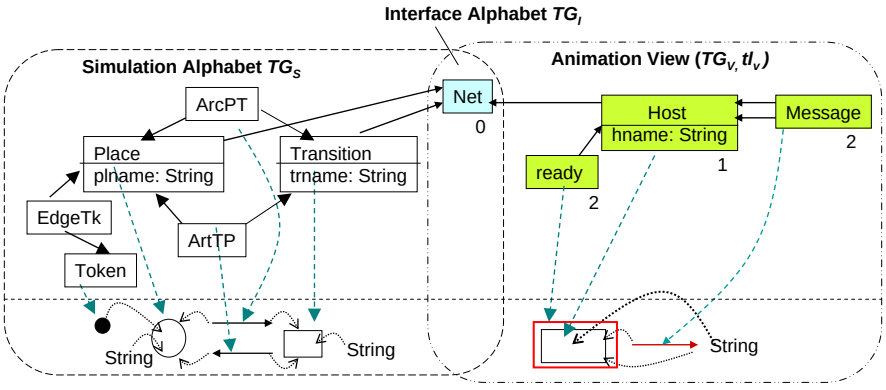
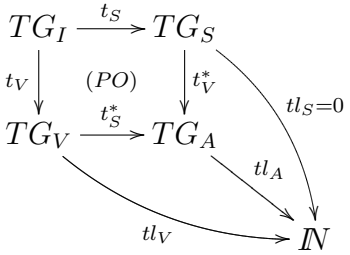


Figure 4.3: Animation Alphabet TG_A for the Echo Model

a red frame around the Host's rectangle shape. The type layers for TG_V are defined as follows: the interface type Net gets the layer number 0. The symbol type Host and the link type between Host and Net belong to layer 1, and the remaining symbol types ready, Message and their adjacent edges are types of layer 2 (as indicated by index numbers at the type nodes in the animation view part of Fig. 4.3. Obviously, the layering conditions are satisfied, as all symbol types are linked only to symbol types of a lower layer, and all link types belong to the same layer as their source symbol types. △

Definition 4.2.4 (*Animation Alphabet* A_A)
Let $A_V = (TG_V, tl_V)$ be the animation view, TG_S the simulation alphabet, and TG_I the interface type graph of TG_V and TG_S. The *animation alphabet* $A_A = (TG_A, tl_A)$ is a layered alphabet, defined as follows:
The type graph $TG_A = TG_S +_{TG_I} TG_V$ is constructed as pushout of TG_S and TG_V over TG_I with the injective morphisms $TG_V \xleftarrow{t_V} TG_I \xhookrightarrow{t_S} TG_S$ in **Graphs**, as shown in the diagram to the right. To extend the layering of types from the animation view A_V to the complete animation alphabet A_A, we assign the layer number 0 to all types in TG_A with an origin in TG_S (types in TG_I have the layer number 0 already, according to Def. 4.2.2):



- $\forall v, e \in t_V^*(TG_S) : tl_{A_V}(v) = 0$ and $tl_{A_E}(e) = 0$
- $\forall v, e \in t_S^*(TG_V) : tl_{A_V}(v) = tl_{V_V}(v)$ and $tl_{A_E}(e) = tl_{V_E}(e)$

The layering is well-defined, since $tl_S \circ t_S = tl_V \circ t_V$. △

The forward retyping $f^{>}_{TG_S}(S)$ of simulation specification $S = (TG_S, P_S)$ by the forward retyping functor $f^{>}_{TG_S} : \mathbf{Graphs_{TG_S}} \rightarrow \mathbf{Graphs_{TG_A}}$, leads to the retyped simulation specification (TG_A, P_S) which is behaviorially equivalent to S as the simulation

rules remain identical. From now on, by S we mean the retyped simulation specification $S = (TG_A, P_S)$.

Example 4.2.5 (*Animation Alphabet for the Echo Model*)
The type graph TG_A of the animation alphabet A_A consists of the composition of the simulation type graph TG_S for C/E nets and the animation view type graph TG_V for a network of hosts. The complete graph in Fig. 4.3 shows TG_A for the Echo model, with the node symbol type Net being the only element of the interaction type graph TG_I. The type layer number for the node types Net, Place, Transition, ArcPT, ArcTP, EdgeTk and Token, and for their adjacent edges is 0. The type layer number for the animation view symbol and link types are the type layers defined in Example 4.2.3. △

4.2.2 *S2A* Transformation

In this section, we define *S2A* transformation rules. The use of *S2A* transformation rules is twofold: On the one hand, they define *S2A model transformation* from graphs occurring in a simulation scenario to graphs in the corresponding animation view; on the other hand they are used for *S2A rule transformation* from a simulation specification S into an animation specification A.

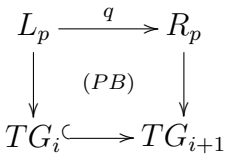
S2A transformation rules are non-deleting rules adding only symbols typed over TG_V. Moreover, *S2A* model transformation rules are assigned to rule layers which provide a controlled rule application.

Definition 4.2.6 (*Layered Graph Transformation System*)
A TG-typed graph transformation system with rules P is called *layered graph transformation system* if for each rule $p \in P$ we have a rule layer $rl(p) = k$ with $0 \leq k \leq n (k, n \in \mathbb{N})$ where n is the number of layers.

Layered graph transformation based on a layered graph transformation system is performed in the following way: Starting with rule layer number $k = 0$, all rules p with $rl(p) = k$ are applied as long as possible. After termination of all rules in the current layer, the transformation continues with the next layer ($k = k + 1$). △

Definition 4.2.7 (*Layered Type-Increasing Graph Transformation System*)

A *layered graph transformation system* (TG, P) is called *type-increasing* if we have disjoint layers $P_0, .., P_n$ with $\bigcup_{i=0}^{n} P_i = P$, and type graphs $TG_0 \subseteq TG_1 .. \subseteq TG_n$ such that for each $p : L_p \rightarrow R_p \in P_i$, L_p is typed over TG_i, R_p is typed over TG_{i+1}, and $R_p|_{TG_i} = L_p$, i.e. the diagram to the right is a pullback.



△

Definition 4.2.8 (*S2A Transformation System*)
Let $A_A = (TG_A, tl_A)$ be an animation alphabet according to Def. 4.2.4. An *S2A transformation system* $S2A = (TG_A, Q)$ is a layered type-increasing graph transformation system, consisting of a set Q of layered TG_A-typed non-deleting rules $q = (L_q \xrightarrow{q} R_q)$ with $NAC_q = (L_q \xrightarrow{n} N_q)$. The *rule layer* of an $S2A$ transformation rule q is computed depending on the type layer function tl_A of the animation alphabet A_A: the rule layer $rl(q)$ of $q \in Q$ which creates symbols and links of type layer $tl_A = i$ is computed by $rl(q) = i - 1$.

Additionally, $S2A$ transformation rules have have the following properties:

1. the left-hand side L_q is never empty;
2. there is an isomorphism $n'_q : N_q \to R_q$ such that the following diagram commutes:

$$N_q \xleftarrow[n_q]{} L_q \xrightarrow[q]{} R_q, \quad N_q \xrightarrow{n'_q} R_q$$

3. all $S2A$ transformation rules belonging to the same rule layer are parallel independent.

△

Remarks:

- Due to $S2A$ transformation rules being type-increasing, each q creates objects that are typed over $TG_V - TG_I$ only, since L_q is never empty, and R_q contains only objects of type layers greater than those of the L_q (i.e. at least of type layer 1). Moreover, due to being non-deleting, $S2A$ transformation rules preserve at least all objects typed over TG_S. Hence, $f^{<}_{TG_S}(L_q \xrightarrow{q} R_q) = id$, which implies that $L_q|_{TG_S} = R_q|_{TG_S}$.
- The parallel independence of rules in the same layer implies that for q, q' with $rl(q) = rl(q') = i$ we get the parallel rule $(q + q')$ such that $(R_q + R_{q'})|_{TG_i} = (L_q + L_{q'})$, and we can assume that in each layer at most one parallel rule is applied.
- The NAC $N_q \cong R_q$ forbids the application of an $S2A$ model transformation rule more than once at the same essential match. Informally, an essential match m_0 of a match $m_1 : L_q \to G'$ for a transformation $G \overset{*}{\Longrightarrow} G'$ with $G \subseteq G'$ means that m_1 can be restricted to $m_0 : L_q \to G$. This is an important restriction which is needed to ensure termination of $S2A$ model transformation (see Section 4.3).

□

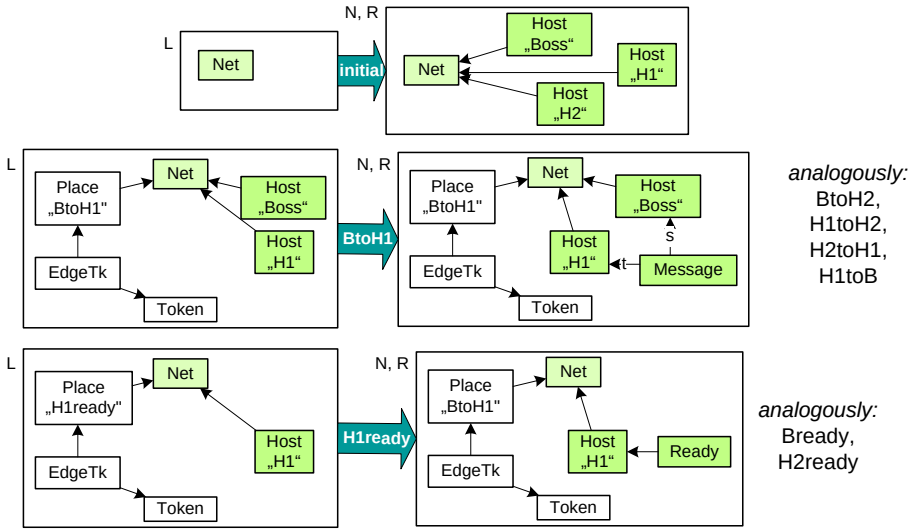


Figure 4.4: $S2A$ Transformation Rules for the Echo Model

Example 4.2.9 (*$S2A$ Transformation Rules for the Echo Model*)
On the basis of the animation alphabet given in Example 4.2.3, we can now define the $S2A$ transformation rules for the Echo model.

Rule initial adds the symbols for the host nodes, Host “Boss”,Host “H1”, and Host “H2” to the net. Rule BtoH1 (and analogous rules) adds a Message edge between two host nodes if the corresponding place is marked in the Petri net. Rule H1ready (and analogously, rule Bready and rule H2ready mark a host with a ready-flag in the case that the corresponding place is marked. Each rule has a NAC N, which is isomorphic to the right-hand rule side. Rule initial has rule layer number 0, as the types of the symbols added by the rule have the layer number 1. All other rules are assigned to rule layer 1, as they add symbols typed over types of layer 2. △

Definition 4.2.10 (*$S2AM$ Transformation*)
Let $SimSpec_{VL_S} = (VL_S, P_S)$ be a simulation specification according to Def. 3.2.3, with VL_S typed over TG_S. Let TG_A be an animation type graph with $TG_S \subseteq TG_A$. Then a *simulation-to-animation model transformation*, short *$S2AM$ transformation*, $S2AM : VL_S \to VL_A$, is given by $S2AM = (VL_S, TG_A, Q)$, where (TG_A, Q) is an $S2A$ transformation system according to Def. 4.2.8. An *$S2AM$ transformation sequence* is denoted by $G_S \overset{Q\ !}{\Longrightarrow} G_A$ with $G_S \in VL_S$, meaning that transformation steps $G_1 \overset{q}{\Longrightarrow} G_2$ with $q \in Q$ are applied as long as possible, starting with $G_1 = G_S$. The *animation language* VL_A is defined by $VL_A = \{G_A | \exists G_S \in VL_S \wedge G_S \overset{Q\ !}{\Longrightarrow} G_A\}$. This means $G_S \overset{Q\ !}{\Longrightarrow} G_A$ implies $G_S \in VL_S$ and $G_A \in VL_A$, where each intermediate step $G_i \overset{q_i}{\Longrightarrow} G_{i+1}$ is called *$S2AM$ step*. △

Example 4.2.11 (*$S2AM$ Transformation of the Initial Echo Model State*)
The initial state of the Echo net (see Fig. 3.11) has three tokens on the places B_idle, H1_idle and H2_idle. The only $S2A$ transformation rule applicable to the graph $G_{initial}$ modeling this initial state, is rule initial. Hence, the only possible $S2A$ transformation $G_{initial} \overset{Q\ !}{\Longrightarrow} H_{initial}$ consists of one transformation step, namely the application of rule initial to graph $G_{initial}$. The rule is applicable to $G_{initial}$ only once, as there exists exactly one match, and the NAC of rule initial equals its right-hand side. Fig. 4.5 shows the $S2A$ transformation $G_{initial} \overset{Q\ !}{\Longrightarrow} H_{initial}$.

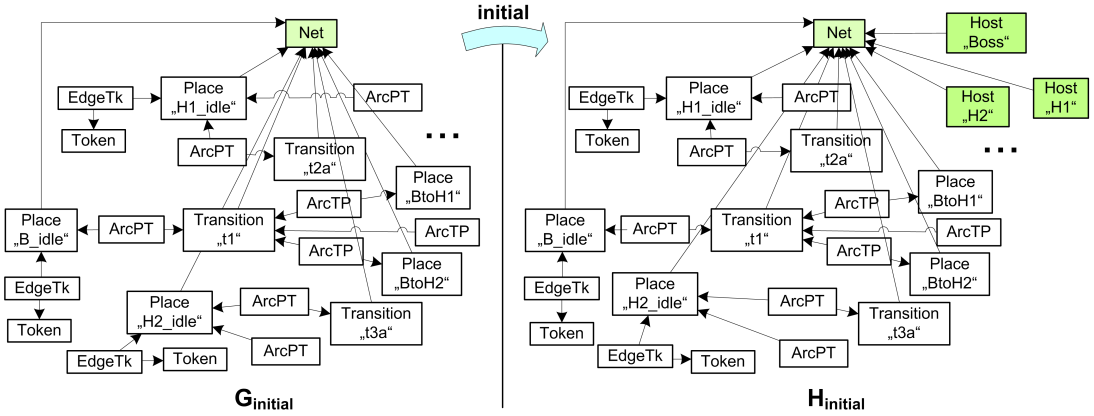


Figure 4.5: $S2AM$ Transformation of the Initial State of the Echo Model

Rule initial adds the symbols for the host nodes, Host “Boss”,Host “H1”, and Host “H2” to the net graph. Due to space limitations, only parts of the abstract syntax graphs of $G_{initial}$ and $H_{initial}$ are shown in Fig. 4.5.

△

In order be able to apply $S2A$ transformation rules not only to graphs but also to the simulation rules, we need a construction defining how to apply non-deleting rules to rules. The following definition extends the construction for rewriting rules by rules given by Parisi-Presicce in [PP96], where a rule q is only applicable to another rule p if it is applicable to the interface graph of p. This means, q cannot be applied if p deletes or generates objects which q needs. In this thesis, we want to add animation symbols to simulation rules even if the $S2A$ transformation rule is *not* applicable to the interface of the simulation rule. Hence, we distinguish four cases in Def. 4.2.12. Case (1) corresponds to the notion of rule rewriting in [PP96], adapted to non-deleting $S2A$ transformation rules. In Case (2), the $S2A$ transformation rule q is not applicable to the interface, but only to the left-hand side of a rule p (p deletes something that is needed by q), and in Case (3), q is only applicable to the right-hand side of p (p generates something that q needs). Case (4) covers the case that q is applicable only to a NAC of p, but to none of its other rule graphs.

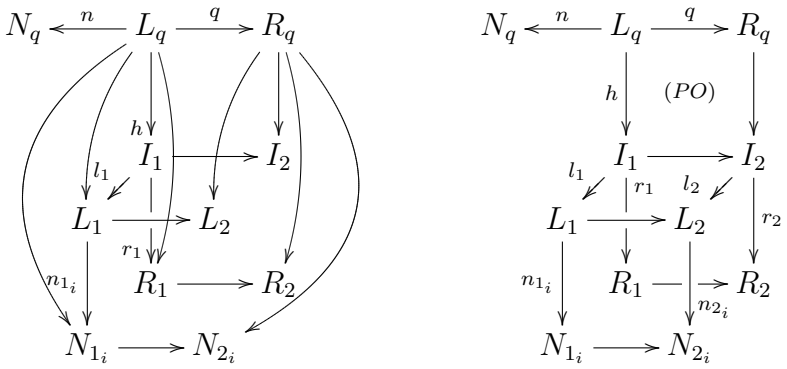
Definition 4.2.12 (*Transformation of Rules by Non-Deleting Rules*)
Given a non-deleting rule $q = (L_q \xrightarrow{q} R_q)$ with $NAC_q = (L_q \xrightarrow{n} N_q)$, and a rule $p_1 = (L_1 \xleftarrow{l_1} I_1 \xrightarrow{r_1} R_1)$ with $NAC_{1_i} = (L_1 \xrightarrow{n_{1_i}} N_{1_i}), (i = 1, .., n)$. Then q is applicable to p_1 leading to a rule transformation step $p_1 \overset{q}{\Longrightarrow} p_2$, if the precondition of one of the following four cases is satisfied and $p_2 = (L_2 \xleftarrow{l_2} I_2 \xrightarrow{r_2} R_2)$ with $NAC_{2_i} = (L_2 \xrightarrow{n_{2_i}} N_{2_i})$ is defined according to the corresponding construction:

- *Case (1)*:

  Precondition (1): There is a match $L_q \xrightarrow{h} I_1$, such that the matches $L_q \xrightarrow{h} I_1, L_q \xrightarrow{l_1 \circ h} L_1, L_q \xrightarrow{r_1 \circ h} R_1$ and $L_q \xrightarrow{n_{1_i} \circ l_1 \circ h} N_{1_i}$ satisfy NAC_q for $i = 1, .., n$.

  Construction (1): First, q is applied to I_1 via match $L_q \xrightarrow{h} I_1$. This results in the pushout $R_q +_{L_q} I_1 = I_2$. Secondly, q is applied to L_1 via match $L_q \xrightarrow{l_1 \circ h} L_1$. This results in the pushout $R_q +_{L_q} L_1 = L_2$. Thirdly, q is applied to all NAC-graphs N_{1_i} via the matches $L_q \xrightarrow{n_{1_i} \circ l_1 \circ h} N_{1_i}$, resulting in the pushouts $R_q +_{L_q} N_{1_i} = N_{2_i}$ Finally, q is applied to R_1 via match $L_q \xrightarrow{r_1 \circ h} R_1$, resulting in the pushout $R_q +_{L_q} R_1 = R_2$.

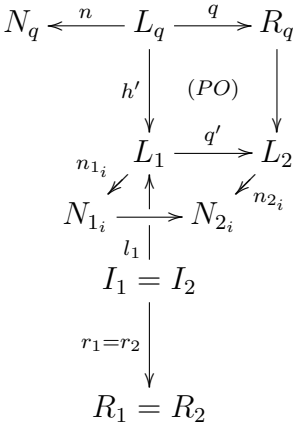
  These transformation steps are depicted in the left diagram below. Due to Property A.11, we get the morphisms $I_2 \xrightarrow{l_2} L_2$, $I_2 \xrightarrow{r_2} R_2$ and $L_2 \xrightarrow{n_{2_i}} N_{2_i}$ such that all squares in the right diagram below are pushouts, and we get the transformed rule $p_2 = (N_{2_i} \xleftarrow{n_{2_i}} L_2 \xleftarrow{l_2} I_2 \xrightarrow{r_2} R_2)$.



- *Case (2)*:

  Precondition (2): Precondition (1) is not satisfied, but there is a match $L_q \xrightarrow{h'} L_1$ such that the matches $L_q \xrightarrow{h'} L_1$ and $L_q \xrightarrow{n_{1_i} \circ l_1 \circ h'} N_{1_i}$ satisfying NAC_q for $i = 1, .., n$.

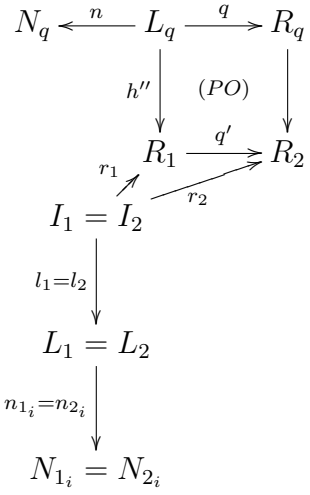
  Construction (2): In this case, q is applied to L_1 and N_{1_i} only. The application yields two pushouts (the two squares in the diagram to the right, constructed in analogy to Case (1)). The interface I_2 and the right-hand side R_2 of p_2 are the unchanged interface and right-hand side of p_1, leading to $p_2 = (N_{2_i} \xleftarrow{n_{2_i}} L_2 \xleftarrow{l_2} I_2 \xrightarrow{r_2} R_2)$ with $l_2 = q' \circ l_1$.



- *Case (3)*:

  Precondition (3): Preconditions (1) and (2) are not satisfied, but there is a match $L_q \xrightarrow{h''} R_1$ which satisfies NAC_q.

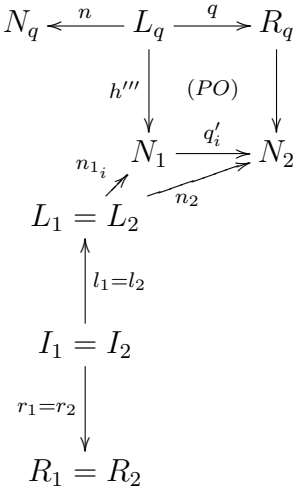
  Construction (3): In this case, q is applied to R_1 only. The result of the application of q to R_1 is illustrated in the diagram to the right, where R_2 is the object resulting from the direct transformation via q of R_1, and I_2, L_2 and N_{2_i} are just the unchanged interface, left-hand side and NACs of p_1, leading to $p_2 = (N_{2_i} \xleftarrow{n_{2_i}} L_2 \xleftarrow{l_2} I_2 \xrightarrow{r_2} R_2)$ with $r_2 = q' \circ r_1$.



- *Case (4)*:

  Precondition (4): Preconditions (1) - (3) are not satisfied, but are matches $L_q \xrightarrow{h'''_i} N_{1_i}$ which satisfy NAC_q.

  Construction (4): In this case, q is applied to N_{1_i} only. The result of the application of q to N_{1_i} is illustrated in the diagram to the right, where N_{2_i} is the object resulting from the direct transformation via q of N_{1_i}, $L_2 = L_1, I_2 = I_1$ and $R_2 = R_1$ are just the unchanged left-hand side, interface and right-hand side of p_1 leading to $p_2 = (N_{2_i} \xleftarrow{n_{2_i}} L_2 \xleftarrow{l_2} I_2 \xrightarrow{r_2} R_2)$ with $n_{2_i} = q'_i \circ n_{1_i}$.



△

Remarks:

- In Case (1), q adds objects to all rule graphs of p_1. by p_1. Moreover, the resulting rule p_2 preserves these newly added objects as the morphisms in $p_2 = (N_{2_i} \xleftarrow{n_{2_i}} L_2 \xleftarrow{l_2} I_2 \xrightarrow{r_2} R_2)$ contain mappings between all objects that are added to all three graphs by q.
- In Case (2), q adds objects only to the left-hand side (and the NACs) of p_1. Thus, the resulting rule p_2 deletes the newly added objects.
- In Case (3), q adds objects only to the right-hand side of p_1. Thus, the resulting rule p_2 generates these objects added by q.
- In Case (4), q adds objects only to the NACs of p_1. The resulting rule p_2 equals p_1 but is applicable in a larger context than p_1. Note that Case (4) is independent of Case (3), i.e. the same rule rule q might be applied first to the NACs and then to the right-hand side of p_1, or vice versa, leading to the same result.

□

In order to be able to show confluence of rewriting rules by non-deleting rules acc. to Def. 4.2.12, we might need additional conditions. We want to be able to show that the application of a set Q of non-deleting rules as long as possible to an arbitrary rule p_1 results in a unique transformed rule p_2.

Let us consider a conflict between two rules $q_1, q_2 \in Q$, where two rules q_1 and q_2 can be applied to a current rule p_1 in such a way that the respective other rule cannot be applied after the application of the first rule. Let us assume that q_1 should to be applied to p_1 according to Case (1), and rule q_2 according to Case (2) In this case, we define that Case(1) has a higher match priority, and that from the set of all applicable rules q_i, one of the rules with the highest match priority is applied next.

Definition 4.2.13 (*Match Priorities of Non-Deleting Rules*)
Let Q be a set of non-deleting transformation rules. All rules $q \in Q$ which can be applied to rule p according to Case(i) of Def. 4.2.12, with $i \in \{1, .., 4\}$, get the match priority i. All rules $q \in Q$ which cannot be applied to rule p get the match priority 5. The match priority with the smallest number is called *highest match priority*. △

Note that according to Def. 4.2.13, all rules in Q which are not applicable to p at all, get the match prirority 4, e.g. the lowest possible match priority.

We call the application of rules in Q to rules $p \in P$ according to the cases of Def. 4.2.12 such that the match priorities are respected, $S2AR$ transformation, to distinguish this form of controlled application of rules to rules from applying the rules in Q to graphs in the usual way of graph transformation ($S2AM$ transformation).

Definition 4.2.14 (*$S2AR$ Transformation*)
Let $\mathbf{SimSpec_{VLs}} = (VL_S, P_S)$ be a simulation specification according to Def. 3.2.3. Let $S2AM = (VL_S, TG_A, Q)$ be an $S2AM$ transformation according to Def. 4.2.10. Then a *simulation-to-animation rule transformation*, short *$S2AR$ transformation*, $S2AR : P_S \rightarrow P_A$, is given by $S2AR = (P_S, TG_A, Q)$, where (TG_A, Q) is an $S2A$ transformation system, and an $S2AR$ transformation sequence $p_S \overset{Q\ !}{\Longrightarrow} p_A$, with $p_S \in P_S$, is computed according to the following algorithm:

1. initialize index $i = 1$ and $p_i = p_S$;
2. compute the match priorities for all rules in Q respecting their application to p_i acc. to Def. 4.2.13;
3. if all rules have the match priority 5, the algorithm stops, and the resulting rule $p_A = S2AR(p)$ is equal to rule p_i;
4. if there are rules with a higher match priority than 5, apply one of the rules q with the highest match priority to p_i acc. to Def. 4.2.12, resulting in rule p_{i+1};
5. increase $i = i + 1$ and return to step 2.

An *$S2AR$ transformation sequence* $p_S \overset{Q\ !}{\Longrightarrow} p_A$ consists of rule transformation steps $p_i \overset{q_i}{\Longrightarrow} p_{i+1}$ where $S2A$ transformation rules $q_i \in Q$, are applied as described above, starting with $p_1 = p_S$. The set of *animation rules* P_A is defined by $P_A = \{p_A | \exists p_S \in VL_S \wedge p_S \overset{Q\ !}{\Longrightarrow} p_A\}$. This means $p_S \overset{Q\ !}{\Longrightarrow} p_A$ implies $p_A \in P_A$, where each intermediate step $p_i \overset{q_i}{\Longrightarrow} p_{i+1}$ is called *$S2AR$ step*.

△

Note that conflicts between rules with the same match priority can still occur and may lead to a non-confluent rule transformation system. A possible solution would be to define additional priorities for the rules in Q. The confluence of $S2AR$ transformation has to be shown for the specific $S2AR$ transformation sequences explicitly and does not hold in general. $S2AR$ transformation is nondeterministic in general, but should become deterministic in the sense of local confluence. This remains to be shown for concrete $S2AR$ transformation examples. Nevertheless, the construction of the $S2AR$ transformation in Def. 4.2.14 reduces the number of possible conflicts and makes it easier to show confluence for specific $S2A$ transformations.

Example 4.2.15 (*$S2AR$ Transformation of an Echo Model Simulation Rule*)
We apply the $S2A$ transformation rules from Example 4.2.9 to the (retyped) simulation

rule t1 in Fig. 3.13. At first, *S2A* transformation rule initial is applied, as it is the only model transformation rule in layer 0. Its application adds the context, i.e. the three host symbols, to all rule graphs of simulation rule t1. As rule initial is applicable to the interface of simulation rule t1, this first rewriting step conforms to Case (1) of Def. 4.2.12. The construction of the first rule transformation step from rule t1 to rule t1' according to Case (1) is depicted in Fig. 4.6. Note that we draw only one of the three NACs of rule t1 in Fig. 4.6. The application changes the other two NACs N_{1-2} and N_{1-3} analogously to N_{1-1} by adding the context symbols.

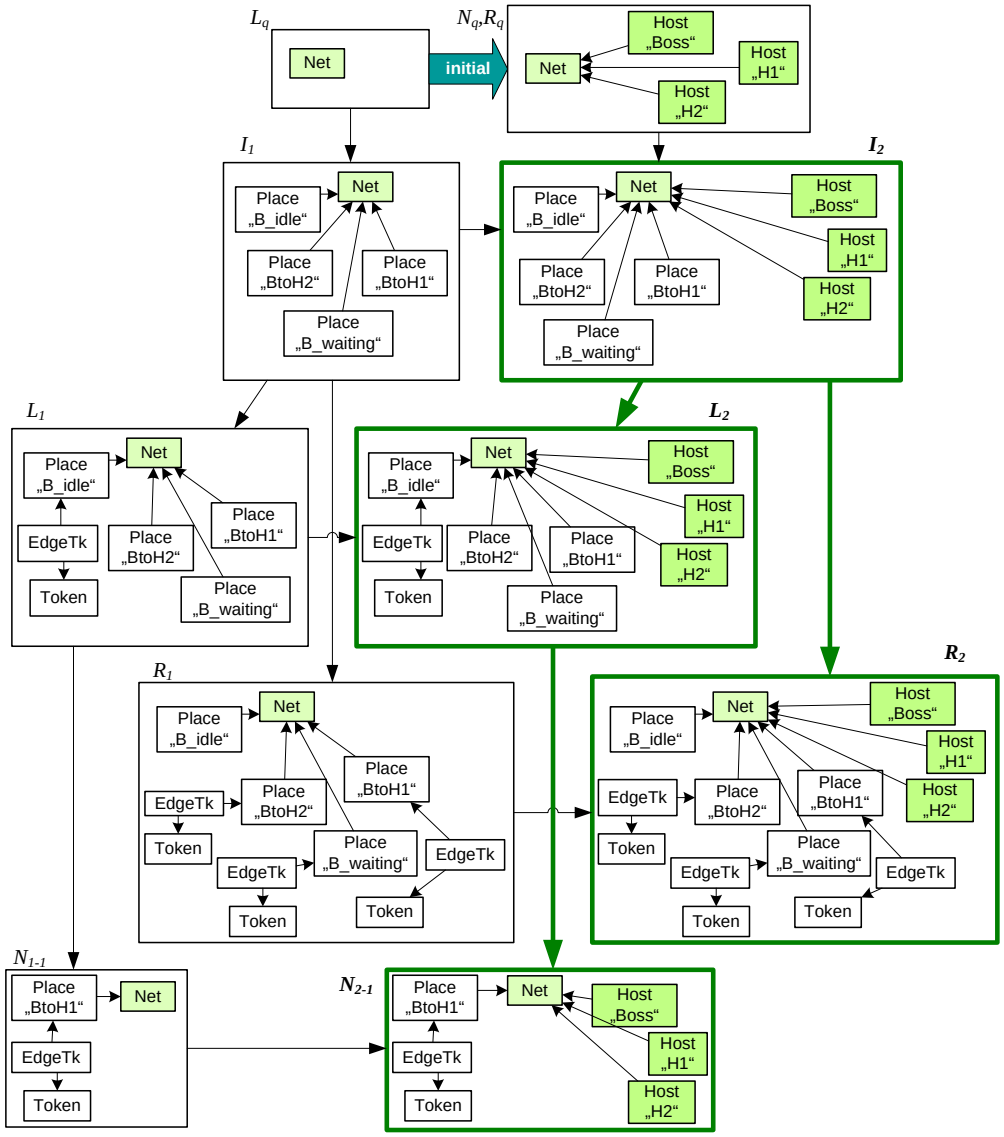


Figure 4.6: Application of Rule initial to Simulation Rule t1

After the first application of *S2A* transformation rule initial, the rule initial is not applicable any more to the transformed rule t1'. The NAC corresponding to the right-hand side

of rule initial is not satisfied, because rule t1' has already got three host nodes in all rule graphs. Hence, Case (5) from Def. 4.2.12 has been reached for rule initial. But there are still other $S2A$ transformation rules in Example 4.2.9 belonging to the next rule layer, which are applicable to the current rule t1'. As second model transformation step we apply $S2A$ transformation rule BtoH1 to rule t1'. Rule BtoH1 is not applicable to the interface of rule t1', and not to its left-hand side. But it is applicable to the right-hand side (Case (3)), and to the NAC N_{1-1} (Case (4)). These transformations are independent of each other, and can be realized in arbitrary order. Fig. 4.7 shows the application of rule BtoH1 to rule t1', changing its right-hand side according to Case $(23$, which results in rule t1".

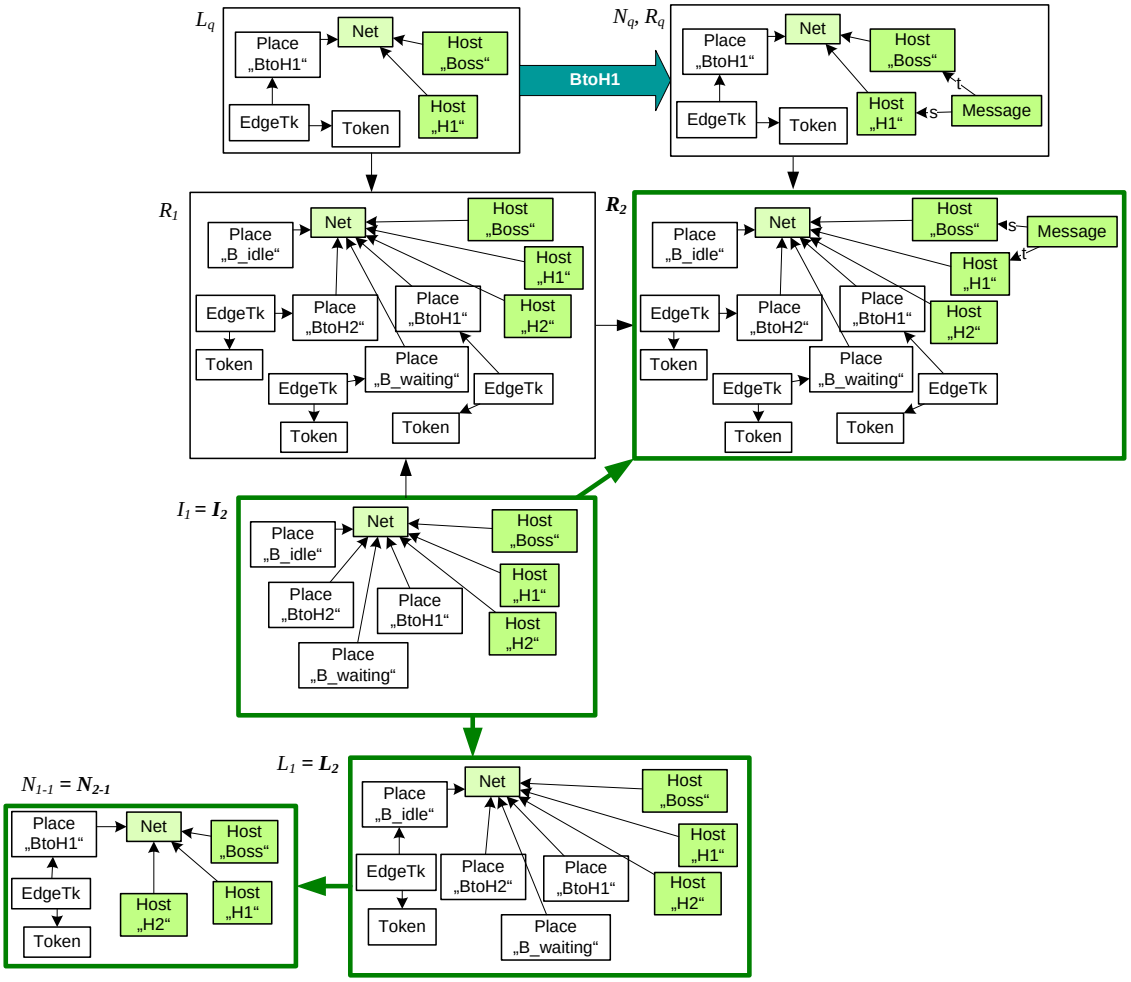


Figure 4.7: Application of Rule BtoH1 to Rule t1'

Next, rule BtoH1 is applied to t1" at the match into the NAC N_{1-1} resulting in rule t1"' (not depicted). After this transformation step, rule BtoH1 is not applicable any more to t1"'. The only model transformation rule applicable to t1"' is rule BtoH2 which can be applied analogously to BtoH1, both to the right-hand side and to NAC N_{1-3}. This leads to two more

transformation steps (not depicted) after which rule t1 has been transformed completely, i.e. no more model transformation rules are applicable to the resulting rule.

△

Definition 4.2.16 (*Animation Specification and $S2A$ Transformation*)
Let $\mathbf{SimSpec_{VL_S}} = (VL_S, P_S)$ be a simulation specification according to Def. 3.2.3. Let $S2AM = (VL_S, TG_A, Q) : VL_S \rightarrow VL_A$ be an $S2AM$ transformation according to Def. 4.2.10, and $S2AR = (VL_S, TG_A, Q) : P_S \rightarrow P_A$ be an $S2AR$ transformation according to Def. 4.2.14.

Then,

1. $\mathbf{AnimSpec_{VL_A}} = (VL_A, P_A)$ is called *animation specification*, and each step $G_A \overset{p_A}{\Longrightarrow} H_A$ with $G_A, H_A \in VL_A$ and $p_A \in P_A$ is called *animation step*.
2. $S2A : \mathbf{SimSpec_{VL_S}} \rightarrow \mathbf{AnimSpec_{VL_A}}$, defined by $S2A = (S2AM, S2AR)$ is called *simulation-to-animation model and rule transformation*, short $S2A$ transformation.

△

Remark: We will discuss important properties of $S2A$ transformation, namely termination, syntactical and semantical correctness and completeness, in Sections 4.3 - 4.6. □

Example 4.2.17 (*Animation Specification of the Echo Model*)
The subsequent application of the $S2A$ transformation rules from Example 4.2.9 according to Def. 4.2.14 to each of the simulation rules of the Echo model (some of them are shown in Fig. 3.13), results in the set of P_A of *animation rules*. The animation rule which corresponds to simulation rule t1 from Fig. 3.13 is shown in Fig. 4.8.

△

4.2.3 Continuous Animation in the Animation View

The animation specification resulting from an $S2A$ transformation still contains all language elements from the original simulation specification. In general, it is preferable to visualize scenarios only in their animation view, i.e. restricted to the animation view TG_V.

Definition 4.2.18 (*Animation Scenario in Animation View*)
Let (TG_V, tl_V) be an animation view according to an application domain. Let $S = (VL_S, P_S)$ be a simulation specification, and $A = (VL_A, P_A)$ be the animation specification constructed from S by $S2AR$ transformation. Let $H_0 \in VL_A$ be a graph of the

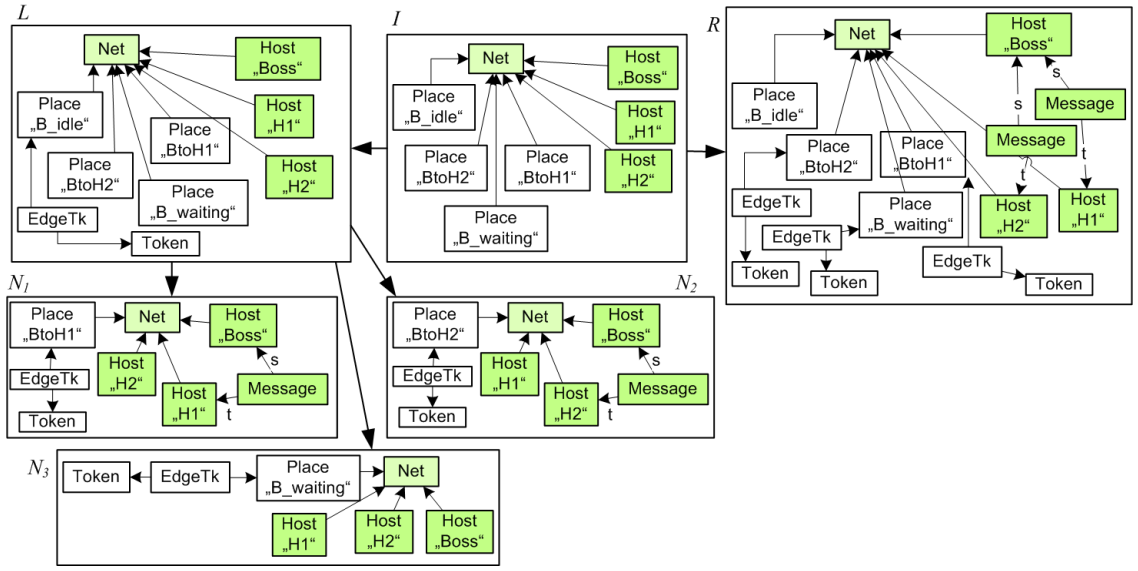
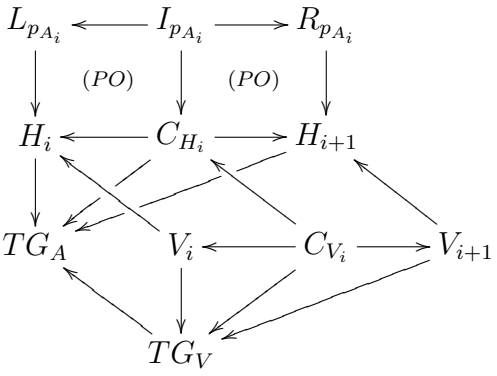


Figure 4.8: Animation Rule for Simulation Rule t1

animation language VL_A, i.e. there exists an $S2AM$ transformation $G_0 \overset{Q\ !}{\Longrightarrow} H_0$ with $G_0 \in VL_S$.

A domain-specific *animation scenario* is a transformation sequence $H_0 \overset{p_{A_0}}{\Longrightarrow} H_1 \overset{p_{A_1}}{\Longrightarrow} H_2 \Longrightarrow .. \Longrightarrow H_n$ over the animation specification A, with $p_{A_i} \in P_A$.

The *animation scenario shown in the animation view* is defined by restricting the spans of the animation scenario to the animation view TG_V. This results in a sequence of spans $(V_0 \leftarrow C_0 \rightarrow V_1), (V_1 \leftarrow C_1 \rightarrow V_2), .., (V_{n-1} \leftarrow C_{n-1} \rightarrow V_n)$, where each span $V_i \leftarrow C_{V_i} \rightarrow V_{i+1}$ is the restriction of the corresponding transformation step span $H_i \leftarrow C_{H_i} \rightarrow H_{i+1}$ to the animation view TG_V of the application domain, as shown to the right.



△

Remark: An alternative to define animation scenarios in the animation view would have been the restriction of the complete animation specification to TG_V using the backward retyping functor $f^{<}_{TG_V} : AnimSpec_{VL_A} \rightarrow (TG_V, P_V)$. Unfortunately, the resulting graph transformation system $V = (TG_V, P_V)$ would allow also transformations that cannot be constructed by restricting corresponding transformations defined by the animation specification. To get only transformations corresponding to $Trans(A)$, the semantics of V would have to be restricted again to those transformations which can be constructed by restricting valid transformations from $Trans(A)$. The same result is achieved directly in our proce-

dure of restricting the transformation spans themselves to the animation view TG_V. □

Animation rules can be additionally enhanced by operations for continuous changes of objects such as motions or changes of size or color. This is realized by annotating the animation rules with animation operations using the GENGED animation rule editor (see Chapter 5). There are four different kinds of animation operations, namely Linear Move for movements of symbols, Visibility to define a duration for a symbol to be invisible, Resize to change the size of a symbol, and Color to change its color. Input attributes of the operations define its duration time, and the symbol's target position, size or color.

Example 4.2.19 (*Adding Animation Operations to the Echo Model Animation Rules*)
In our echo example, an animation operation is needed for all rules which insert a Message edge (rule BtoH1 etc.). The animation operation fixes a time duration for the visibility of the Message arc. Without this operation, an animation run would show the appearance and disappearance of the Message arc according to the computation speed of the machine the animation scenario is executed on. Thus, the state changes of the system would hardly be visible. △

Each animation scenario in the animation view corresponds to a the simulation run defined by applying the corresponding simulation rules of the original simulation specification S to the original model graph G_0 before the $S2AM$ transformation from G_0 to H_0 and the $S2AR$ transformation from S to A.

Example 4.2.20 (*Animation View for a Scenario of the Echo Algorithm*)
The concrete syntax of the animation scenario in the animation view corresponding to the simulation scenario from Fig. 3.18 is shown in Fig. 4.9. Here, it is much easier to see in which order messages are sent by whom and to whom. Considering scenarios in their animation view it is easier to find functional errors and thus to validate the correctness of the Echo algorithm.

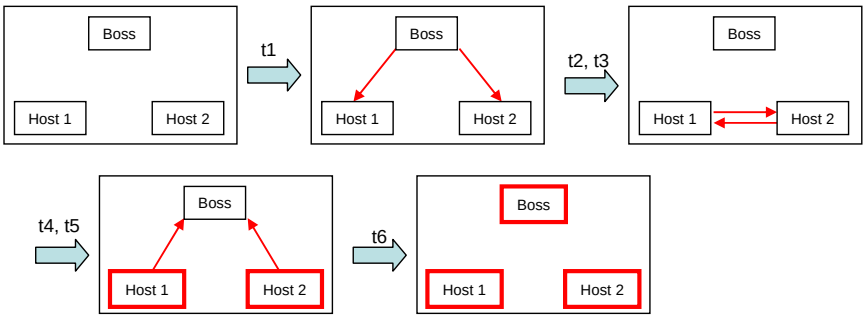


Figure 4.9: A Scenario of the Echo Model in its Animation View

△

Deletion Layer Conditions (k is a deletion layer)	Nondeletion Layer Conditions (k is a nondeletion layer)
1. p is deleting at least one item	1. p is nondeleting, i.e. $K = L$ s.t. $p : L \to R$
2. $0 \leq cl(t) \leq dl(t)$ for all $t \in TG$	2. p has NAC $n : L \to N$, and there is an injective $n' : N \to R$ with $n' \circ n = p$
3. p deletes item of type $t \Longrightarrow dl(t) \leq rl(p)$	3. $x \in L \Longrightarrow cl(t(x)) \leq rl(p)$
4. p creates item of type $t \Longrightarrow cl(t) > rl(p)$	4. p creates item of type $t \Longrightarrow cl(t) > rl(p)$

Table 4.1: Layering Conditions for Termination

4.3 Termination of $S2A$ Transformation

In order to show the termination of $S2A$ transformation, we first apply the termination criteria for model transformation from [EEdL$^+$05] to our notion of $S2AM$ transformation and show that $S2AM$ transformation terminates. Based on this result, we show that given a finite rule set P_S, then also $S2AR$ transformation terminates.

For general model transformation, the termination criteria are defined for layered graph transformation systems. To show termination of a layered graph transformation system, each rule layer has to be either a *nondeletion layer* or a *deletion layer*. Furthermore, each type defined in the type graph of the layered graph transformation system has to be assigned to a creation layer and to a deletion layer, as well. Informally, the *deletion layer conditions* express that the last creation of a node of a certain type should precede the first deletion of a node of the same type. On the other hand, *nondeletion layer conditions* ensure that if an element of a certain type occurs in the left-hand side of a rule then all elements of the same type were already created in previous layers.

Definition 4.3.1 (*Layering Conditions for Termination*)
Let (TG, P) be a layered graph transformation system where each rule $p \in P$ is assigned a rule layer $rl(p) = k$ with $0 \leq k \leq n (k, n \in I\!N)$.

For each type $t \in TG$ we have a creation and a deletion layer $cl(t), dl(t) \in I\!N$, and each rule layer k is either a deletion layer or a nondeletion layer satisfying the conditions in Table 4.3.1, called *deletion layer conditions* and *nondeletion layer conditions*, for all $p \in P$ with $rl(p) = k$.

△

Definition 4.3.2 (*Termination of Layered Graph Transformation Systems*)
A layered graph transformation system $LGTS = (TG, P)$ terminates if for all start graphs

G_0, there is no infinite derivation sequence from G_0 via $P = [P_k = p \in P | rl(p) = k]_{k=0..k_0}$, where starting with layer $k = 0$ rules $p \in P_k$ are applied as long as possible before going over to layer $k + 1 \leq k_0$. △

Theorem 4.3.3 (*Termination of Layered Graph Transformation Systems*)
Each layered graph transformation system with injective matches which fulfills the layering conditions given in Def. 4.3.1 terminates. △

Proof: In [EEdL+05], the proof is given for layered graph *grammars* with a fixed start graph G_0. As the proof does not rely on properties of G_0, it is also valid for layered graph transformation *systems* (without a fixed start graph). □

$S2AM$ transformation given by $S2AM = (VL_S, TG_A, Q)$ is (a special form of) model transformation, where all rules $q \in Q$ are nondeleting. Moreover, we have a NAC $N_q \leftarrow L_q$ with N_q isomorphic to the right-hand side R_q in each $S2A$ transformation rule q, and we have the condition that each rule adds symbols of one type layer only, which is a higher layer than the type layers used in the left-hand side L_q (see Def. 4.2.8).

Hence, to prove termination of $S2AM$ transformation, we need nondeletion layers for the types of the animation view type graph TG_V (Def. 4.2.4). These nondeletion layers according to Def. 4.3.1 are given by the type layers defined in Def. 4.2.4. In the following, we show that $S2A$ transformation rules in Q typed over TG_A satisfy the nondeletion condition.

Proposition 4.3.4 (*$S2A$ Transformation Rules Satisfy the Nondeletion Condition*)
Let $S2AM = (VL_S, TG_A, Q)$ be the definition of an $S2AM$ transformation $S2AM : VL_S \rightarrow VL_A$. Let $LGTS_A = (TG_A, Q)$ be the layered graph transformation system based on $TG_A = TG_S +_{TG_I} TG_V$, the type graph of the animation specification, which is defined together with layers for types, according to Def. 4.2.4. Q is a set of layered $S2A$ transformation rules typed over TG_A, which are defined together with their rule layers according to Def. 4.2.8.

Then, $LGTS_A$ satisfies the nondeletion layer conditions given in Def. 4.3.1, where the creation layers for types correspond to the type layers of TG_A. △

Proof: We have to show that the nondeletion conditions are fulfilled for all rules $q \in Q$.

1. *all rules are nondeleting*:
   this is true for all $S2A$ transformation rules acc. to Def. 4.2.8.

2. *q has NAC $n : L_q \rightarrow N_q$, and there is an injective $n' : N \rightarrow R$ with $n' \circ n = q$*:
   this is true acc. to Def. 4.2.8.

3. $x \in L_q \Longrightarrow cl(t(x)) \leq rl(q)$:
   Let q be a rule with a symbol x in its left-hand side L_q. Rule q creates symbols of type $t(y)$, and the rule layer is computed according to Def. 4.2.8 by $rl(q) = tl(t(y) - 1$.

   We have to consider two cases: either x is typed over TG_S, or x is typed over $TG_V - TG_I$. In the first case, we have $cl(t(x)) = 0$ according to Def. 4.2.4, which is always less or equal to any possible rule layer number. In the second case, x is not typed over TG_S. As x is in L_q, rule q does not create symbols typed over $t(x)$ but symbols typed over $t(y)$ with $tl(t(y)) > tl(t(x))$. By definition we have $rl(q) = tl(t(y)) - 1 = cl(t(y)) - 1$. Hence, we get $rl(q) \geq cl(t(x))$.

4. *q creates item of type t* $\Longrightarrow cl(t) > rl(q)$:
   According to Def. 4.2.8, symbols typed over TG_S are preserved but never created by $S2A$ transformation rules. All symbols and links created by q are typed over TG_A. Moreover, q adds symbols and links of type t only. The rule layer of rule q creating symbols of type t is defined in Def. 4.2.8 depending on the type layer of t by $rl(q) = tl(t) - 1$. As we have $cl(t) = tl(t)$, we get $rl(q) = cl(t) - 1$, which means that the creation layer of a type of a symbol created by the rule is always greater than the rule layer number.

□

Theorem 4.3.5 (*$S2AM$ Transformation Terminates*)
Let $S2AM : VL_S \rightarrow VL_A$ be an $S2AM$ transformation given by $S2AM = (VL_S, TG_A, Q)$, where the layered graph transformation system $LGTS_A = (TG_A, Q)$ defines the $S2A$ transformation rules. Then, for all graphs in VL_S, the rules in Q are applied to, the $S2AM$ transformation defined by $S2AM$ terminates. △

Proof: The theorem follows directly from Proposition 4.3.4 and Theorem 4.3.3. □

Theorem 4.3.6 (*$S2AR$ Transformation Terminates*)
Let $S2AR : P_S \rightarrow P_A$ be an $S2AR$ transformation given by $S2AR = (P_S, TG_A, Q,)$. Then, for all rules $p_S \in P_S$ with P_S being a finite set of rules, the $S2AR$ transformation according to Def. 4.2.14 terminates, if each $p_S \in P_S$ has a finite number of NACs. △

Proof: In Def. 4.2.12, the transformation of a rule $p = (N_i \xleftarrow{n} L \xleftarrow{l} I \xrightarrow{r} R)$ by non-deleting rules of the form $q = (N_q \xleftarrow{n_q} L_q \xrightarrow{q} R_q)$ is defined componentwise by the transformation of the rule graphs I, L, R and N_i according to the respective application case, if the NAC N_q is satisfied for the respective matches. Rule q is applicable to p only once at the same elementary match due to the fact that the NAC N_q is isomorphic to the right-hand side R_q.

Moreover, as shown in Theorem 4.3.5, rule q can be applied to a single rule graph G_p only a finite number of times even if q is applicable to G_p again at different matches. Hence, we conclude that q is applicable to one complete rule p also only a finite number of times, provided that the number of rule graphs of p (i.e. the number of NACs) is finite.

We assume now that q has been applied as long as possible to p, resulting in rule p', and cannot be applied any more. We have to show that q is never again applicable to any rule resulting from the application of any other $S2A$ transformation rules to p'. According to the layering of $S2A$ transformation rules, only those $S2A$ transformation rules may be applied after q which belong to the same rule layer as q or to higher rule layers. We do not have to consider higher rule layers, as q must not be applied after $S2A$ transformation rules of higher rule layers. According to Def. 4.2.8, all $S2A$ transformation rules in the same rule layer as q create only symbols of types which do not occur in L_q. Hence, after applying one or more other $S2A$ transformation rules from the layer of q to p', resulting in p'', q is still not applicable to p'', as no new matches become possible. Moreover, if NAC N_q has not been satisfied before, it is still not satisfied after the application of other rules because all $S2A$ transformation rules are nondeleting and do not change the codomain of the morphism from N_q to the rule graph. Thus, we have shown so far that the $S2AR$ transformation of a single rule p terminates, if rule p does not have an infinite number of NACs.

We conclude that $S2AR$ transformation of a finite set P_S of rules also terminates, as it terminates for each element of P_S. □

4.4 Syntactical Correctness of $S2A$ Transformation

Syntactical correctness of S2A transformation $S2A = (S2AM, S2AR)$ means syntactical correctness of $S2AM$ transformation on the one hand, and syntactical correctness of $S2AR$ transformation on the other hand.

An $S2AM$ transformation $S2AM = (VL_S, TG_A, Q)$ is called syntactically correct, if for all $S2AM$ transformation sequences $G_S \overset{Q\ !}{\Longrightarrow} G_A$, with $G_S \in VL_S$, the resulting graphs G_A are typed over TG_A. According to this simple notion of syntactical correctness, $S2AM$ transformation is syntactically correct, since the $S2A$ rules in Q are typed over TG_A, and each graph $G_S \in VL_S$ is (via retyping) also typed over TG_A. Hence, also the resulting graph $S2AM(G_S)$ is typed over TG_A.

An $S2AR$ transformation $S2AR = (P_S, P_A, Q)$ is called syntactically correct, if for all $S2AR$ transformation sequences $p_S \overset{Q\ !}{\Longrightarrow} p_A$, with $p_S \in P_S$, the resulting rules p_A are typed over TG_A. According to this simple notion of syntactical correctness, $S2AR$ transformation is syntactically correct, since the $S2A$ transformation rules in Q are typed over TG_A, and each rule $p_S \in P_S$ is (via retyping) also typed over TG_A. Hence, also the

resulting rule p_A is typed over TG_A.

Provided we have syntactical correctness of an $S2A$ transformation, then all possibly derivable graphs V_i in the animation view (and hence, all possible animation scenarios in the animation view) are typed over TG_V due to the construction according to Def. 4.2.18: a graph V_i in an animation scenario in the animation view is defined by restricting a graph H_i from an animation scenario to the animation view type graph TG_V, i.e. $V_i = H_i|_{TG_V}$.

4.4.1 Confluence of $S2A$ Transformation

A stronger notion of syntactical correctness of $S2A$ transformation would include well-definedness, i.e. *functional behavior* of both $S2AM$ and $S2AR$ transformation. Functional behavior means that, in addition to the termination of $S2AM$ and $S2AR$ transformation (shown in Section 4.3), *local confluence* is required. Local confluence of $S2AM$ and $S2AR$ transformation is not shown in general in this thesis. Instead, for our running example and additional applications in Section 4.7, we argue that the specific $S2AM$ and $S2AR$ transformations are locally confluent.

Local confluence of graph transformation systems can be shown using the concept of critical pairs [EPT04, HKT02a]. Critical pairs, which can be detected and analyzed statically, represent potential conflicts in a minimal context. If a graph transformation system is terminating, confluence follows if all critical pairs can be joined.

Since $S2A$ transformation rules are non-deleting, the only reason why two direct transformations may be in conflict, is that the application of the first rule generates graph objects the existence of which are prohibited by a NAC of the second rule.

We will argue that our sample $S2A$ transformation of the Echo model is confluent by finding and analyzing the conflicting transformations. Since $S2A$ transformation is defined as a layered graph transformation system, the conflict analysis can be optimized in the way that conflicts are searched for rules from the same layer only.

Example 4.4.1 (*Confluence of $S2A_{Echo}$*)
The $S2A$ transformation for the Echo model is given by $S2A_{Echo} = (S2AM_{Echo}, S2AR_{Echo})$, with $S2AM_{Echo} = (VL_S, TG_A, Q)$ and $S2AR_{Echo} = (P_S, TG_A, Q)$. The simulation alphabet TG_S is depicted in Fig. 3.12, the simulation language VL_S consists of all those TG_S-typed graphs which model the Echo net structure in Fig. 3.11 with different markings. Three sample simulation rules from the set of simulation rules P_S are shown in Fig. 3.13. The animation alphabet TG_A is shown in Fig. 4.3, and the set of $S2A$ transformation rules Q is given in Fig. 4.4.

$S2AM$ transformation of the initial model state The $S2AM$ transformation of the initial Echo model state is considered in Example 4.2.11. This transformation consists of one

$S2AM$ step only, and is trivially confluent.

$S2AR$ **transformation of the simulation rules** The first layer of Q contains one $S2A$ transformation rule, the rule initial. This rule is applicable to all simulation rules in P_S according to Case (1) of Def. 4.2.12, at exactly one match each. (The application of initial to rule t1 has been shown in Figure 4.6.) Thus we get no conflicts for transformations of the first layer of Q. After the first layer has terminated (i.e. the $S2A$ transformation rule initial is not applicable any more to any simulation rule), we now have initially extended all simulation rules which now have the three Host nodes attached to the Net node in each of the rule graphs, with morphisms inbetween, such that the new three Host nodes are preserved by all initially extended simulation rules.

The second layer of Q contains all remaining $S2A$ transformation rules. Considering the marking rules first, we find that there is at most one $S2A$ transformation rule q which can be applied to a marking rule at exactly one match, according to Cases (3) and (4). Hence, for the transformation of the marking rules we do not get any conflicts. Analogously, for the unmarking rules, we also do not have conflicts, as we find again at most one $S2A$ transformation rule which is applicable to an unmarking rule at a deterministic match (this time according to Case (2)). The (initially extended) marking and unmarking rules become thus finally extended by $S2AR$ transformation, with rule q either inserting a Message symbol between two Host symbols (if the place in the match is contained in $\{BtoH1, BtoH2, H1toH2, H2toH1, H1toB\}$), or connecting a Ready symbol to a Host symbol (if the place in the match is contained in $\{H1ready, Bready, H2ready\}$. Afterwards, no more $S2A$ model transformation rules are applicable to the finally extended marking and unmarking rules. Finally, we consider the firing rules in P_S (one for each transition in the original Petri net). We find that more than one $S2A$ transformation rule may be applicable to a firing rule. Consider the initially extended simulation rule t1' after the application of $S2A$ transformation rule initial to simulation rule t1 (see Fig. 4.6). Two $S2A$ transformation rules are applicable to t1', namely rule BtoH1 and rule BtoH2. Both rules are applicable according to Cases (3) and (4) at exactly one match each. (The application of BtoH1 to rule t1' is shown in Fig. 4.7.) Obviously, the resulting direct rule transformations are parallel independent because none of the two $S2A$ transformation rules generates elements which are forbidden by a NAC of the other rule. Hence they can be applied in any order. The same argument can be used for the analysis of the $S2AR$ transformation sequences of the remaining firing rules.

Thus, the $S2AR$ transformation for the Echo Model is confluent. Since it is also terminating, it has functional behavior.

△

4.5 Semantical Correctness of $S2A$ Transformation

The $S2AR$ transformation $p_S \overset{Q\ !}{\Longrightarrow} p_A$ of a rule $p_S = (N_{S_i} \leftarrow L_S \leftarrow I_S \rightarrow R_S)$ to a rule $p_A = (N_{A_i} \leftarrow L_A \leftarrow I_A \rightarrow R_A), i \in \{1, .., n\}$ should be reflected in the correspondence between the graphs derived by p_S and those derived by p_A. More precisely, given the $S2AR$ transformation $p_S \overset{Q\ !}{\Longrightarrow} p_A$, and the $S2AM$ transformation $G_S \overset{Q\ !}{\Longrightarrow} G_A$, then the graph H_S resulting from the simulation step $G_S \overset{p_S}{\Longrightarrow} H_S$ and the graph H_A resulting from the animation step $G_A \overset{p_A}{\Longrightarrow} H_A$ are related by $S2AM$ transformation, i.e. there is an $S2AM$ transformation $H_S \overset{Q\ !}{\Longrightarrow} H_A$. Furthermore, also the rule applicability should be preserved, i.e. a transformed rule p_A should be applicable exactly to the transformed graph G_A iff the original rule p_S is applicable to the original graph G_S.

In more detail, the requirements for semantical correctness of $S2A$ transformation are the following: Let $G_S \overset{Q\ !}{\Longrightarrow} G_A$ be the $S2AM$ transformation of graph G_S into G_A, and $p_S \overset{Q\ !}{\Longrightarrow} p_A$ be the $S2AR$ transformation of rule p_S into p_A according to Def. 4.2.14.

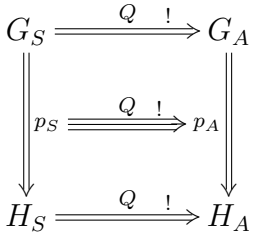
1. *NAC-Compatibility:*
   If there exists a match $L_S \xrightarrow{m_S} G_S$ of rule p_S satisfying NACs N_{S_i}, then there exists a match $L_A \xrightarrow{m_A} G_A$ satisfying the corresponding NACs N_{A_i}.

2. *Local Semantical Correctness:* For each simulation step $G_S \overset{p_S, m_S}{\Longrightarrow} H_S$ there is an animation step $G_A \overset{p_A, m_A}{\Longrightarrow} H_A$ such that there exists an $S2AM$ transformation $H_S \overset{Q\ !}{\Longrightarrow} H_A$.

The two conditions stated above comprise the notion of *semantical correctness* of $S2A$ transformation (Def. 4.5.1). In Theorem 4.5.11, we show that the conditions are satisfied for $S2A$ transformation, provided we have NAC-compatibility of the $S2AM$ transformation. This NAC-compatibility has to be checked individually for concrete $S2AM$ transformations, whereas the NAC-compatibility of $S2AR$ transformations is shown in general in Fact 4.5.4.

Definition 4.5.1 (*Semantical Correctness of $S2A$ Transformation*)
An $S2A$ transformation $S2A : SimSpec_{VL_S} \rightarrow AnimSpec_{VL_A}$ given by $S2A = (S2AM : VL_S \rightarrow VL_A, S2AR : P_S \rightarrow P_A)$ is called *semantically correct* if for each simulation step $G_S \overset{p_S}{\Longrightarrow} H_S$ with $G_S \in VL_S$ and each $S2AR$ transformation sequence $p_S \overset{Q\ !}{\Longrightarrow} p_A$, we have

1. $S2AM$ transformation sequences $G_S \overset{Q\ !}{\Longrightarrow} G_A$ and $H_S \overset{Q\ !}{\Longrightarrow} H_A$, and
2. an animation step $G_A \overset{p_A}{\Longrightarrow} H_A$.



△

Definition 4.5.2 (*NAC-Compatibility of $S2AM$ Transformation*)
An $S2AM$ transformation $S2AM : VL_S \to VL_A$ based on the $S2A$ transformation system (TG_A, Q) is called *NAC-compatible* if the following holds for all $q \in Q$ and $G_i \overset{p_i}{\Longrightarrow} H_i$ derivable from some $G_S \overset{p_S}{\Longrightarrow} H_S$ by $S2A$ transformation: If q is applicable to p_i (i.e. there is a match from q to a rule graph of p_i which satisfies $NAC_q = (L_q \overset{n}{\to} N_q)$), then each match of q in G_i (resp. H_i) satisfies NAC_q. More precisely, we only need for the construction in the proof of Theorem 4.5.7 that in

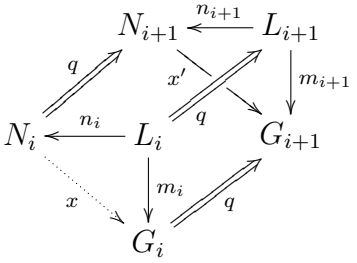
Case (1): $G_i \overset{q}{\Longrightarrow} G_{i+1}$ and $H_i \overset{q}{\Longrightarrow} H_{i+1}$ satisfy NAC_q,
Case (2): $G_i \overset{q}{\Longrightarrow} G_{i+1}$ satisfies NAC_q,
Case (3): $H_i \overset{q}{\Longrightarrow} H_{i+1}$ satisfies NAC_q. △

Definition 4.5.3 (*NAC-Compatibility of $S2AR$ Transformation*)
An $S2AR$ transformation $S2AR : P_S \to P_A$ with $S2AR = (P_S, TG_A, Q)$ is *NAC-compatible* if the following holds for all $q \in Q$ and $G_i \overset{p_i}{\Longrightarrow} H_i$ derivable from some $G_S \overset{p_S}{\Longrightarrow} H_S$ by $S2A$ transformation: If $G_i \overset{p_i}{\Longrightarrow} H_i$ satisfies NAC_{p_i} and $p_i \overset{q}{\Longrightarrow} p_{i+1}$ satisfies NAC_q, then $G_{i+1} \overset{p_{i+1}}{\Longrightarrow} H_{i+1}$ satisfies $NAC_{p_{i+1}}$. △

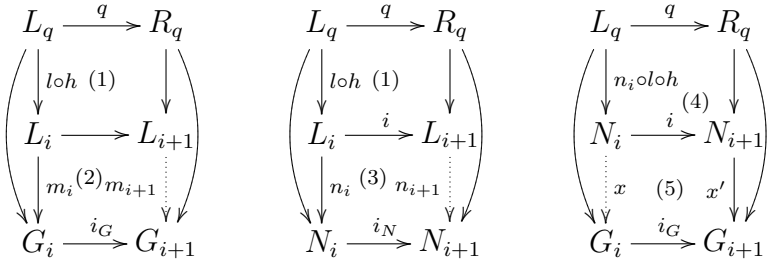
Fact 4.5.4 (*NAC-Compatibility of $S2AR$ Transformations*)
An $S2AR$ transformation $S2AR : P_S \to P_A$ with $S2AR = (P_S, TG_A, Q)$ is always NAC-compatible in the sense of Def. 4.5.3. △

Proof: We know that p_i satisfies NAC_i. This means, there does not exist an injective graph morphism $x : N_i \to G_i$ with $x \circ n_i = m_i$. We must show that then there does not exist an injective graph morphism $x' : N_{i+1} \to G_{i+1}$ with $x' \circ n_{i+1} = m_{i+1}$.

We assume that there exists such an injective graph morphism $x' : N_{i+1} \to G_{i+1}$ with $x' \circ n_{i+1} = m_{i+1}$. Then we have the situation depicted in the diagram to the right. If we can show that now we get an injective graph morphism $x : N_i \to G_i$ with $x \circ n_i = m_i$, we have a contradiction to the precondition.



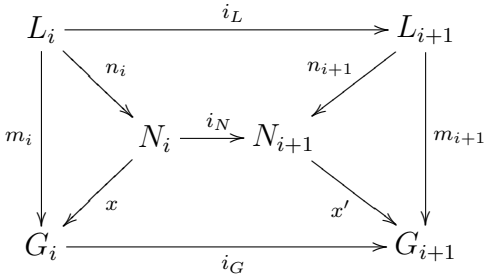
Case (1)**, Case** (2):
Consider the diagrams below.



Square (1) and the surrounding square $(1+2)$ are pushouts (the transformations $L_i \overset{q}{\Longrightarrow} L_{i+1}$ and $G_i \overset{q}{\Longrightarrow} G_{i+1}$). Thus the morphism $L_{i+1} \overset{m_{i+1}}{\longrightarrow} G_{i+1}$ is unique, and square (2) is a pushout as well, due to Property A.11.

Analogously, we get the unique morphism $L_{i+1} \overset{n_{i+1}}{\longrightarrow} N_{i+1}$, and a pushout in square (3). By assumption we know that the morphism $N_{i+1} \overset{x'}{\longrightarrow} G_{i+1}$ in square (5) is a monomorphism. As square (4) is a pushout, and the rule morphism $L_q \overset{q}{\longrightarrow} R_q$ is injective, we can apply Property A.10 and obtain an injective morphism $N_i \overset{x}{\longrightarrow} G_i$ and a pushout in square (5).

Now we have the situation depicted in the diagram to the right, where we have:



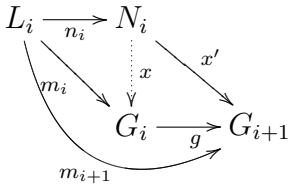
$$
\begin{aligned}
i_G \circ m_i &= m_{i+1} \circ i_L \quad \text{due to PO (2)} \\
&= x' \circ n_{i+1} \circ i_L \quad \text{due to assumption} \\
&= x' \circ i_N \circ n_i \quad \text{due to PO (3)} \\
&= i_G \circ x \circ n_i \quad \text{due to PO (5)}
\end{aligned}
$$

Since i_G is injective (due to pushout $(1+2)$), we get $m_i = x \circ n_i$. This is a contradiction to the precondition that there is no injective x with $x \circ n_i = m_i$. We conclude that the assumption is wrong, i.e. there does not exist a morphism $N_{i+1} \overset{x'}{\longrightarrow} G_{i+1}$ with $x' \circ n_{i+1} = m_{i+1}$.

Case (3):

For this case we have $L_{i+1} = L_i, N_{i+1} = N_i$ and $n_{i+1} = n_i$. If there is a match $L_q \to G_i$, we have the transformation $G_i \overset{q}{\Longrightarrow} G_{i+1}$, inducing the inclusion morphism $G_i \overset{i_G}{\longrightarrow} G_{i+1}$. Again, we assume we had a morphism $N_{i+1} \overset{x'}{\longrightarrow} G_{i+1}$ with $x' \circ n_i = i_G \circ m_i$.

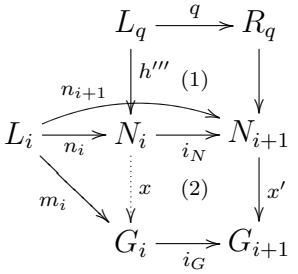
We define the match $L_i \overset{m_{i+1}}{\longrightarrow} G_{i+1}$ by $m_{i+1} = i_G \circ m_i$. We now have the situation shown in the diagram to the right. We construct the morphism $N_i \overset{x}{\longrightarrow} G_i$ by restriction of $N_{i+1} \overset{x'}{\longrightarrow} G_{i+1}$, i.e. by applying the backwards retyping functor $f^{<}_{TG_i}$ based on the type graph inclusion $TG_i \hookrightarrow TG_{i+1}$. This functor maps G_{i+1} to G_i due to the fact $S2A$ rules are type-increasing. N_i is mapped to N_i, and the injective morphism x' is mapped to an injective morphism $N_i \overset{x}{\longrightarrow} G_i$ due to Fact 2.1.26.



Due to the functor properties of the backward retyping functor $f^{<}_{TG_i}$, commuting diagrams are preserved by $f^{<}_{TG_i}$. Hence, we have the commuting diagram $x \circ n_i = m_i$ with x injective. This is a contradiction to the precondition that there is no injective x with $x \circ n_i = m_i$. We conclude that the assumption is wrong, i.e. there does not exist a morphism $N_{i+1} \xrightarrow{x'} G_{i+1}$ with $x' \circ n_{i+1} = m_{i+1}$.

Case (4):
In this case, the only rule graph transformed by q is the NAC N_i.

We define the match $L_i \xrightarrow{m_{i+1}} G_{i+1}$ by $m_{i+1} = i_G \circ m_i$. We have the situation shown in the diagram to the right, with a pushout in square (1). Moreover, $n_{i+1} = i_N \circ n_i$. Again, we assume we had an injective morphism $N_{i+1} \xrightarrow{x'} G_{i+1}$ with $x' \circ n_{i+1} = m_{i+1}$. We construct the morphism $N_i \xrightarrow{x} G_i$ as for Case (3) by restriction of $N_{i+1} \xrightarrow{x'} G_{i+1}$, i.e. by applying the backwards retyping functor $f^{<}_{TG_i}$ based on the type graph inclusion $TG_i \hookrightarrow TG_{i+1}$.



This functor maps G_{i+1} to G_i due to the fact that $S2A$ rules are type-increasing. N_{i+1} is mapped to N_i, and the injective morphism x' is mapped to an injective morphism $N_i \xrightarrow{x} G_i$ due to Fact 2.1.26. Due to the functor properties of the backward retyping functor $f^{<}_{TG_i}$, commuting diagrams are preserved by $f^{<}_{TG_i}$. Hence, we have the commuting diagram $x \circ n_i = m_i$ with x injective. This is a contradiction to the precondition that there is no injective x with $x \circ n_i = m_i$. We conclude that the assumption is wrong, i.e. there does not exist a morphism $N_{i+1} \xrightarrow{x'} G_{i+1}$ with $x' \circ n_{i+1} = m_{i+1}$.

□

Definition 4.5.5 (*NAC-Compatibility of $S2A$ Transformation*)
An $S2A$ transformation $S2A = (S2AM : VL_S \to VL_A, S2AR : P_S \to P_A)$ based on the $S2A$ transformation system (TG_A, Q) is called *NAC-compatible* if the following conditions hold for all $q \in Q$ and $G_i \overset{p_i}{\Longrightarrow} H_i$ derivable from some $G_S \overset{p_S}{\Longrightarrow} H_S$ by $S2A$ transformation:

1. NAC-compatibility of $S2AM$ (Def. 4.5.2), and
2. NAC-compatibility of $S2AR$ (Def. 4.5.3).

△

Remark: According to Fact 4.5.4, each $S2AR$ transformation $S2AR : P_S \to P_A$ with $S2AR = (P_S, TG_A, Q)$ is NAC-compatible provided that (TG_A, Q) is a type-increasing graph transformation system. Thus, for specific applications, it suffices to show only NAC-compatibility of $S2AM$, where the general criteria are still missing. □

Fact 4.5.6 (*Example: NAC-Compatibility of the Echo $S2AM$ Transformation*)
The Echo model $S2AM$ transformation is NAC-compatible (see Def. 4.5.2) in the following sense: For all $p_i \overset{q}{\Longrightarrow} p_{i+1}$ with $q = (L_q \overset{q}{\longrightarrow} R_q)$ and $NAC_q = (L_q \overset{q}{\longrightarrow} R_q)$ such that the match from q to p_i satisfies NAC_q, the following $S2AM$ steps also satisfy NAC_q according to the rule transformation cases below:

Case (1): $G_i \overset{q}{\Longrightarrow} G_{i+1}$ and $H_i \overset{q}{\Longrightarrow} H_{i+1}$,
Case (2): $G_i \overset{q}{\Longrightarrow} G_{i+1}$,
Case (3): $H_i \overset{q}{\Longrightarrow} H_{i+1}$. △

Proof: We show for all $q \in Q$ that for a match $L_q \to X$ there is no NAC-morphism $(R_q - L_q) \overset{x}{\longrightarrow} X$. Due the property of all $q \in Q$ being type-increasing, in this case NAC_q is satisfied for this match.

The only $S2A$ rule which can be applied to any rule p_i according to Case (1) is rule initial. As rule initial belongs to rule layer 1, all rules p_i it can be applied to, are the original simulation rules, and do not contain symbols typed over TG_V. Hence, a step involving the application of initial to a rule p_i is always NAC-compatible, since

- the match $L_q \overset{h}{\longrightarrow} I_{p_i} \overset{l_{p_i}}{\longrightarrow} L_{p_i} \overset{m_{p_i}}{\longrightarrow} G_i$ satisfies NAC_q as G_i does not contain TG_V-typed elements, and hence there is no NAC-morphism $(R_q - L_q) \overset{x}{\longrightarrow} G_i$;

- the match $L_q \overset{h}{\longrightarrow} I_{p_i} \overset{r_{p_i}}{\longrightarrow} R_{p_i} \overset{m^*_{p_i}}{\longrightarrow} H_i$ satisfies NAC_q as H_i does not contain TG_V-typed elements, and hence there is no NAC-morphism $(R_q - L_q) \overset{x}{\longrightarrow} H_i$;

All subsequent $S2A$ transformation steps are either according to Case (2), or Case (3) (we need not consider Case (4) for NAC-compatibility).

Let us consider a step according to Case (2), first: We assume that q is applicable to p_i, i.e. there is a match $L_q \overset{h}{\longrightarrow} L_i$ satisfying NAC_q. Now, if NAC_q is not satisfied for the match $L_q \overset{h}{\longrightarrow} L_i \overset{m}{\longrightarrow} G_i$, then this means that a rule q' must have been applied before to a predecessor rule p_j according to Case (2) with $p_j \overset{q'}{\Longrightarrow} p_{j+1} \Longrightarrow \cdot\cdot \Longrightarrow p_i$ with $j < i$, such that the objects in $(R_{q'} - L_{q'})$ added by q' are also added by q, i.e. $(R_{q'} - L_{q'}) = (R_q - L_q)$, and the matches of q' and q overlap in those objects the newly generated objects are linked to. But then, the objects in $(R_q - L_q)$ would also have been added to L_j and in this case we have a NAC-morphism $(R_q - L_q) \to L_{j+1} \to L_i$ which is a contradiction to our assumption that NAC_q is satisfied for the match $L_q \to L_i$. Hence, NAC_q must be satisfied for the match $L_q \overset{h}{\longrightarrow} L_i \overset{m}{\longrightarrow} G_i$.

Analogously, we can argue for the Case (3) steps. □

The most important requirement for the semantical correctness of the $S2A$ transformation is that the application of an $S2A$ transformation rule to a simulation rule preserves a

simulation step, i.e. each direct $S2A$ transformation step is *locally semantically confluent*. We call this requirement *local semantical correctness* of $S2A$ transformation and show that it is fulfilled in Theorem 4.5.7.

Please note that for showing local semantical correctness we have to require NAC-compatibility of the $S2AM$ part of the $S2A$ transformation, as NAC-compatibility of $S2AM$ does not hold in general for all $S2A$ transformation systems.

Figure 4.10 shows a counter example for Case (1) where NAC $N_q \xleftarrow{n_q} L_q$ is satisfied for all required matches from L_q to rule graphs Ii, L_i and R_i of p_i and to G_i, but not for the match from L_q to H_i.

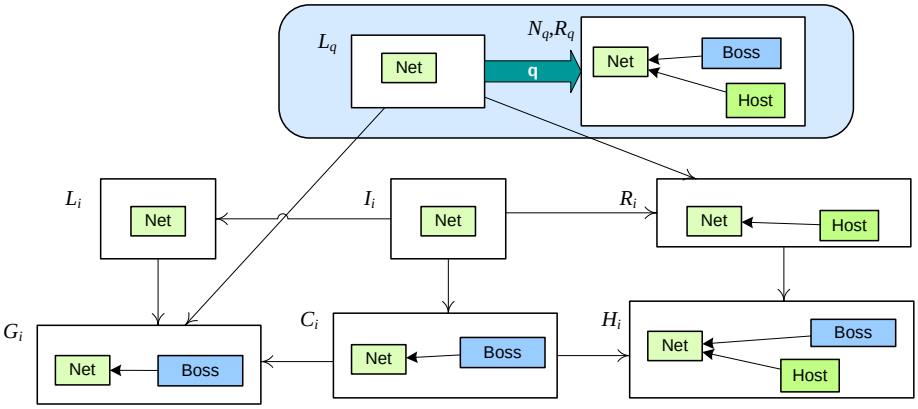


Figure 4.10: Counter Example: No NAC_q Compatibility

We show that in our running example (see Example 4.5.8) and in all applications considered (see Section 4.7), we have NAC-compatibility of $S2AM$ in all possible $S2A$ transformation steps.

Theorem 4.5.7 (*Local Semantical Correctness of $S2A$ Transformation*)
Given a NAC-compatible $S2A$ transformation $S2A : \mathbf{SimSpec_{VL_S}} \rightarrow \mathbf{AnimSpec_{VL_A}}$ with $S2A = (S2AM : VL_S \rightarrow VL_A, S2AR : P_S \rightarrow P_A)$ and an $S2AR$ transformation sequence $p_S \overset{Q}{\Longrightarrow}{}^{!} p_A$ with intermediate $S2AR$ step $p_i \overset{q}{\Longrightarrow} p_{i+1}$ with $q \in Q$.

Then for each graph transformation step $G_i \overset{q}{\Longrightarrow} H_i$ with $G_i, H_i \in \mathbf{Graphs_{TG_A}}$ we have

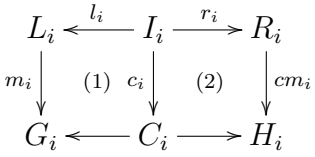
1. Graph transformation steps $G_i \overset{q}{\Longrightarrow} G_{i+1}$ in Cases (1) and (2), $G_i \overset{id}{\Longrightarrow} G_{i+1}$ in Case (3), $H_i \overset{q}{\Longrightarrow} H_{i+1}$ in Cases (1) and (3), and $H_i \overset{id}{\Longrightarrow} H_{i+1}$ in Case (2) of Def. 4.2.12;

2. Graph transformation step $G_{i+1} \overset{p_{i+1}}{\Longrightarrow} H_{i+1}$ with $G_{i+1}, H_{i+1} \in \mathbf{Graphs_{TG_A}}$.

$$\begin{array}{ccc} G_i & \overset{q/id}{\Longrightarrow} & G_{i+1} \\ \Downarrow & & \Downarrow \\ p_i & \overset{q}{\Longrightarrow} & p_{i+1} \\ \Downarrow & & \Downarrow \\ H_i & \overset{q/id}{\Longrightarrow} & H_{i+1} \end{array}$$

△

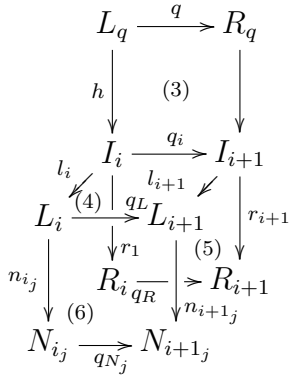
Proof:
Given $G_i \overset{p_i}{\Longrightarrow} H_i$ by the DPO

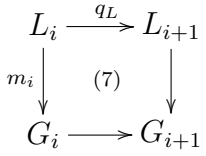


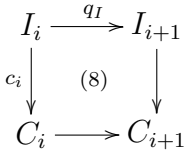
Then we consider for $p_i \overset{q}{\Longrightarrow} p_{i+1}$ according to Def. 4.2.12 the following four Cases:

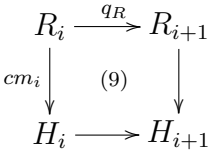
Case (1):
There is a match $L_q \xrightarrow{h} I_i$, and $N_{i+1_j}, I_{i+1}, L_{i+1}, R_{i+1}, n_{i+1_j}, l_{i+1}, r_{i+1}$ are defined by the pushouts $(3)-(6)$ in the left diagram below, leading to rule $p_{i+1} = (N_{i_j} \xleftarrow{n_{i+1_j}} L_{i+1} \xleftarrow{l_{i+1}} I_{i+1} \xrightarrow{r_{i+1}} R_{i+1})$.

Now we construct G_{i+1}, C_{i+1} and H_{i+1} as pushout objects in $(7), (8)$ and (9), respectively. Composing pushouts $(3), (4)$ and (7), we obtain $G_i \overset{q}{\Longrightarrow} G_{i+1}$, and analogously $H_i \overset{q}{\Longrightarrow} H_{i+1}$ by pushouts $(3), (5)$ and (9).

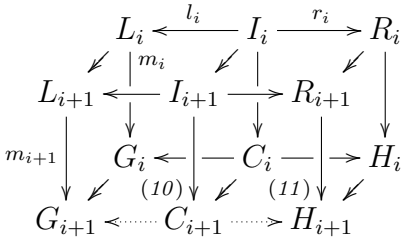








It remains to show $G_{i+1} \overset{p_{i+1}}{\Longrightarrow} H_{i+1}$ by showing that the front squares in double cube shown to the right are pushouts. The back squares are pushouts (1) and (2), the top squares are pushouts (4) and (5), the diagonal squares are pushouts $(7), (8)$ and (9). This leads to unique morphisms $C_{i+1} \to G_{i+1}$ and $C_{i+1} \to H_{i+1}$, such that all squares commute.



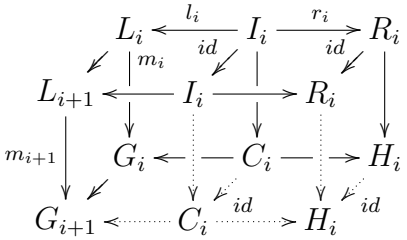
We get pushout (10) in the left bottom and and pushout (11) in the right bottom square by pushout decomposition property A.7, 2, as we have pushouts in the diagonal square (8), and in the composition of the left top and the left diagonal square $(4 + 7)$, as well as in the composition of the right top and the right diagonal square $(5 + 9)$. Using pushout decomposition once more, we can conclude that both front squares are pushouts, as we have pushouts in the respective top squares (4) and (5), as well as in the compositions of the respective back and bottom squares $(1 + 10)$ and $(2 + 11)$. Hence, we have got the double

pushout (the two front squares) corresponding to the transformation step $G_{i+1} \overset{p_{i+1}}{\Longrightarrow} H_{i+1}$.

Case (2):
There is a match $L_q \xrightarrow{h'} L_i$, and L_{i+1}, N_{i+1_j} and n_{i+1_j} are defined by pushouts (12) and (13), and $I_{i+1} = I_i, R_{i+1} = R_i, r_{i+1} = r_i$ and $l_{i+1} = q' \circ l_i$. Now we define G_{i+1} as pushout object in (14), and let $C_{i+1} = C_i, H_{i+1} = H_i$. Thus, composing (12) and (14), we obtain $G_i \overset{q}{\Longrightarrow} G_{i+1}$, and we have $H_i \overset{id}{\Longrightarrow} H_{i+1}$.

$$\begin{array}{ccc} L_q & \xrightarrow{q} & R_q \\ {\scriptstyle h'}\downarrow & (12) & \downarrow \\ L_i & \xrightarrow{q'} & L_{i+1} \\ {\scriptstyle n_{i_j}}\downarrow & (13) & \downarrow{\scriptstyle n_{i+1_j}} \\ N_{i_j} & \longrightarrow & N_{i+1_j} \end{array} \qquad \begin{array}{ccc} L_i & \xrightarrow{q'} & L_{i+1} \\ {\scriptstyle m_i}\downarrow & (14) & \downarrow \\ G_i & \xrightarrow[q']{} & G_{i+1} \end{array}$$

Similar to Case (1) we now get pushouts in the front of the following double cube, leading to $G_{i+1} \overset{p_{i+1}}{\Longrightarrow} H_{i+1}$. Note, however, that in contrast to Case (1), the top and bottom squares of the left cube are no pushouts, but all the diagonal squares are.

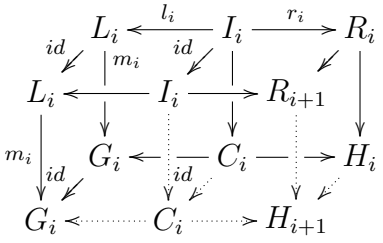


Case (3):
There is a match $L_q \xrightarrow{h''} R_i$, and R_{i+1} is defined by pushout (15), and $I_{i+1} = I_i, L_{i+1} = L_i, N_{i+1_j} = N_{i_j}, l_{i+1} = l_i, n_{i+1_j} = n_{i_j}$ and $r_{i+1} = q' \circ r_i$.

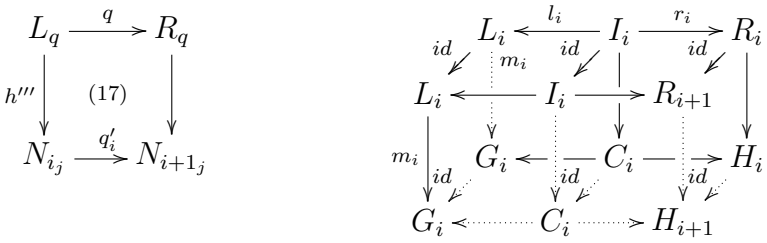
Now we define $C_{i+1} = C_i$, and $G_{i+1} = G_i$, leading to $G_i \overset{id}{\Longrightarrow} G_{i+1}$ and to H_{i+1} by pushout (16).

$$\begin{array}{ccc} L_q & \xrightarrow{q} & R_q \\ {\scriptstyle h''}\downarrow & (15) & \downarrow \\ R_i & \xrightarrow{q'} & R_{i+1} \end{array} \qquad \begin{array}{ccc} R_i & \xrightarrow{q'} & R_{i+1} \\ {\scriptstyle cm_i}\downarrow & (16) & \downarrow \\ H_i & \longrightarrow & H_{i+1} \end{array}$$

Similar to Case (1), we obtain pushouts in the front of the following double cube, leading to $G_{i+1} \overset{p_{i+1}}{\Longrightarrow} H_{i+1}$. Note, however, that in contrast to Case (1), the top and bottom squares of the right cube are no pushouts, but all the diagonal squares are.



Case (4):
There is a match $L_q \xrightarrow{h''} N_{i_j}$, and N_{i+1_j} is defined by pushout (17), and $I_{i+1} = I_i, L_{i+1} = L_i, R_{i+1} = R_i, l_{i+1} = l_i, r_{i+1} = r_i$ and $n_{i+1_j} = q' \circ n_{i_j}$. Now we define $C_{i+1} = C_i$, leading to $G_i \overset{id}{\Longrightarrow} G_{i+1}$ and to $H_i \overset{id}{\Longrightarrow} H_{i+1}$. We obtain a double cube which consists of identities in all diagonal morphisms, leading to $G_{i+1} \overset{p_{i+1}}{\Longrightarrow} H_{i+1}$ which is equivalent to $G_i \overset{p_i}{\Longrightarrow} H_i$, as the corresponding double pushouts are equal.



□

Example 4.5.8 (*Local Semantical Correctness of* $S2A_{Echo}$)
Let us consider layer 1 of the Echo simulation rules (consisting of marking and unmarking rules). According to Example 4.4.1, to each of the layer-1-rules, we can apply at first the $S2A$ rule initial (and no other $S2A$ rule) according to Case (1) of Def. 4.2.12. If we assume as initial model the unmarked Echo C/E net from Fig. 3.11, and a simulation step which marks the place B_idle (by applying simulation rule markB_idle from Fig. 3.13, we get the local confluence diagram in Fig. 4.11 (where only parts of the net are drawn due to space limits):

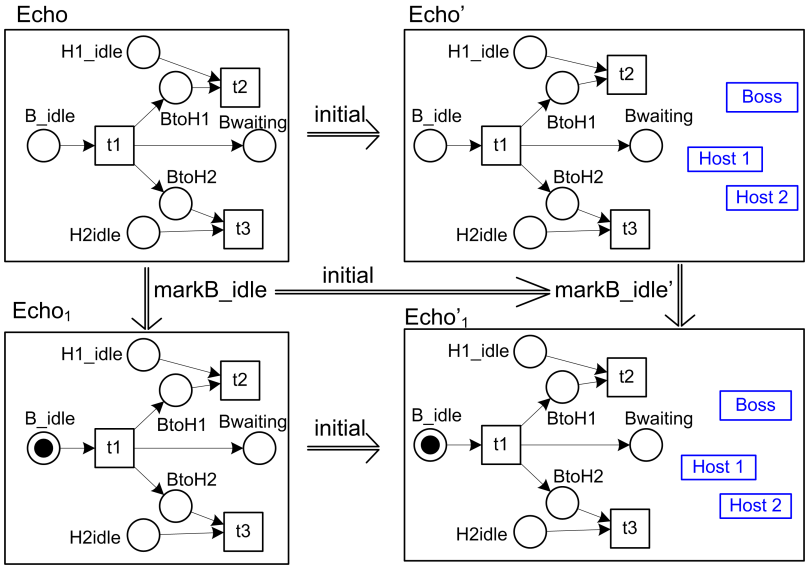


Figure 4.11: Local Semantical Confluence Diagram for Transforming markB_idle

The result of applying $S2A$ rule initial to simulation rule markB_idle is an extended rule markB_idle', which contains additinally the three Host nodes, linked to the Net node in all three rule graphs, with morphisms between the nodes with equal names, such that the new three Host nodes are preserved by rule markB_idle'. Please note that the NAC of $S2A$ rule initial is satisfied both for the match into graph $Echo$ and for the match into the extended graph $Echo_1$.

We get similar diagrams for the pair of the $S2A$ rule initial and all other marking (and unmarking) rules.

In graph $Echo_1$, we have the necessary marking to apply the firing rule t1. To firing rule t1, the $S2A$ rule initial has to be applied first, yielding the first (left) confluent step in Fig. 4.12, where the resulting rule t1' extends t1, and the resulting graph $Echo'_2$ extends $Echo_2$ by the three Host nodes.

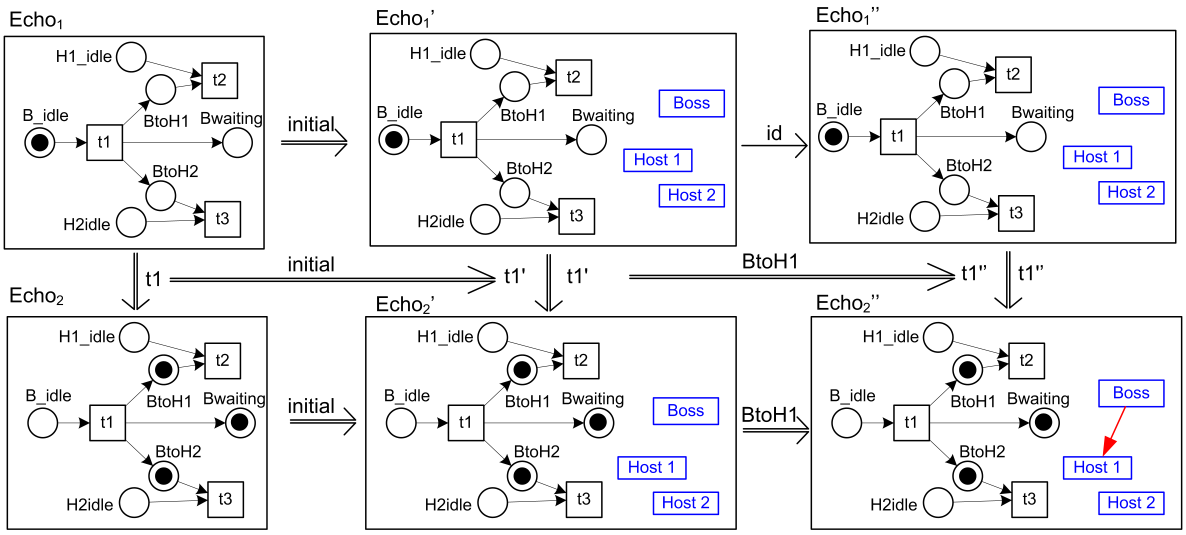


Figure 4.12: Local Semantical Confluence Diagrams for transforming t1

Now we can apply two more $S2A$ rules to t1', namely BtoH1 and BtoH2. As shown in Example 4.4.1, both rules are parallel independent and may be applied in any order. In Fig. 4.12, the second (right) confluent step corresponds to the application of $S2A$ rule BtoH1. △

In addition to the previously shown two requirements for semantical correctness, NAC-compatibility (Def. 4.5.5, Fact 4.5.4) and local semantical correctness (Theorem 4.5.7), we require for the semantical correctness of all possible $S2A$ transformation sequences that the $S2A$ transformation rules $q \in Q$ are parallel and sequential independent from the animation rules $p_A \in A$. If this is the case, we can apply $S2A$ transformation rules q to graphs G_n even after the $S2A$ transformation $p_S \overset{Q\ !}{\Longrightarrow} p_n$ is finished, i.e. $p_n = p_A$.

Definition 4.5.9 (*Rule Compatibility of $S2A$ Transformation*)
An $S2A$ transformation $S2A = (S2AM : VL_S \to VL_A, S2AR : P_S \to P_A)$ with $S2AM = (VL_S, TG_A, Q)$ is called *rule-compatible* if for all $p_A \in P_A$ and $q \in Q$ we have that p_A and q are parallel and sequential independent. More precisely, for each $G \overset{p_A}{\Longrightarrow} H$ with $G_S \overset{Q\ *}{\Longrightarrow} G$ and $H_S \overset{Q\ *}{\Longrightarrow} H$ for some $G_S, H_S \in VL_S$ and each $G \overset{q}{\Longrightarrow} G'$ (resp. $H \overset{q}{\Longrightarrow} H'$), we have parallel (resp. sequential) independence of $G \overset{p_A}{\Longrightarrow} H$ and $G \overset{q}{\Longrightarrow} G'$ (resp. $H \overset{q}{\Longrightarrow} H'$). △

Fact 4.5.10 (*Example: Rule Compatibility of the Echo $S2A$ Transformation*)
The Echo Model $S2A$ transformation is rule compatible in the sense of Def. 4.5.9, i.e. all p_A and all q are parallel and sequential independent. △

Proof: If p_A is applicable to a graph G, then there is a match $L_A \xrightarrow{m} G$. Therefore, symbols of at least those types from TG_V that are contained in L_A have also to be contained in G. So, in the sequence $G_S \overset{Q*}{\Longrightarrow} G$ there have been applied at least those rules $q \in Q$ which have also been applied in $p_S \overset{Q\ !}{\Longrightarrow} p_A$ according to Case (1) or (2) (i.e. applied to some L_i, $i = 0, .., n$). All those rules q have a match $L_q \to L_A \to G$ but are not applicable any more to L_A because of their NACs NAC_q. Neither are they applicable to G, due to NAC-compatibility.

So we have to consider only those overlappings L_A/L_q, where $h(L_q)$ is not completely included in $m(L_A)$. There exists exactly one instance of type Net in graph G, in all L_q and in all L_A. Hence, all pairs L_A/L_q overlap in the Net symbol. However, this is uncritical, as the Net symbol is always preserved by all rules $q \in Q$ and $p_A \in P_A$. Moreover, there exist exactly three instances of type Host in graph G and in all L_A. Again, overlapping in Host nodes is uncritical, as they are always preserved by all rules. Furthermore, there exists at most one Place node with a specific name. So, all pairs L_A and L_q which both contain a Place node with the same name, overlap in this node. Again, this is uncritical as Place symbols are always preserved by all rules. The only critical symbols L_A and L_q could overlap at, are the Token symbols (and the corresponding EdgeTk edges connecting the tokens to the places). There is exactly one Token symbol in the LHS of all $S2A$ rules. This Token symbol is the last node apart from the Net node, the Host nodes and the Place node. As we have argued before, L_A and L_q overlap already in the Net node, the Host node and in the Place node. As there is at most one Token node connected to a Place, they overlap also in the Token node. Thus, L_A and L_q always overlap completely, if they overlap at all. For this case it was shown above that neither p_A nor q are applicable due to their NACs. Hence, all pairs p_A/q are parallel independent. Due to the Local Church Rosser Theorem 2.1.19, we know that if $G \overset{p_A}{\Longrightarrow} H$ and $G \overset{q}{\Longrightarrow} G'$ are parallel independent (which was shown above), then $G \overset{p_A}{\Longrightarrow} H$ and $H \overset{q}{\Longrightarrow} H'$ are sequential independent. □

Theorem 4.5.11 (*Semantical Correctness of $S2A$ Transformation*)
Each rule compatible $S2A$ transformation $S2A = (S2AM, S2AR)$ is semantically correct, provided that $S2AM$ is terminating and NAC-compatible. △

Proof: Given $S2A = (S2AM : VL_S \to VL_A, S2AR : P_S \to P_A)$ with terminating $S2AM = (VL_S, TG_A, Q)$, a simulation step $G_S \overset{p_S}{\Longrightarrow} H_S$ with $G_S \in VL_S$, and an $S2AR$ transformation sequence $p_S \overset{Q\ !}{\Longrightarrow} p_A$ with $p_S = p_0 \overset{q_0}{\Longrightarrow} p_1 \overset{q_1}{\Longrightarrow} .. \overset{q_{n-1}}{\Longrightarrow} p_n = p_A$ with $n \geq 1$, then we can apply the Local Correctness Theorem 4.5.7 for $i = 0, .., n-1$, leading to the following diagram

$$\begin{array}{ccccccccccc}
G_S = G_0 & \overset{q_0}{\Longrightarrow} & G_1 & \overset{q_1}{\Longrightarrow} & G_2 & \overset{q_2}{\Longrightarrow} & \cdots & \Longrightarrow & G_{n-1} & \overset{q_{n-1}}{\Longrightarrow} & G_n \\
\Downarrow & & \Downarrow & & \Downarrow & & & & \Downarrow & & \Downarrow \\
p_S{=}p_0 & \Longrightarrow & p_1 & & p_2 & & & & p_{n-1} & & p_n{=}p_A \\
H_S = H_0 & \underset{q_0}{\Longrightarrow} & H_1 & \underset{q_1}{\Longrightarrow} & H_2 & \underset{q_2}{\Longrightarrow} & \cdots & \Longrightarrow & H_{n-1} & \underset{q_{n-1}}{\Longrightarrow} & H_n
\end{array}$$

which includes the case $n = 0$ with $G_S = G_0, H_S = H_0$ and $p_S = p_0 = p_A$, where no $q \in Q$ can be applied to $p_S = p_0 = p_A$. If no $q \in Q$ can be applied to G_n and H_n anymore, we are ready, because the top sequence is $G_S \overset{Q\ !}{\Longrightarrow} G_n = G_A$, and the bottom sequence is $H_S \overset{Q\ !}{\Longrightarrow} H_n = H_A$.

Now assume that we have $q_n \in Q$ which is applicable to G_n leading to $G_n \overset{q_n}{\Longrightarrow} G_{n+1}$. Then, rule compatibility implies parallel independence with $G_A \overset{p_A}{\Longrightarrow} H_A$, and the Local Church Rosser Theorem leads to square (n):

$$\begin{array}{ccccccccc}
G_n & \overset{q_n}{\Longrightarrow} & G_{n+1} & \Longrightarrow & \cdots & \Longrightarrow & G_{m-1} & \underset{q_{m-1}}{\Longrightarrow} & G_m = G_A \\
\Downarrow & & \Downarrow & & & & \Downarrow & & \Downarrow \\
H_n & \overset{q_n}{\Longrightarrow} & H_{n+1} & \Longrightarrow & \cdots & \Longrightarrow & H_{m-1} & \overset{q_{m-1}}{\Longrightarrow} & H_m = H_A
\end{array}$$

This procedure can be repeated as long as rules $q_i \in Q$ are applicable to G_i for $i \geq n$. Since $S2AM$ transformation is terminating, we have some $m > n$ such that no $q \in Q$ is applicable to G_m anymore, leading to a sequence $G_S = G_0 \overset{Q\ !}{\Longrightarrow} G_m = G_A$.

Now assume that there is some $q \in Q$ which is still applicable to H_m leading to $H_m \overset{q}{\Longrightarrow} H_{m+1}$. Now rule compatibility implies sequential independence of $G_m \overset{p_A}{\Longrightarrow} H_m \overset{q}{\Longrightarrow} H_{m+1}$. In this case, the Local Church Rosser Theorem would lead to a sequence $G_m \overset{q}{\Longrightarrow} G_{m+1} \overset{p_A}{\Longrightarrow} H_{m+1}$ which contradicts the fact that no $q \in Q$ is applicable to G_m anymore. This implies that also $H_0 \overset{Q\ *}{\Longrightarrow} H_n \overset{Q\ *}{\Longrightarrow} H_m$ is terminating, leading to the required sequence $H_S = H_0 \overset{Q\ !}{\Longrightarrow} H_m = H_A$. □

Fact 4.5.12 (*Example: Semantical Correctness of the Echo $S2A$ Transformation*)
The $S2A$ transformation $S2A = (S2AM, S2AR)$ based on the $S2A$ transformation for the Echo model, where the $S2A$ transformation rules Q are shown in Fig. 4.4, and the simulation rules P_S are given in Fig. 3.13 is semantically correct. △

Proof: Termination has been shown to be fulfilled for general type-increasing $S2A$ transformation systems in Theorem 4.3.6. Since the Echo $S2A$ transformation is NAC-compatible due to Fact 4.5.6, and rule-compatible due to Prop. 4.5.10, it is hence also semantically correct due to Theorem 4.5.11. □

4.6 Semantical Equivalence of Simulation and Animation Specifications

To show semantical equivalence of $\mathbf{SimSpec_{VL_S}}$ and $\mathbf{AnimSpec_{VL_A}}$, we need completenes of the $S2A$ transformation in the sense that we also have semantical correctness in the backward direction. For each animation step $G_A \stackrel{p_A}{\Longrightarrow} H_A$, a so-called $A2S$ backward transformation with $A2S = (A2SM, A2SP)$ should map graphs G_A to G_S and H_A to H_S by $A2SM$ transformation, and the animation rule $p_A = (N_{A_i} \leftarrow L_A \leftarrow I_A \rightarrow R_A)$ to the simulation rule $p_S = (N_{S_i} \leftarrow L_S \leftarrow I_S \rightarrow R_S), i \in \{1, .., n\}$ by $A2SR$ transformation, such that there is a valid simulation step $G_S \stackrel{p_S}{\Longrightarrow} G_A$.

For an $S2A$ transformation, we define an $A2S$ backward transformation by restriction of graphs and rules to TG_S, and show that $A2S$ has the desired property that $A2S \circ S2A \subseteq Id_{VL_S}$. If $S2AM$ is total, we even get $A2S \circ S2A = Id_{VL_S}$.

4.6.1 $A2S$ Backward Transformation

In this section we consider the relation between an animation specification $\mathbf{AnimSpec_{VL_A}}$ and the corresponding simulation specification $\mathbf{SimuSpec_{VL_S}}$ related by $S2A$ transformation. We show in Theorem 4.6.5 that for each $S2A$ transformation there is a backward transformation $A2S : \mathbf{AnimSpec_{VL_A}} \rightarrow \mathbf{SimSpec_{VL_S}}$, i.e. we get $A2S \circ S2A \subseteq Id_{VL_S}$

Definition 4.6.1 (*Characterization of Backward Transformations*)

1. Given an $S2AM$ transformation $S2AM : VL_S \rightarrow VL_A$, then a transformation $A2SM : VL_A \rightarrow VL_S$ is called *backward transformation of $S2AM$* if we have

$$A2SM \circ S2AM \subseteq Id_{VL_S},$$

i.e. $\forall G_S, G'_S \in VL_S, G_A \in VL_A : [(G_S, G_A) \in S2AM, (G_A, G'_S) \in S2AM \Longrightarrow G_S = G'_S]$

2. Given an $S2AR$ transformation $S2AR : P_S \to P_A$, then the transformtion $A2SR : P_A \to P_S$ is called *backward transformation of* $S2AR$ if we have

$$A2SR \circ S2AR \subseteq Id_{P_S}.$$

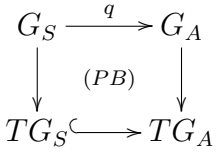
3. Given backward transformations $A2SM$ of $S2AM$ and $A2SR$ of $S2AR$, then $A2S$ = $(A2SM, A2SR)$ is called *backward transformation of* $S2A = (S2AM, S2AR)$.

△

Remark: All transformations in Def. 4.6.1 are considered as relations, and $\circ$ is the relational composition. If $S2AM$ is total, we also require $A2SM$ to be total and $A2SM \circ S2AM = Id_{VL_S}$, and analogously for $S2AR$ and $A2SR$. □

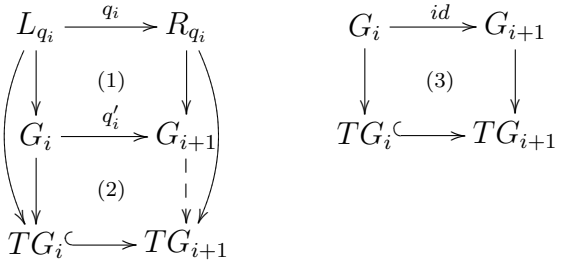
Fact 4.6.2 (*Restriction of* $S2AM$ *to* TG_S)

Given an $S2AM$ transformation $S2AM : VL_S \to VL_A$ based on a layered type-increasing GTS (TG_A, Q) with $TG_S \subseteq TG_A$, then we have: $G_S \overset{Q\ !}{\Longrightarrow} G_A$ with $G_S \in VL_S$ implies $G_A|_{TG_S} = G_S$, i.e. the diagram to the right is a pullback.

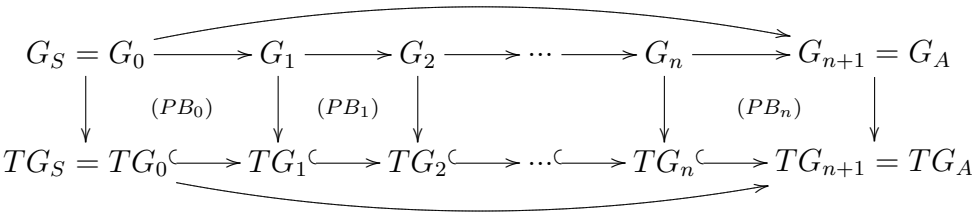


△

Proof: Given $G_S \overset{Q\ !}{\Longrightarrow} G_A$, we can assume to have a sequence $G_S = G_0 \overset{q_0}{\Longrightarrow} G_1 \overset{q_1}{\Longrightarrow} .. \overset{q_n}{\Longrightarrow} G_{n+1} = G_A$ where each q_i is either a parallel rule over Q_i or an identity step. In each single step we have in the first case pushout (1) and the commutative square (2), where the typing $G_{i+1} \to TG_{i+1}$ is induced from $G_i \to TG_i$ and pushout (1), and $L_{q_i} \to TG_i, R_{q_i} \to TG_{i+1}$ are given by our layered type-increasing GTS (TG_A, Q), such that the outer diagram $(1 + 2)$ is a pullback and all horizontal morphisms are monomorphisms.

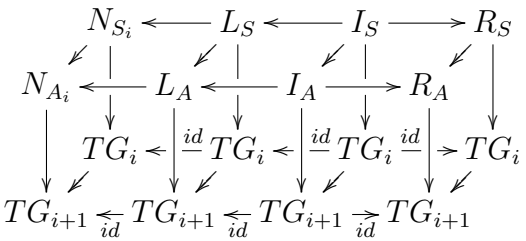


Hence, the special pushout-pullback-decomposition property A.8, implies that (1) and (2) are pullbacks. In the case that $q_i = id$, diagram (3) is a pullback because $TG_i \hookrightarrow TG_{i+1}$ is monomorphism. This leads to the following sequence of pullbacks, which can be composed to one pullback:



□

Fact 4.6.3 (*Restriction of $S2AR$ to TG_S*)
Given an $S2AR$ transformation $S2AR : P_S \to P_A$ based on a layered type-increasing GTS (TG_A, Q) with $TG_S \subseteq TG_A$, then we have: $p_S \overset{Q\,!}{\Longrightarrow} p_A$ with $p_S \in P_S$ implies $p_A|_{TG_S} = p_S$, i.e. for $p_S = (N_{S_i} \leftarrow L_S \leftarrow I_S \rightarrow R_S), p_A = (N_{A_i} \leftarrow L_A \leftarrow I_A \rightarrow R_A)$ the following triple cube commutes with pullbacks in the diagonal squares:

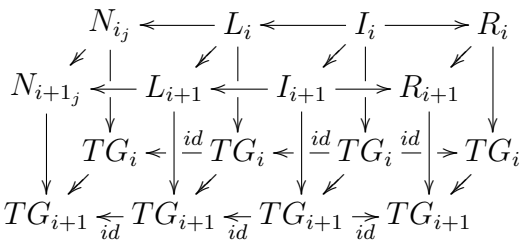


△

Proof: Given $p_S \overset{Q\,!}{\Longrightarrow} p_A$, we consider the subsequences

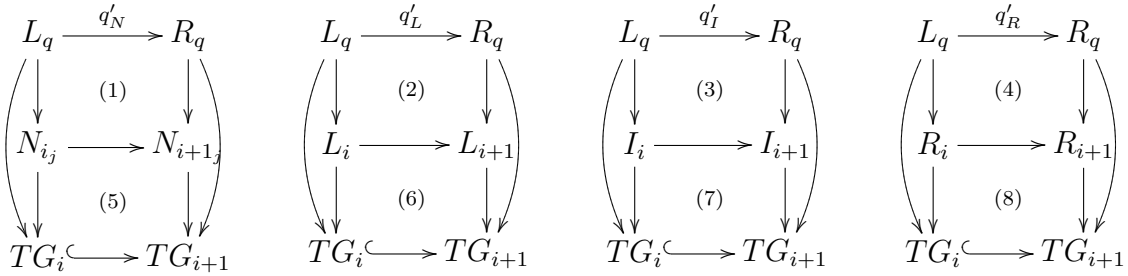
$$p_S = p_0 \overset{Q_0!}{\Longrightarrow} p_1 \overset{Q_1!}{\Longrightarrow} p_2 ... p_n \overset{Q_n!}{\Longrightarrow} p_{n+1} = p_A$$

according to the layers Q_i of Q and show for each $i = 0, .., n$ the following triple cube with pullbacks in the diagonal squares,

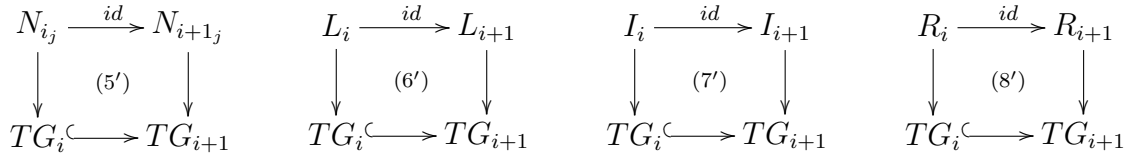


which can be composed to the required triple cube with p_S and p_A. In step $p_i \overset{Q_i!}{\Longrightarrow} p_{i+1}$, we use some rules $q_1, .., q_m \in Q_i$, where according to Def. 4.2.12 we apply $q'_{1N}, .., q'_{mN}$ to N_{i_j}, $q'_{1L}, .., q'_{mL}$ to L_i, $q'_{1I}, .., q'_{mI}$ to I_i, and $q'_{1R}, .., q'_{mR}$ to R_i with $q'_{xy} = q_x$ or $q'_{xy} = id_y$ for $x = 1, .., m$ and $y \in \{N_{i_j}, L_i, I_i, R_i\}$.

Now let q'_L be the parallel rule of all q_{xL} over $X_L = \{x | q'_{xL} \neq id\}$ in the case $X_L \neq \emptyset$, and similarly q'_I over $X_I \neq \emptyset$, q'_R over $X_R \neq \emptyset$ and q'_N over $X_N \neq \emptyset$. In these cases we obtain the diagrams below with pushouts $(1) - (4)$, where the outer diagrams are pullbacks by assumption on Q, leading to pullbacks $(5) - (8)$ by the special pushout-pullback decomposition property A.8.

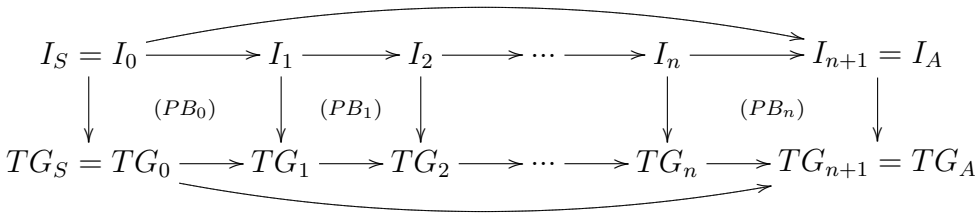


In the case $X_N = \emptyset$, we replace pullback (5) by pullback $(5')$, and analogously for $X_L = \emptyset$ (resp. $X_I = \emptyset$ and $X_R = \emptyset$) we replace pullback (6) by pullback $(6')$ (resp. (7) by $(7')$, and (8) by $(8')$):



In each case we obtain the pullbacks in the diagonal squares in the triple cube above. Moreover, $I_i \to L_i$ (and similarly $L_i \to N_{i_j}$ and $I_i \to R_i$) is the unique TG_i-typed morphism with $I_i \to L_i \to L_{i+1} = I_i \to I_{i+1} \to L_{i+1}$ and $L_i \to N_{i_j} \to N_{i+1_j} = L_i \to L_{i+1} \to N_{i+1_j}$.

Having shown that in each possible case of applying q to a rule p_i, we get the desired pullbacks for the transformation step $p_i \overset{q}{\Longrightarrow} p_{i+1}$, we can now compose the step pullbacks to obtain the pullback of a complete transformation. Such a transformation $p_S \overset{Q}{\Longrightarrow}{}^{!} p_A$ can be decomposed into the steps $p_S = p_0 \overset{q_0}{\Longrightarrow} p_1 \overset{q_2}{\Longrightarrow} ... \overset{q_n}{\Longrightarrow} p_{n+1} = p_A$. The transformation sequence for the interface graphs induces the sequence of pullbacks shown in the diagram below, where each single pullback exists, as was shown above.



Applying the pullback composition property A.7, 3, we conclude that the composed square $(PB_1 + PB_2 + .. + PB_n)$ is a pullback. The composition of the step pullbacks for the other rule graphs works analogously.

□

Remark: Fact 4.6.2 implies that there exists a TGTS embedding $f : \mathbf{SimSpec_{VL_S}} \rightarrow \mathbf{AnimSpec_{VL_A}}$ according to Def. 2.1.24 given by $f = (TG_S \xrightarrow{f_T G} TG_A, P_S \xrightarrow{f_P} P_A)$, where f_{TG} is the type graph inclusion, and f_P maps each simulation rule p_S to the rule p_A resulting from the $S2AR$ transformation. As shown in the proof of Fact 4.6.2, applying the backward retyping functor $f^{<}_{TG_S}$ to an animation rule p_A yields the simulation rule p_S, such that we have $p_S \overset{Q}{\Longrightarrow}{}^{!} p_A$. □

Definition 4.6.4 (*$A2S$ Transformation*)
Given an $S2A$ transformation $S2A = (S2AM : VL_S \rightarrow VL_A, S2AR : P_S \rightarrow P_A) : \mathbf{SimSpec_{VL_S}} \rightarrow \mathbf{AnimSpec_{VL_A}}$, then the transformation $A2S : \mathbf{AnimSpec_{VL_A}} \rightarrow \mathbf{SimSpec_{VL_S}}$ is defined by $A2S = (A2SM, A2SR)$ is called *animation-to-simulation model and rule transformation*, short $A2S$ transformation, where

1. $A2SM : VL_A \rightarrow VL_S$ is the *animation-to-simulation model transformation*, short $A2SM$ transformation, defined by restriction to TG_S, i.e. $A2SM(G_A) = G_A|_{TG_S}$,

   and

2. $A2SR : P_A \rightarrow P_S$ is the *animation-to-simulation rule transformation*, short $A2SR$ transformation, defined by restriction to TG_S, i.e. $A2SR(p_A) = p_A|_{TG_S}$.

△

Theorem 4.6.5 (*$A2S$ Transformation is Backward Transformation of $S2A$ Transformation*)

Given an $S2A$ transformation $S2A = (S2AM : VL_S \rightarrow VL_A, S2AR : P_S \rightarrow P_A)$ based on a layered type-increasing GTS (TG_A, Q) with $TG_S \subseteq TG_A$, then the transformation $A2S : \mathbf{AnimSpec_{VL_A}} \rightarrow \mathbf{SimSpec_{VL_S}}$ defined according to Def. 4.6.4, is a backward transformation of $S2A$ in the sense of the characterization of backward transformations given in Def. 4.6.1. △

Proof: $A2SM : VL_A \to VL_S$ for $G_A \in VL_A$ with $G_S \overset{Q\ !}{\Longrightarrow} G_A$ for $G_S \in VL_S$ maps G_A to G_S, because $G_A|_{TG_S} = G_S$ by Fact 4.6.2. This implies $A2SM \circ S2AM \subseteq ID_{VL_S}$. Analogously, $A2SR : P_A \to P_S$ for $p_A \in P_A$ with $p_S \overset{Q\ !}{\Longrightarrow} p_A$ for $p_S \in P_S$ maps p_A to p_S, because we have $p_A|_{TG_S} = p_S$ by Fact 4.6.3. This implies $A2SR \circ S2AR \subseteq ID_{P_S}$.

Hence, $A2SM, A2SR$ and $A2S = (A2SM, A2SR)$ are backward transformations of $S2AM, S2AR$ and $S2A$, respectively. □

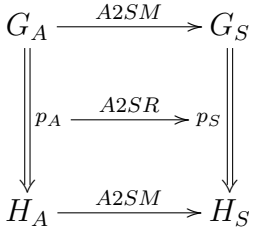
4.6.2 Semantical Correctness of $A2S$ Transformation

Given an $A2S$ backward transformation of $A2S$ with $A2S = (A2SM, A2SR) : \mathbf{Anim-Spec_{VL_A}} \to \mathbf{SimSpec_{VL_S}}$ such that $A2SR(p_A) = p_S$ for $p_A \in P_A, p_S \in P_S$ and $A2SM(G_A) = G_S$ for $G_A \in VL_A, G_S \in VL_S$, then the graph H_S resulting from the simulation step $G_S \overset{p_S}{\Longrightarrow} H_S$ and the graph H_A resulting from the animation step $G_A \overset{p_A}{\Longrightarrow} H_A$ should be related by $A2SM$ backward transformation, i.e. $A2SM(H_A) = H_S$. Furthermore, also the rule applicability should be preserved, i.e. a transformed rule p_S should be applicable exactly to the transformed graph G_S if the original animation rule p_A is applicable to the original graph G_A (NAC-Compatibility).

We show NAC-compatibility of $A2S$ backward transformation in Fact 4.6.7, and semantical correctness in Def. 4.6.6.

Definition 4.6.6 (*Semantical Correctness of $A2S$ Transformation*)

An $A2S$ transformation $A2S : \mathbf{AnimSpec_{VL_A}} \to \mathbf{SimSpec_{VL_S}}$ given by $A2S = (A2SM : VL_A \to VL_S, A2SR : P_A \to P_S)$ is called *semantically correct* if for each animation step $G_A \overset{p_A}{\Longrightarrow} H_A$ with $G_A, H_A \in VL_A$ and $A2SM(G_A) = G_S$ and $A2SR(H_A) = H_S$, there is a corresponding simulation step $G_S \overset{p_S}{\Longrightarrow} H_S$ with $A2SM(H_A) = H_S$.



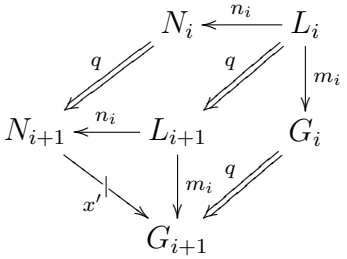
△

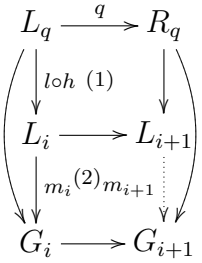
Fact 4.6.7 (*NAC-Compatibility of $A2S$ Transformations*)
An $A2S$ transformation $A2S = (A2SM : VL_A \to VL_S, A2SR : P_A \to P_S)$ is *NAC-compatible* in the following sense: Let $G_S = A2SM(G_A)$ and $p_S = A2SR(p_A)$. Then, if $G_A \overset{p_A}{\Longrightarrow} H_A$ satisfies $NAC(p_A)$ then $G_S \overset{p_S}{\Longrightarrow} H_S$ satisfies $NAC(p_S)$. △

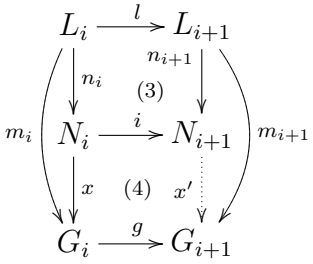
Proof: We know that $L_A \overset{m_A}{\longrightarrow} G_A$ satisfies $NAC(p_A)$. This means, there does not exist an injective graph morphism $x' : N_A \to G_A$ with $x' \circ n_A = m_A$. We must show that then there does not exist an injective graph morphism $x : N_S \to G_S$ with $x \circ n_S = m_S$.

We show that this holds locally for each possible direct transformation $G_i \overset{q}{\Longrightarrow} G_{i+1}$ which is part of the transformation sequence $G_S \overset{Q\ !}{\Longrightarrow} G_A$ with $i \in \{0, .., n\}, G_0 = G_S$ and $G_n = G_A$. Likewise, the corresponding direct transformation step $p_i \overset{q}{\Longrightarrow} p_{i+1}$ is part of the transformation sequence $p_S \overset{Q\ !}{\Longrightarrow} p_A$, where $p_0 = p_S$ and $p_m = p_A$.

Case (1)**, Case** (2):
We have the situation shown in the diagram to the right. We must show that there does not exist an injective graph morphism $x : N_i \to G_i$ with $x \circ n_i = m_i$. We assume we had a morphism $N_i \xrightarrow{x} G_i$ with $x \circ n_i = m_i$ and show that this leads to a contradiction with the precondition that there does not exist a morphism $N_{i+1} \xrightarrow{x'} G_{i+1}$ with $x' \circ n_{i+1} = m_{i+1}$. Let us consider the diagrams below.



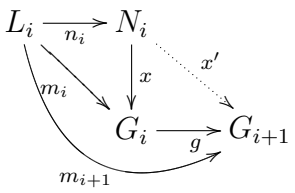




Square (1) and the surrounding square $(1 + 2)$ are pushouts (the transformations $L_i \overset{q}{\Longrightarrow} L_{i+1}$ and $G_i \overset{q}{\Longrightarrow} G_{i+1}$). Thus the morphism $L_{i+1} \xrightarrow{m_{i+1}} G_{i+1}$ is unique, and square (2) is a pushout as well, due to Property A.11.

Square (3) is a pushout due to Def. 4.2.12, Case (1) and Case (2). The surrounding square is pushout (2). Thus we get the unique morphism $N_{i+1} \xrightarrow{x'} G_{i+1}$ such that $x' \circ n_{i+1} = m_{i+1}$ and square (4) is a pushout, due to Property A.11. This is a contradiction to the precondition that there exists no such x'. Thus, the assumption must be wrong, and there exists no morphism $N_i \xrightarrow{x} G_i$ with $x \circ n_i = m_i$.

Case (3):
For this case, we have $L_{i+1} = L_i, N_{i+1} = N_i$ and $n_{i+1} = n_i$. Again, we assume we had a morphism $N_i \xrightarrow{x} G_i$ with $x \circ n_i = m_i$. If there is a match $L_q \to G_i$, we have the transformation $G_i \overset{q}{\Longrightarrow} G_{i+1}$, inducing the inclusion morphism $G_i \xrightarrow{g} G_{i+1}$.



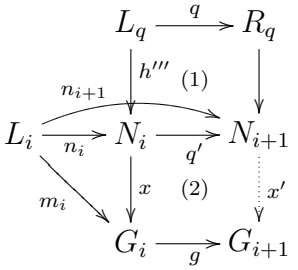
We define the match $L_i \xrightarrow{m_{i+1}} G_{i+1}$ by $m_{i+1} = g \circ m_i$. We now have the situation shown in the diagram to the right. There exists a morphism $N_i \xrightarrow{x'} G_{i+1}$, given by $x' = g \circ x$, for

which holds:

$$\begin{aligned} g \circ x \circ n_i &= g \circ m_i \quad \text{due to precondition} \\ \Leftrightarrow x' \circ n_i &= g \circ m_i \quad \text{due to def. of x'} \\ \Leftrightarrow x' \circ n_i &= m_{i+1} \quad \text{due to def. of } m_{i+1} \end{aligned}$$

This is a contradiction to the precondition that there is no x' with $x' \circ n_i = m_{i+1}$. We conclude that the assumption is wrong, i.e. there does not exist a morphism $N_i \xrightarrow{x} G_i$ with $x \circ n_i = m_i$.

Case (4):
In this case, the only rule graph transformed by q is the NAC N_i. We have the situation shown in the diagram to the right, where the upper square is a pushout and $n_{i+1} = q' \circ n_i$. Again, we assume we had a morphism $N_i \xrightarrow{x} G_i$ with $x \circ n_i = m_i$. Then, we also have a match $L_q \xrightarrow{x \circ h'''} G_i$, and the transformation $G_i \overset{q}{\Longrightarrow} G_{i+1}$. Hence, the surrounding square $(1+2)$ is a pushout. By Property A.11, we get the morphism $N_{i+1} \xrightarrow{x'} G_{i+1}$ such that square (2) is a pushout. We define the match $m_{i+1} = g \circ m_i$. Thus, we have



$$\begin{aligned} g \circ x \circ n_i &= x' \circ q' \circ n_i \quad \text{due to pushout (2)} \\ \Leftrightarrow g \circ m_i &= x' \circ q' \circ n_i \quad \text{due to assumption} \\ \Leftrightarrow g \circ m_i &= x' \circ n_{i+1} \quad \text{due to definition of } n_{i+1} \\ \Leftrightarrow m_{i+1} &= x' \circ n_{i+1} \quad \text{due to definition of } m_{i+1} \end{aligned}$$

This is a contradiction to the precondition that there is no x' with $x' \circ n_{i+1} = m_{i+1}$. We conclude that the assumption is wrong, i.e. there does not exist a morphism $N_i \xrightarrow{x} G_i$ with $x \circ n_i = m_i$. □

Theorem 4.6.8 (*Semantical Correctness of $A2S$ Backward Transformation*)
Each $A2S$ backward transformation $A2S = (A2SM, A2SR)$ of an $S2A$ transformation $S2A = (S2AM, S2AR)$ is semantically correct. △

Proof: The semantical correctness of $A2S$ backward transformation holds due to the fact that the $S2A$ transformation induces a TGTS embedding from S to A (see Remark to Fact 4.6.3). As shown in Theorem 2.1.27, TGTS embeddings reflect the behavior in the sense that if we have the transformation $G_A \overset{p_A, m_A}{\Longrightarrow} H_A$ in A, we get the transformation $G_S \overset{p_S, m_S}{\Longrightarrow} H_S$ such that $m_A|_{TG_S} = m_S$, provided that the restricted NACs N_{S_i} are satisfied. This is the case due to NAC-compatibility of $A2S$, which was shown in Fact 4.6.7. □

4.6.3 Criteria for Semantical Equivalence

Theorem 4.6.9 (*Semantical Completeness of $S2A$ Transformation*)
Given an $S2A$ transformation based on a layered type-increasing $S2A$ transformation system (TG_A, Q), then $S2A$ is semantically complete in the sense that there is a semantically correct backward transformation $A2S$ of $S2A$. △

Proof: The proof is a direct consequence of Theorem 4.6.5 and Theorem 4.6.8. □

Definition 4.6.10 (*Semantical Equivalence of Simulation and Animation Specifications*)

An $S2A$ transformation $S2A = (S2AM, S2AR) : \mathbf{SimSpec_{VL_S}} \rightarrow \mathbf{AnimSpec_{VL_A}}$ is called *semantical equivalence of* $\mathbf{SimSpec_{VL_S}}$ *and* $\mathbf{AnimSpec_{VL_A}}$ if $S2A$ is semantically correct and complete, and we have in addition:

$$A2S \circ S2A = Id \text{ and } S2A \circ A2S = Id$$

for $A2S = (A2SM, A2SR)$ backward transformation of $S2A$. △

To show that an $S2A$ transformation $S2A = (S2AM : VL_S \rightarrow VL_A, S2AR : P_S \rightarrow P_A)$ is a *semantical equivalence of* $\mathbf{SimSpec_{VL_S}}$ *and* $\mathbf{AnimSpec_{VL_A}}$, we need a few structural properties of $S2A$ and $A2S$ transformations.

Lemma 4.6.11 (*Properties of $S2AM$ Transformation*)
$S2AM$ transformation $S2AM : VL_S \rightarrow VL_A$ considered as relation based on the $S2A$ transformation system $GTS_{S2A} = (TG_A, Q)$ is

1. left total if $S2AM$ is terminating,
2. right total,
3. left unique if GTS_{S2A} is layered type-increasing,
4. right unique if GTS_{S2A} is confluent.

△

Proof:

1. $S2AM$ terminating implies $\forall G_S \in VL_S : G_S \overset{Q\ !}{\Longrightarrow} G_A$;
2. by definition of $VL_A = \{G_A | \exists G_S \in VL_S : G_S \overset{Q\ !}{\Longrightarrow} G_A\}$;

3. by Fact 4.6.2;
4. Given $G_S \overset{Q\ !}{\Longrightarrow} G_A$ and $G_S \overset{Q\ !}{\Longrightarrow} G'_A$, then confluence implies $G_A = G'_A$ (up to isomorphism).

□

Lemma 4.6.12 (*Properties of $S2AR$ Transformation*)
$S2AR$ transformation $S2AR : P_S \rightarrow P_A$ considered as relation based on the $S2A$ transformation system $GTS_{S2A} = (TG_A, Q)$ is

1. left total if $S2AR$ is terminating,
2. right total,
3. left unique if GTS_{S2A} is layered type-increasing,
4. right unique if $S2AR$ is confluent.

△

Proof:

1. $S2AR$ terminating implies $\forall p_S \in P_S : \exists\, p_S \overset{Q\ !}{\Longrightarrow} p_A$;
2. by definition of $P_A = \{p_A | \exists p_S \in P_S :\ p_S \overset{Q\ !}{\Longrightarrow} p_A\ \}$;
3. by Fact 4.6.3;
4. Given $p_S \overset{Q\ !}{\Longrightarrow} p_A$ and $p_S \overset{Q\ !}{\Longrightarrow} p'_A$, then confluence of $S2AR$ implies $p_A = p'_A$ (up to isomorphism).

□

Lemma 4.6.13 (*Properties of $A2SM$ Transformation*)
An $A2SM$ backward transformation of $S2AM$, given by $A2SM : VL_A \rightarrow VL_S$ and defined by restriction with VL_A based on the $S2A$ transformation system $GTS_{S2A} = (TG_A, Q)$, is considered as relation, and is

1. left total if GTS_{S2A} is layered type-increasing,
2. right total if $S2AM$ is terminating and GTS_{S2A} layered type-increasing,
3. left unique if GTS_{S2A} is confluent and layered type-increasing,
4. right unique.

△

Proof:

1. by Fact 4.6.2, we have for each $G_A \in VL_A$: $G_A|_{TG_S} = G_S \in VL_S$ for $G_S \overset{Q\ !}{\Longrightarrow} G_A$;
2. GTS_{S2A} terminating implies $\forall G_S \in VL_S : G_S \overset{Q\ !}{\Longrightarrow} G_A$ with $G_A|_{TG_S} = G_S$;
3. for $G_A, G'_A \in VL_A$ with $G_A|_{TG_S} = G'_A|_{TG_S}$ we have by definition of VL_A that for $G_S, G'_S \in VL_S : G_S \overset{Q\ !}{\Longrightarrow} G_A, G'_S \overset{Q\ !}{\Longrightarrow} G'_A$, and by Fact 4.6.2 $G_A|_{TG_S} = G_S$ and $G'_A|_{TG_S} = G'_S$, which implies $G_S = G'_S$, and by confluence of GTS_{S2A}, also $G_A = G'_A$;
4. because restriction construction is well-defined.

□

Lemma 4.6.14 (*Properties of $A2SR$ Transformation*)
An $A2SR$ backward transformation of $S2AR$, given by $A2SR : P_A \to P_S$ and defined by restriction with P_A based on the $S2A$ transformation system $GTS_{S2A} = (TG_A, Q)$, is considered as relation, and is

1. left total if GTS_{S2A} is layered type-increasing,
2. right total if $S2AR$ is terminating and GTS_{S2A} layered type-increasing,
3. left unique if $S2AR$ is confluent and GTS_{S2A} layered type-increasing,
4. right unique.

△

Proof:

1. by Fact 4.6.3 we have for each $p_A \in P_A$: $p_A|_{TG_S} = p_S \in P_S$ for $p_S \overset{Q\ !}{\Longrightarrow} p_A$;
2. GTS_{S2A} terminating implies $\forall p_S \in P_S :\ p_S \overset{Q\ !}{\Longrightarrow} p_A$ with $p_A|_{TG_S} = p_S$;
3. for $p_A, p'_A \in P_A$ with $p_A|_{TG_S} = p'_A|_{TG_S}$ we have by definition of P_A $p_S, p'_S \in P_S$, $p_S \overset{Q\ !}{\Longrightarrow} p_A$, $p'_S \overset{Q\ !}{\Longrightarrow} p'_A$, and by Fact 4.6.3 $p_A|_{TG_S} = p_S$ and $p'_A|_{TG_S} = p'_S$, which implies $p_S = p'_S$, and by confluence of $S2AR$, also $p_A = p'_A$;
4. because restriction construction is well-defined.

□

Theorem 4.6.15 (*Semantical Equivalence of Simulation and Animation Specifications*) An $S2A$ transformation $S2A = (S2AM : VL_S \to VL_A, S2AR : P_S \to P_A)$ based on a layered type-increasing $S2A$ transformation system $GTS_{S2A} = (TG_A, Q)$ has a backward transformation $A2S = (A2SM, A2SR)$ defined by restriction to TG_S leading to a semantical equivalence of $\mathbf{SimSpec_{VL_S}}$ and $\mathbf{AnimSpec_{VL_A}}$ if

- $S2A$ is rule compatible,
- $S2AM$ is terminating,
- GTS_{S2A} is layered type-increasing,
- $S2AM$ and $S2AR$ are confluent.

△

Proof: By Theorems 4.5.11 and 4.6.9 we have that $S2A$ is semantically correct and complete with $A2S \circ S2A \subseteq Id$. Moreover, we have $A2S \circ S2A = Id$ by Lemmas 4.6.11, 1, and 4.6.12, 1, and $S2A \circ A2S = Id$ by Lemmas 4.6.11 - 4.6.14. □

Remark: Vice versa, $A2S \circ S2A = Id$ and $S2A \circ A2S = Id$ implies that $A2S$ and $S2A$ are left and right total and also left and right unique. □

Fact 4.6.16 (*Example Echo Model: Semantical Equivalence*) The $S2A$ transformation $S2A = (S2AM, S2AR)$ based on the $S2A$ transformation for the Echo model, where the $S2A$ transformation rules Q are shown in Fig. 4.4, and the simulation rules P_S are given in Fig. 3.13, leads to a semantical equivalence of $\mathbf{SimSpec_{VL_S}}$ and $\mathbf{AnimSpec_{VL_A}}$. △

Proof: Termination of $S2A$ has been shown to be fulfilled for general type-increasing $S2A$ transformation systems in Theorems 4.3.5 and 4.3.6. The Echo $S2A$ transformation is semantically correct due to Fact 4.5.12. Furthermore, due to Theorem 4.6.5, we have a backward transformation $A2S : \mathbf{AnimSpec_{VL_A}} \to \mathbf{SimSpec_{VL_S}}$, defined by type restrictions, such that the $S2A$ transformation is also semantically complete. In addition to semantical completeness, according to Theorem 4.6.15, the requirements for an $S2A$ transformation being a semantical equivalence, are rule compatibility and confluence. These requirements have been shown for the Echo $S2A$ transformation in Fact 4.5.10 (rule compatibility) and Fact 4.4.1 (confluence). □

4.7 Applications

In addition to the running example of this chapter, the Echo Algorithm, in this section we consider three more animation view definitions and show that they are semantically equivalent to their corresponding simulation specifications. In Section 4.7.1, we consider an animation view for the AHL net modeling the Dining Philosophers from Chapter 3.5.1. The behavior of the Dining Philosopher model is animated as five people around a table with chopsticks. An earlier version of this animation view was the running example in our paper [EE05b]. In Section 4.7.2, an animation view is developed for the Radio Clock Statechart introduced in Chapter 3.1.2, Example 3.1.2. The different modes of the clock are visualized by different displays showing the date, the time and the alarm time. Moreover, the part of the alarm time is high-lighted which can be changed in the current state. The last application, presented in Section 4.7.3, deals with an animation view for the Drive-Through UML model introduced in Chapter 3.5.3, where the behavior of the Drive-Through model is animated as a queue of cars in front of a restaurant building. A short version of this application (without considering the semantical correctness) is also described in our paper [EHKZ05]. More applications not given in detail in this thesis have been published in our papers [EET06] (the animation view for an AHL net modeling an elevator), and in [EB04, BEE+02] (showing the behavior of a finite automaton modeling a producer-consumer system in the animation view of a kitchen, where the producer is a cook, the buffer is a table, and the consumer is a person sitting at that table and consuming the produced goods (cakes)).

4.7.1 Animation View for the AHL Net *Dining Philosophers*

In this section we develop an animation view for the AHL net modeling the Dining Philosophers from Chapter 3.5.1.

Simulation Specification

The upper part of Fig. 4.13 shows the AHL Net modeling the behavior of five dining philosophers, who sit around a table. Between each two philosophers there is one chopstick lying on the table. A philosopher who is eating uses the two chopsticks to his right and to his left. While a philosopher is not eating, he is thinking. We identify the five philosophers as well as their chopsticks by numbers. Fig. 4.13 shows the initial situation where all philosophers are thinking and all chopsticks are lying on the table.

The abstract syntax graph of the net is the modeled by the graph G_I in the lower part of Fig. 4.13, which is also the initial state for our dining philosopher simulation.

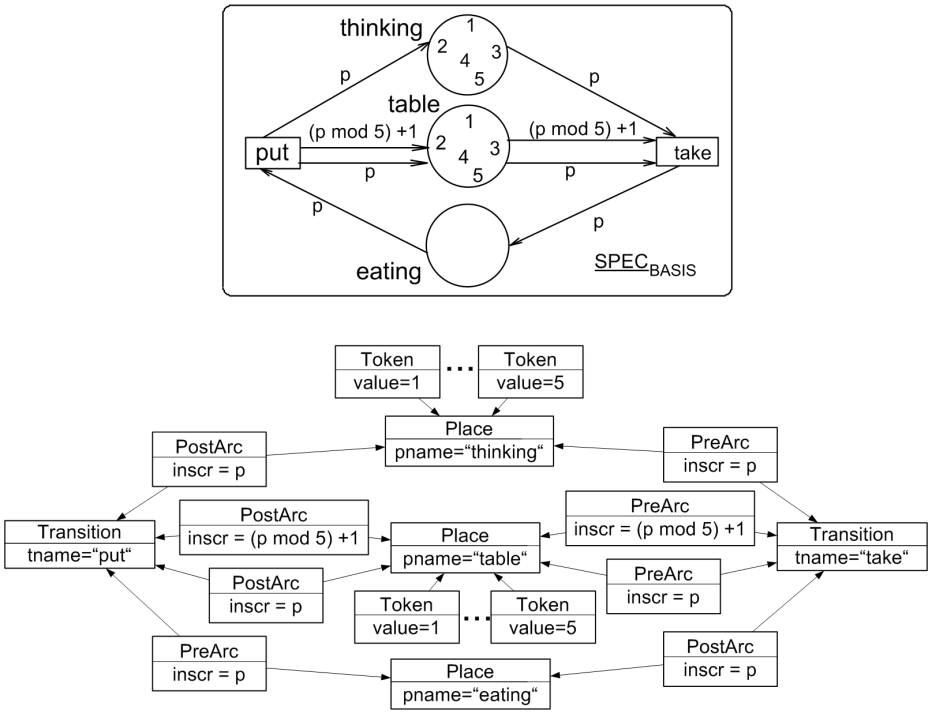


Figure 4.13: Initial State of a Dining Philosophers Simulation (Concrete Notation and Abstract Syntax)

The model-specific simulation rules for the AHL net *Dining Philosophers* have been introduced already in Chapter 3.5.1.

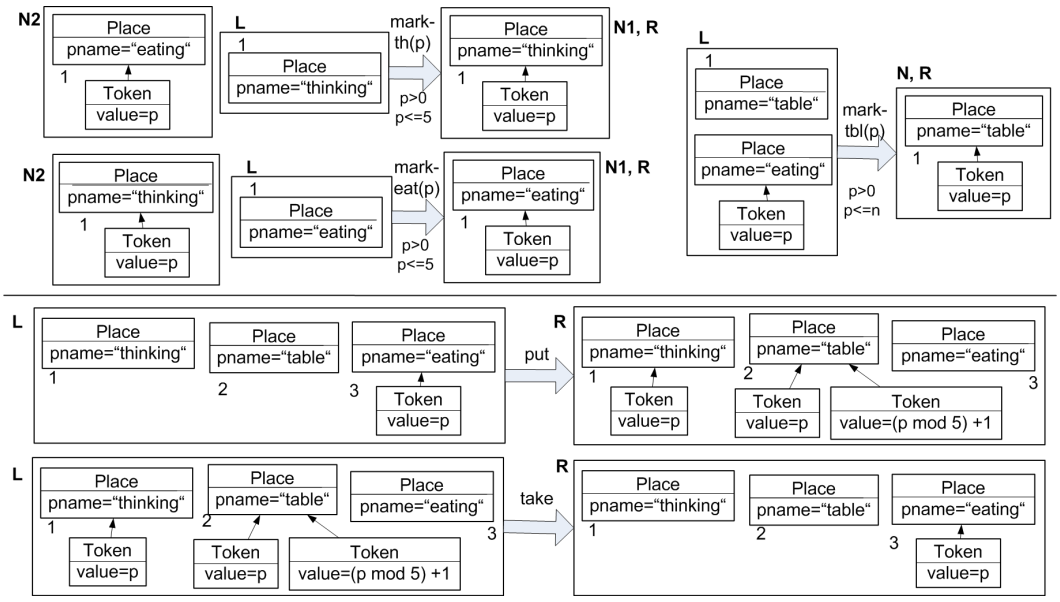


Figure 4.14: Simulation Rules for the Dining Philosophers

We add a first rule layer for the generation of the initial marking. In order to allow more than one initial state for simulation runs, the rules mark-th(n), mark-tbl(n) and mark-eat(n) model the marking process of the three places. The second layer contains the two remaining rules which model the firing behavior of the two transitions put and take.

The simulation specification $\mathbf{SimSpec_{VL_S}} = (VL_S, P_S)$ consists of the simulation language VL_S typed over TG_S, where TG_S is the simulation alphabet depicted in the left part of Fig. 4.15, P_S is the set of simulation rules shown in Fig. 4.14, and VL_S consists of all graphs that can occur in any Dining Philosophers simulation scenario: $VL_S = \{G_S | \exists G_0 \overset{P_S*}{\Longrightarrow} G_S\}$, where G_0 is the graph corresponding to G_I, but without token symbols. Note that G_I is an element of VL_S which can be obtained by applying the marking rules mark-th and mark-tbl with adequate input parameter values.

Animation Alphabet

Fig. 4.15 shows the animation alphabet TG_A, which is a disjoint union of the simulation alphabet TG_S and a new visualization alphabet TG_V which is shown in the right part of Fig. 4.15, and which models the elements for a domain-specific visualization of the dining philosophers' behavior. (Since we do not need the concrete syntax of the AHL nets to define the animation view, we do not depict it in Fig. 4.15.)

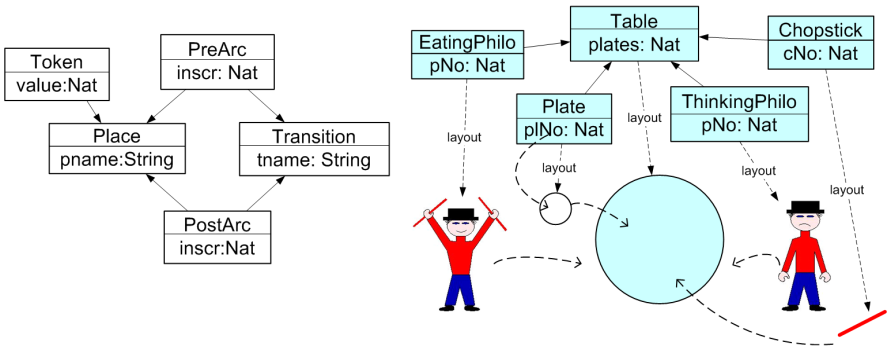


Figure 4.15: Animation Alphabet for the Dining Philosophers

The symbol Table is the root symbol of the animation. and belongs to type layer 1. On the table, there are the plates, one for each philosopher in the net, layouted in a circle. Their layout positions are computed depending on the number of plates (philosophers / chopsticks) represented in the attribute plates. The size of the angle between two plates is computed by dividing the table circle in plates segments of the same size. The plate position then is computed by multiplying the angle size and the respective plate number plNo. The philosophers are visualized by two different graphics: a happy philosopher holding two chopsticks, and a philosopher without chopsticks, looking thoughtful. The layout positions of the philosopher graphics and the chopsticks are computed like the plate

positions. Philosophers are positioned with a fixed distance from the plate, outside the table, and the chopsticks are positioned within the table circle with an angular offset besides the plate.

The *Dining Philosophers* animation alphabet is a valid animation alphabet according to Def. 4.2.4, where the type Table has type layer 1, type Plate has type layer 2, and the remaining types belong to type layer 3.

$S2A$ Transformation

The $S2A$ transformation rules Q, shown in Fig. 4.16, add corresponding visualization elements to the simulation rules and to the initial graph corresponding to the unmarked net.

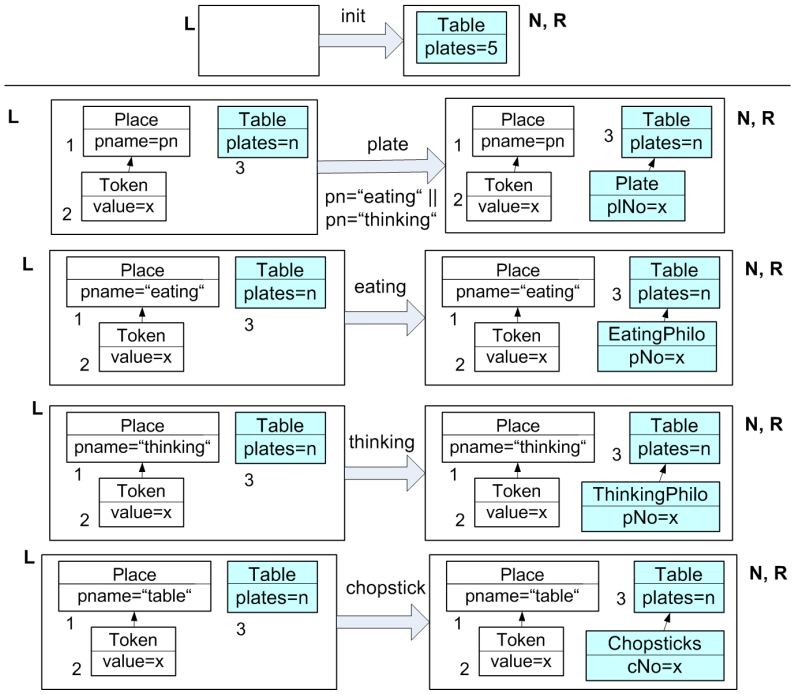


Figure 4.16: $S2A$ Rules for the Dining Philosophers

$S2A$ rule init (rule layer 0) initializes the animation view part by adding a Table symbol to all graphs it is applied to. The input parameter fixes the number of philosophers, plates and chopsticks to be layouted.

The remaining $S2A$ rules all belong to rule layer 1. Rules plates, eating and thinking generate a plate symbol or, respectively, a philosopher graphic for each token on either place Eating or place Thinking. Rule chopsticks adds a chopstick icon for each token on place Table.

All $S2A$ transformation rules are typed over TG_A, each $S2A$ transformation rule has a NAC which equals its RHS. Moreover, all rules within the same rule layer are parallel

independent, as none of them generates elements which are forbidden by the NACs of the other rules in the layer. Hence, together with their rule and type layers, the $S2A$ rules comprise a valid, type-increasing $S2A$ transformation system according to Def. 4.2.8.

The Dining Philosophers $S2AM$ transformation $S2AM : VL_S \rightarrow VL_A$ is hence given by $S2AM = (VL_S, TG_A, Q)$, where the animation language $VL_A = \{G_A | \exists G_S \in VL_S : G_S \overset{Q}{\Longrightarrow}{}^{*} G_A\}$.

We consider a sample $S2A$ transformation sequence which transforms the simulation rule take. In the first $S2A$ transformation step, only the $S2A$ rule table is applicable. It is applied according to Case (1) of Def. 4.2.12 to all three rule graphs of rule put, which results in an intermediate rule take_0. Now, several rules from rule layer 2 become applicable to rule take_0: rule plate according to Case (2) and Case (3), rule thinking according to Case (2), rule eating according to Case (3), and rule chopstick according to Case (2) at two different matches. Considering the match priorities, we start with the Case (2) applications: The application of rule plate adds a Plate symbol with plNo=p to the LHS of rule take_0, resulting in rule take_1. The application of rule thinking to rule take_1 adds a ThinkingPhilo symbol to the LHS of take_1, resulting in rule take_2. Applying rule chopstick first to rule take_2 and then to the result of this application, rule take_3 adds one Chopstick symbol with cNo=p and another one with cNo=(p mod 5) +1, resulting in rule take_4. Now, no more $S2A$ rules are applicable according to Case (2), but we have two more Case (3) - rule applications: The application of rule plate to rule take_4 adds a Plate symbol to its RHS (which yields rule take_5), and the application of rule eating to rule take_5 adds an EatingPhilo symbol to its RHS, resulting in rule take_6, which is the final result, the animation rule. This last transformation step is depicted in Fig. 4.25.

The Dining Philosophers $S2AR$ rule transformation $S2AR : P_S \rightarrow P_A$ is given by $S2AR = (P_S, TG_A, Q)$, where the animation rules $P_A = \{p_A | \exists p_S \in P_S : p_S \overset{Q}{\Longrightarrow}{}^{!} p_A\}$.

The Dining Philosophers animation specification $\mathbf{AnimSpec_{VL_A}} = S2A(\mathbf{SimSpec_{VL_S}})$ based on the $S2A$ transformation $S2A = (S2AM, S2AR)$ is given by $\mathbf{AnimSpec_{VL_A}} = (VL_A, P_A)$, where VL_A is the animation language obtained by the Dining Philosophers $S2AM$ transformation, and P_A are the animation rules obtained by the Dining Philosophers $S2AR$ transformation, some of which are shown in Fig. 4.18.

Fig. 4.27 shows an animation scenario by applying the animation rules from Fig. 4.18, starting with the animation view corresponding to the initial state of the Dining Philosophers net in Fig. 4.13.

Correctness and Completeness of the $S2A$ Transformation

To ensure the semantical correctness of the Dining Philosophers $S2A$ transformation, we have to check the NAC-compatibility. of the $S2AM$ transformation.

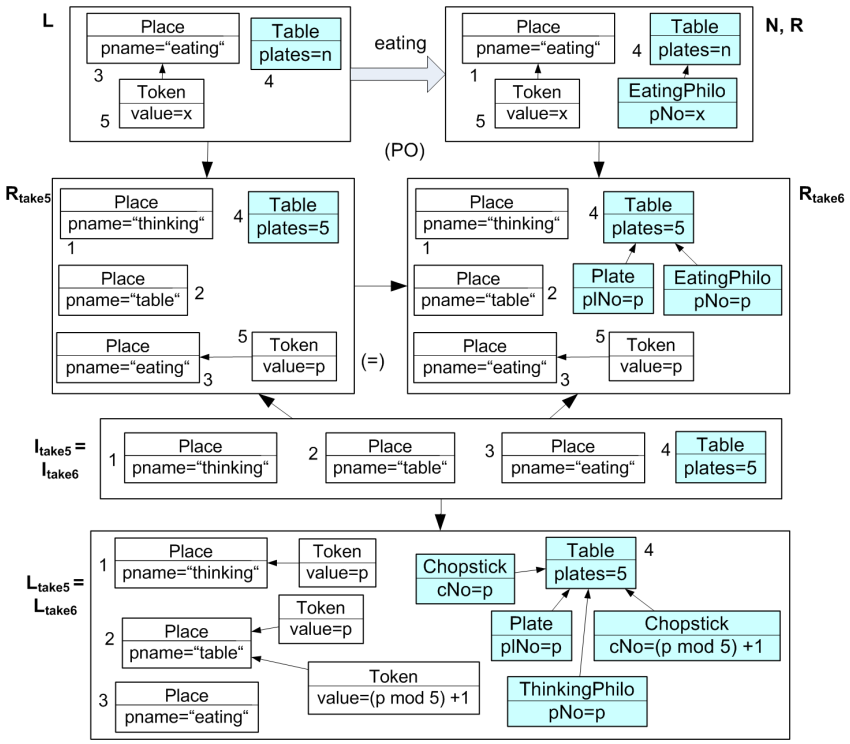


Figure 4.17: Application of $S2A$ Rule eating to Rule take$_5$

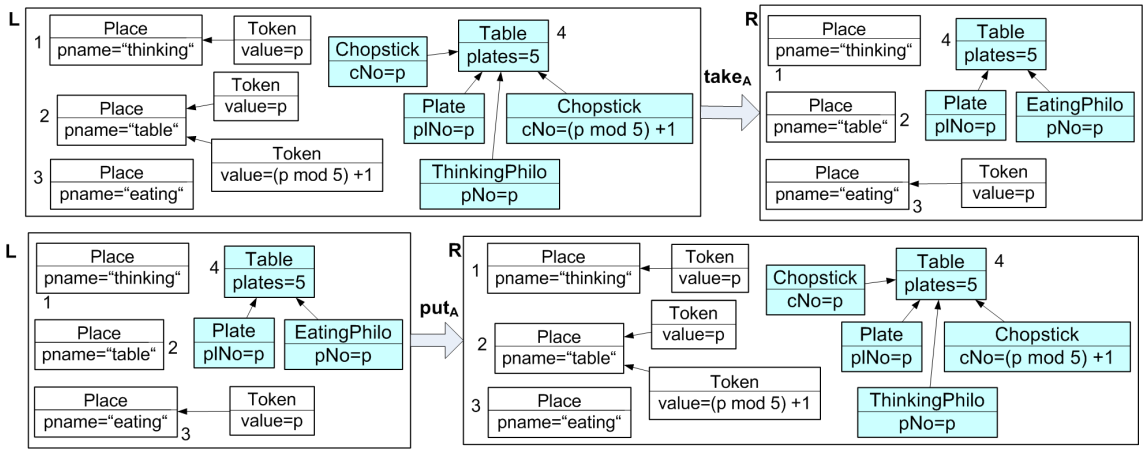


Figure 4.18: Animation Rules for the Dining Philosophers

Fact 4.7.1 (*NAC-Compatibility of Dining Philosophers $S2AM$ Transformation*) The Dining Philosophers $S2AM$ transformation is NAC-compatible (see Def. 4.5.2) in the following sense: For all $p_i \overset{q}{\Longrightarrow} p_{i+1}$ with $q = (L_q \overset{q}{\longrightarrow} R_q)$ and $NAC_q = (L_q \overset{q}{\longrightarrow} R_q)$ such that the match from q to p_i satisfies NAC_q, the following $S2AM$ steps also satisfy NAC_q according to the rule transformation cases below:

Case (1): $G_i \overset{q}{\Longrightarrow} G_{i+1}$ and $H_i \overset{q}{\Longrightarrow} H_{i+1}$,

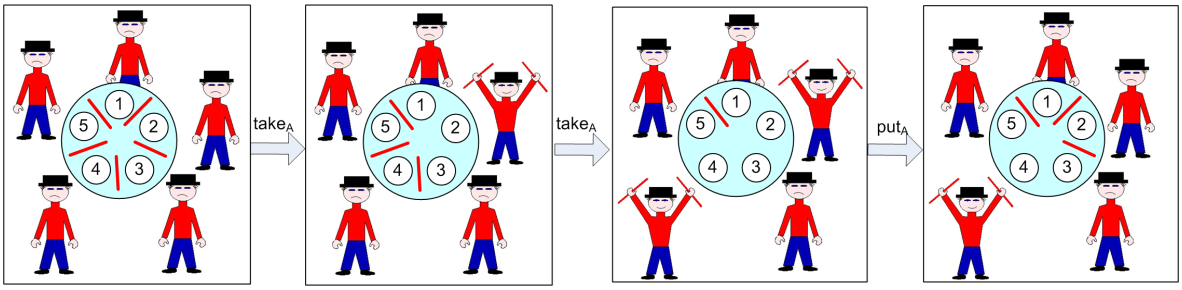


Figure 4.19: Animation Scenario of the Dining Philosophers

Case (2): $G_i \stackrel{q}{\Longrightarrow} G_{i+1}$,
Case (3): $H_i \stackrel{q}{\Longrightarrow} H_{i+1}$. △

Proof: We show for all $q \in Q$ that for a match $L_q \to X$ there is no NAC-morphism $(R_q - L_q) \stackrel{x}{\longrightarrow} X$. Due the property of all $q \in Q$ being type-increasing, in this case NAC_q is satisfied for this match.

The only $S2A$ rule which can be applied to any rule p_i according to Case (1) is rule init. As rule init belongs to rule layer 1, all rules p_i it can be applied to, are the original simulation rules, and do not contain symbols typed over TG_V. Hence, a step involving the application of init to a rule p_i is always NAC-compatible, since

- the match $L_q \stackrel{h}{\longrightarrow} I_{p_i} \stackrel{l_{p_i}}{\longrightarrow} L_{p_i} \stackrel{m_{p_i}}{\longrightarrow} G_i$ satisfies NAC_q as G_i does not contain TG_V-typed elements, and hence there is no NAC-morphism $(R_q - L_q) \stackrel{x}{\longrightarrow} G_i$;

- the match $L_q \stackrel{h}{\longrightarrow} I_{p_i} \stackrel{r_{p_i}}{\longrightarrow} R_{p_i} \stackrel{m^*_{p_i}}{\longrightarrow} H_i$ satisfies NAC_q as H_i does not contain TG_V-typed elements, and hence there is no NAC-morphism $(R_q - L_q) \stackrel{x}{\longrightarrow} H_i$;

All subsequent $S2A$ transformation steps are either according to Case (2), Case (3) or Case (4). Note that the right-hand sides of all $S2A$ rules do not overlap in their generated elements, i.e. in $(R_q - L_q)$.

Let us consider a step according to Case (2), first: We assume that q is applicable to p_i, i.e. there is a match $L_q \stackrel{h}{\longrightarrow} L_i$ satisfying NAC_q. Now, if NAC_q is not satisfied for the match $L_q \stackrel{h}{\longrightarrow} L_i \stackrel{m}{\longrightarrow} G_i$, then this means that q must have been applied before to another rule p_j according to Case (2) with $p_j \stackrel{q}{\Longrightarrow} p_{j+1} \Longrightarrow \cdot\cdot \Longrightarrow p_i$ with $j < i$, since q is the only $S2A$ rule which could add the elements $(R_q - L_q)$ to G_i. But in this case, we have a NAC-morphism $(R_q - L_q) \to L_{j+1} \to L_i$ which is a contradiction to our assumption that NAC_q is satisfied for the match $L_q \to L_i$. Hence, NAC_q must be satisfied for the match $L_q \stackrel{h}{\longrightarrow} L_i \stackrel{m}{\longrightarrow} G_i$.

Analogously, we can argue for the Case (3) and Case (4) steps. □

Fact 4.7.2 (*Rule Compatibility of Dining Philosophers $S2A$ Transformation*)
The Dining Philosophers $S2A$ transformation is rule compatible in the sense of Def. 4.5.9, i.e. all p_A and all q are parallel and sequential independent. △

Proof: If p_A is applicable to a graph G, then there is a match $L_A \xrightarrow{m} G$. Therefore, symbols of at least those types from TG_V that are contained in L_A have also to be contained in G. So, in the sequence $G_S \overset{Q*}{\Longrightarrow} G$ there have been applied at least those rules $q \in Q$ which have also been applied in $p_S \overset{Q\ !}{\Longrightarrow} p_A$ according to Case (1) or (2) (i.e. applied to some L_i, $i = 0, .., n$). All those rules q have a match $L_q \to L_A \to G$ but are not applicable any more to L_A because of their NACs NAC_q. Neither are they applicable to G, due to NAC-compatibility.

So we have to consider only those overlappings L_A/L_q, where $h(L_q)$ is not completely included in $m(L_A)$. As the left-hand side of $S2A$ rule init is empty, we do not have to consider it at all. Moreover, there exists exactly one instance of type Table in graph G, in all L_q and in all L_A. Hence, all pairs L_A/L_q overlap in the Table symbol. However, this is uncritical, as the Table symbol is always preserved by all rules. Furthermore, there exists at most exactly one Place node named thinking, one Place node named eating, and one Place node named table. So, all pairs L_A and L_q which both contain a Place node with the same name, overlap in this node. Again, this is uncritical as Place symbols are always preserved by all rules. The only critical symbols L_A and L_q could overlap at, are the Token symbols. There is exactly one Token symbol in the LHS of all $S2A$ rules. This Token symbol is the last node apart from the Table and the Place node. As we have argued before, L_A and L_q overlap already in the Table and in the Place node. If they would overlap also in the Token node, they would overlap completely, i.e. $h(L_q)$ would be included in $m(L_A)$. In this case, q would not be applicable as shown above. If L_A and L_q do not overlap in the Token nodes, they only overlap in uncritical nodes that are preserved by the rules, which are therefore parallel independent.

Due to the Local Church Rosser Theorem 2.1.19, we know that if $G \overset{p_A}{\Longrightarrow} H$ and $G \overset{q}{\Longrightarrow} G'$ are parallel independent (which was shown above), then $G \overset{p_A}{\Longrightarrow} H$ and $H \overset{q}{\Longrightarrow} H'$ are sequential independent. □

Fact 4.7.3 (*Confluence of Dining Philosophers $S2A$ Transformation*)
The $S2A$ transformation $S2A = (S2AM, S2AR)$ based on the $S2A$ transformation system (TG_A, Q) for the Dining Philosophers model, where the $S2A$ transformation rules Q are shown in Fig. 4.16, is confluent. △

Proof: The Dining Philosophers model contains always exactly one instance of the symbol type Table and one instance of each Plate symbol with the same name. Moreover, we have at most one instance of a Token symbol with the same value number. These type constraints

eliminate the potential conflicts which are found by AGG (see Fig. 4.20) in the following way: Consider the three critical pairs in Fig. 4.20 found for application of rule plate and again rule plate.

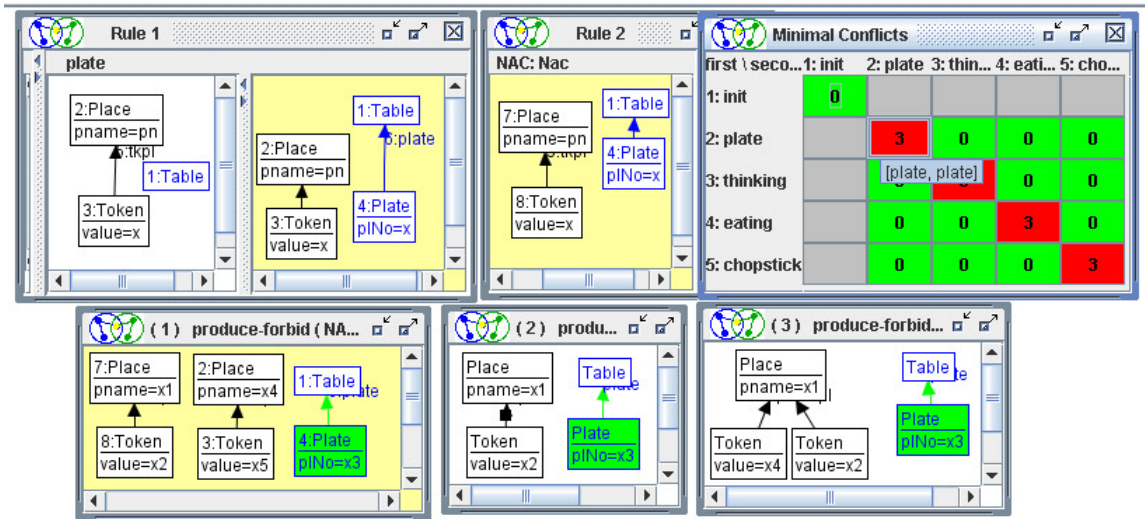


Figure 4.20: Critical Pairs of the Dining Philosophers $S2AM$ Transformation

They are all produce-forbid-conflicts, i.e. the NAC of the second rule plate forbids its application after its first application. This situation is intended if the two matches overlap completely (critical pair (2)), because we do not want the rule to be applied again at the same essential match. Hence, Critical Pair (2) does not lead to an unwanted conflict. Critical Pairs (1) and (3) show a situation where the RHS of the firstly applied rule plate and the NAC of the secondly applied rule plate overlap only in the two nodes Table and Plate. Since they overlap in Plate, both Token nodes must have the same value. In our model, two different Token nodes can never have the same value. Moreover, a Token node is always connected to only one Place node. Hence, Critical Pairs (1) and (3) do not lead to conflicts as the depicted overlappings can never occur. Analogously, the 3 Critical Pairs found for each of the other rule pairs can be analyzed and lead to no unwanted conflicts.

Hence, $S2AM$ transformation is confluent.

Regarding $S2AR$ transformation, we have to check what happens if two $S2A$ transformation rules q_1 and q_2 from the same rule layer are applied to the same rule p_i with $i = 0, .., n$ and $p_S = p_0, p_n = p_A$ in $p_S \overset{Q,!}{\Longrightarrow} p_A$. If they are applied according to the same rule transformation case (see Def. 4.2.12), i.e. they change the same rule graph(s), then we have a conflict which can always be resolved, since q_1 and q_2 are parallel independent by Definition 4.2.8. If they are applicable according to different rule transformation cases, then the match priorities define an application order, such that we will not get conflicts. □

Fact 4.7.4 (*Semantical Equivalence of* $SimSpec_{VL_S}$ *and* $AnimSpec_{VL_A}$)
The $S2A$ transformation $S2A = (S2AM, S2AR)$ based on the $S2A$ transformation system (TG_A, Q) for the Dining Philosophers model, where the $S2A$ transformation rules Q are shown in Fig. 4.16, is a semantical equivalence of $\mathbf{SimSpec_{VL_S}}$ and $\mathbf{AnimSpec_{VL_A}}$. $\triangle$

Proof: Termination has been shown to be fulfilled for general type-increasing $S2A$ transformation systems in Theorems 4.3.5 and 4.3.6). Since the Dining Philosophers $S2A$ transformation is NAC-compatible due to Fact 4.7.1, it is hence also semantically correct due to Theorem 4.5.11. Furthermore, due to Theorem 4.6.5, we have a backward transformation $A2S : \mathbf{AnimSpec_{VL_A}} \rightarrow \mathbf{SimSpec_{VL_S}}$, defined by type restrictions, such that the $S2A$ transformation is also semantically complete. In addition to semantical completeness, according to Theorem 4.6.15, the requirements for an $S2A$ transformation being a semantical equivalence, are rule compatibility and confluence. These requirements have been shown for the Dining Philosophers $S2A$ transformation in Fact 4.7.2 (rule compatibility) and Fact 4.7.3 (confluence). □

4.7.2 Animation View for the Statechart Modeling a Radio Clock

In this section we develop an animation view for the Radio Clock Statechart introduced in Chapter 3.1.2, Example 3.1.2.

Simulation Specification

The upper part of Fig. 4.21 shows the Statechart modeling the behavior of a radio clock, which can show alternatively the time or the date or allows to set the alarm time. The changes between the modes (realized by pressing a *Mode* button on the clock) are modeled in the Statechart by transitions labeled with the event Mode. The nested state Alarm allows to switch between modes for setting the hours and the minutes (realized by pressing a *Select* button on the clock) which is modeled by the transitions labeled with the event Select. The Set event increments the number of hours or minutes which are currently displayed. The abstract syntax graph of the Radio Clock is the modeled by the graph G_I in the lower part of Fig. 4.21, which is also the initial state for the radio clock simulation.

The model-specific simulation rules for the Radio Clock Statechart have been introduced already in Chapter 3.4.0.1. For simplicity, we here restrict ourselves to a set of simulation rules without the intermediate link, and without rule priorities, and require that the user selects only terminating simulation scenarios without loops where an up rule is immediately followed by a down rule. The set of model-specific simulation rules is shown in Fig. 4.22.

In the first rule layer, the initialization of an event queue is realized by the rules initial(h,m,e) and addEvent(e) which generate the object, set its current pointer to the top level

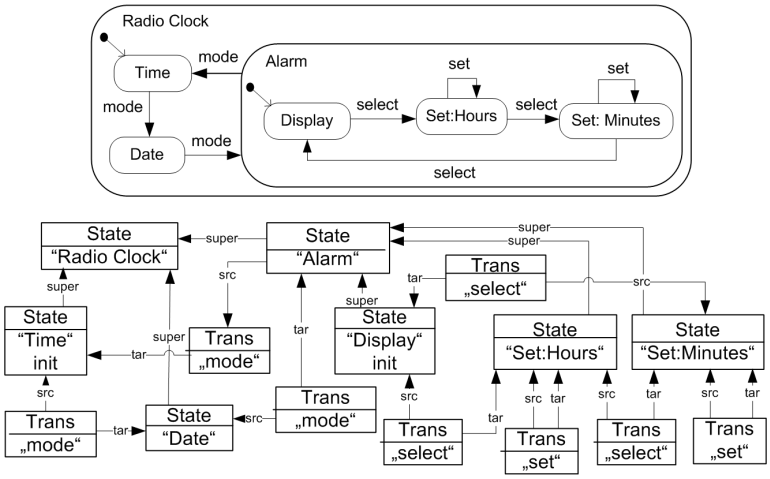


Figure 4.21: Initial State of a Radio Clock Simulation

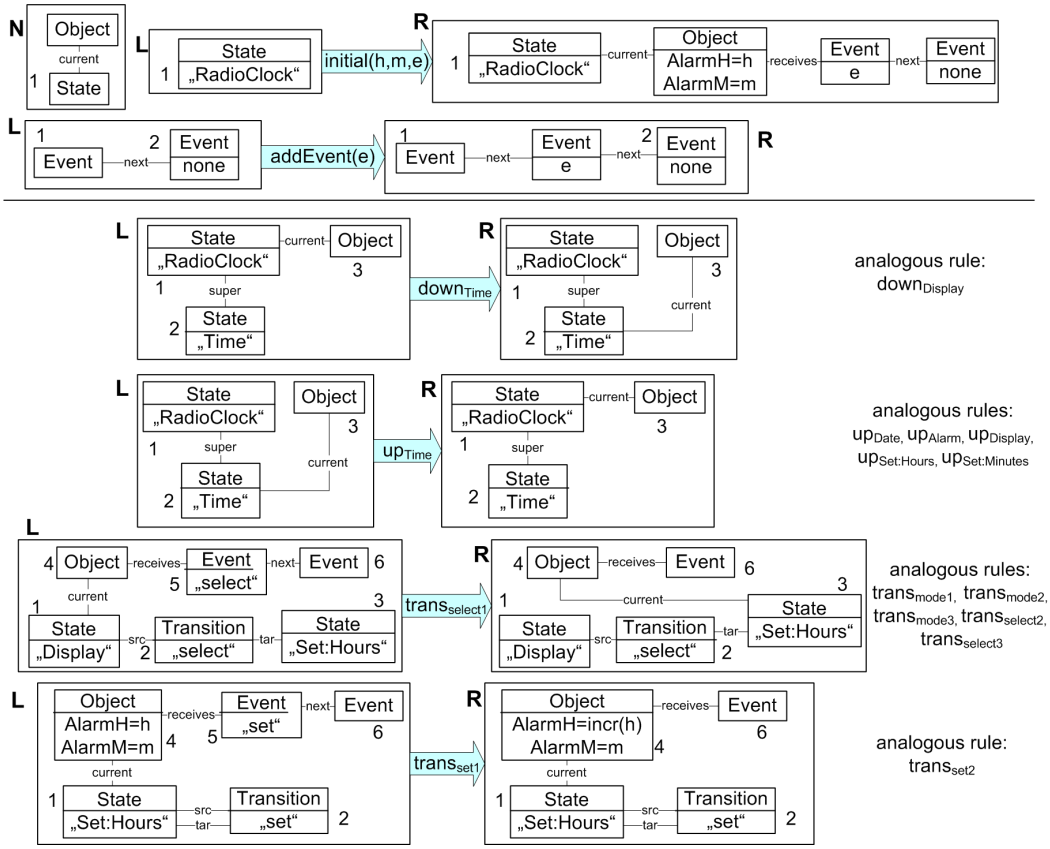


Figure 4.22: Simulation Rules for the Radio Clock

state "Radio Clock" and fill its event queue. In this way, the events that should be processed during a simulation run, can be defined in the beginning of the simulation. Alternatively, events also may be inserted at the end of the queue while a simulation is running. Further-

more, the object node holds values for the initial alarm time given by the rule parameters of rule initial. The second layer contains all remaining rules and realizes the actual simulation, processing the events in the queue. For each super-state there is a down rule which moves the current pointer from the super-state to its initial substate. For each substate there is an up rule moving the current pointer from the substate to its super-state. For each transition there is a trans rule moving the current pointer from the source state of the transition to its target state, if the next event in the queue is the triggering event of the transition. For the transitions named "set", the value of hours or the minutes of the current alarm time are incremented by the respective rule.

The simulation specification $\mathbf{SimSpec_{VL_S}} = (VL_S, P_S)$ consists of the simulation language VL_S typed over TG_S, where TG_S is the simulation alphabet depicted in the right-hand side of Fig. 4.23, P_S is the set of simulation rules shown in Fig. 4.22, and VL_S consists of all graphs that can occur in any Radio Clock simulation scenario: $VL_S = \{G_S | \exists G_I \overset{P_S*}{\Longrightarrow} G_S\}$.

Animation Alphabet

Fig. 4.23 shows the animation alphabet TG_A, which is a disjoint union of the simulation alphabet TG_S and a new visualization alphabet TG_V which is shown in the right part of Fig. 4.23, and which models the elements for a domain-specific visualization of the radio clock behavior. (Since we do not need the concrete syntax of the Statecharts to define the animation view, we do not depict it in Fig. 4.23.)

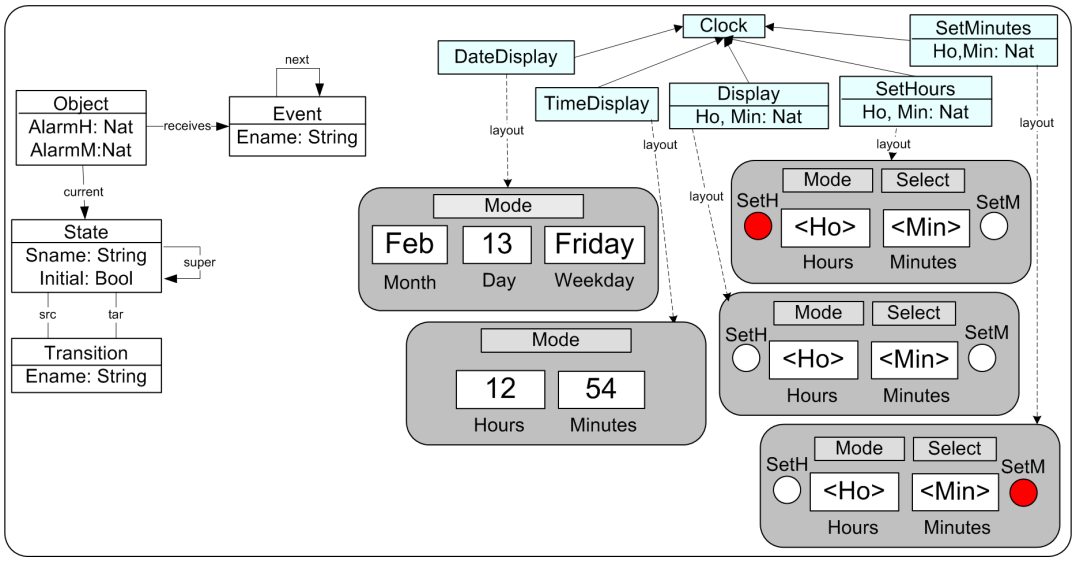


Figure 4.23: Animation Alphabet for the Radio Clock

The symbol Clock is the root symbol of the animation, where all other symbols are linked

to. Hence, the Clock symbol type belongs to type layer 1. The three modes of the clock are visualized by five different displays: a date display, showing the date (month / day / weekday), a time display showing the time (hours / minutes), and three alarm displays showing the time for the alarm to ring, but differing in the states of two lights which indicate the states Display (both lights off), Set:Hours (light SetH on), and Set:Minutes (light SetM on).

State changes between the displays are realized by pressing the Mode or the Select button, which are also visualized in all respective displays.

The Radio Clock Animation Alphabet is a valid animation alphabet according to Def. 4.2.4, where the type clock has type layer 1, and the remaining types belong to type layer 2.

$S2A$ Transformation

The $S2A$ transformation rules Q, shown in Fig. 4.24, add corresponding visualization elements to the simulation rules and to the initial radio clock graph, depending on the state the current pointer is pointing at. We visualize only basic states which do not have any substates. Super-states (i.e. the states Radio Clock and Alarm are not visualized in the animation view, as they are considered as transient, abstract states which are active on the way of the current pointer up and down the state hierarchy between two basic states, but which have no concrete layout themselves.

$S2A$ rule clock initializes the animation view part by adding a Clock symbol to all (rule) graphs it is applied to. This rule belongs to $S2A$ rule layer 0. To this Clock symbol, all animation elements will be linked. $S2A$ rules time and date generate the time display or the date display for the corresponding active state named "Time" or "Date", respectively. $S2A$ rules display, setH and setM generate the different alarm displays, where the numbers of hours and minutes to be shown in the respective display positions are the current values of the corresponding Object attributes.

All Radio Clock $S2A$ transformation rules are typed over TG_A, each $S2A$ transformation rule has a NAC which equals its RHS. Moreover, all rules within the same rule layer are parallel independent, as none of them generates elements which are forbidden by the NACs of the other rules in the layer. Hence, together with their rule and type layers, the $S2A$ rules comprise a valid, type-increasing $S2A$ transformation system according to Def. 4.2.8.

The Radio Clock $S2AM$ transformation $S2AM : VL_S \to VL_A$ is given by $S2AM = (VL_S, TG_A, Q)$, where the animation language $VL_A = \{G_A | \exists G_S \in VL_S : G_S \overset{Q}{\Longrightarrow}{}^{*} G_A\}$.

We consider a sample $S2A$ transformation sequence which transforms the simulation rule up_{Time}. In the first $S2A$ transformation step, only the $S2A$ rule clock is applicable. It is applied according to Case (1) of Def. 4.2.12 to all three rule graphs of rule up_{Time}, which results in an intermediate rule up'_{Time}. Secondly, rule time from $S2A$ rule layer 2

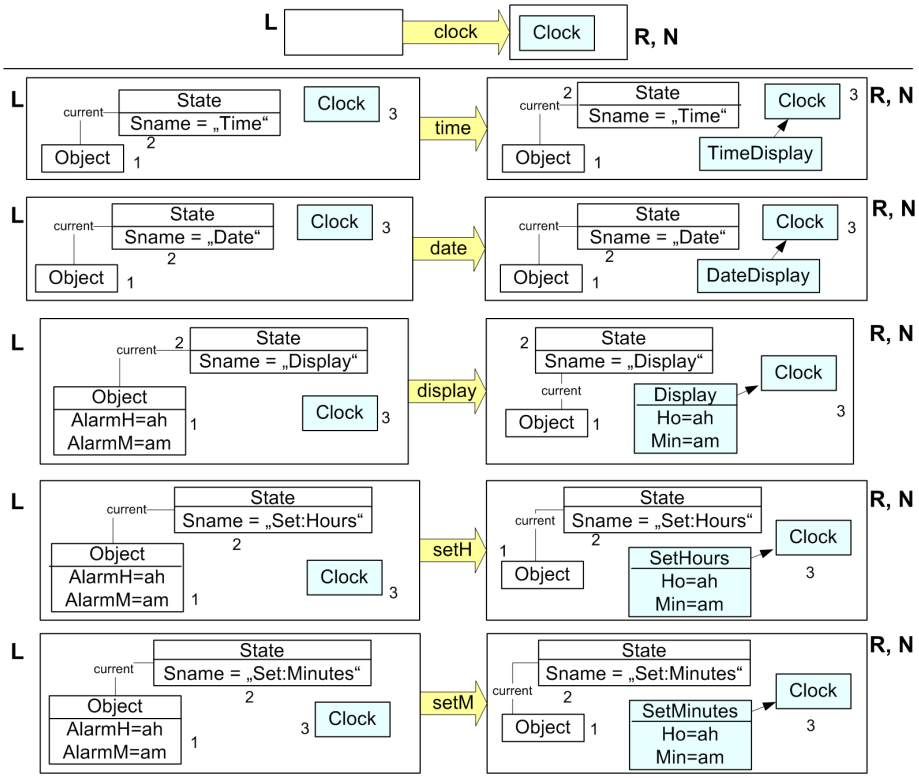


Figure 4.24: $S2A$ Rules for the Radio Clock

is applicable to rule up'_{Time}, according to Case (2), as there is a match to the LHS of rule up'_{Time}. The application adds a symbol of type Display to the LHS graph, and linking the TimeDisplay symbol to the Clock symbol. This transformation step is depicted in Fig. 4.25, resulting in the animation rule $S2A(up_{Time}) = (L'' \leftarrow I'' \rightarrow R'')$, since no more $S2A$ rules can be applied to this rule.

The Radio Clock $S2AR$ transformation $S2AR : P_S \rightarrow P_A$ is given by $S2AR = (P_S, TG_A, Q)$, where the animation rules $P_A = \{p_A | \exists p_S \in P_S : p_S \overset{Q}{\Longrightarrow}{}^{!} p_A \}$.

The Radio Clock animation specification $\mathbf{AnimSpec_{VL_A}} = S2A(\mathbf{SimSpec_{VL_S}})$ based on the $S2A$ transformation $S2A = (S2AM, S2AR)$ is given by $\mathbf{AnimSpec_{VL_A}} = (VL_A, P_A)$, where VL_A is the animation language obtained by the Radio Clock $S2AM$ transformation, and P_A are the animation rules obtained by the Radio Clock $S2AR$ transformation, some of which are shown in Fig. 4.26.

The start state of the sample animation scenario shown in Fig. 4.27 in the concrete notation of the animation view, was obtained from the initial state shown in Fig. 4.21 by applying the rule initial(6,30,mode) setting the alarm time to 06:30 and the first event in the queue to mode, and subsequently applying rule addEvent to fill the queue with the events mode, select, set, mode. Fig. 4.27 shows the subsequent simulation steps by applying animation rules from the second rule layer.

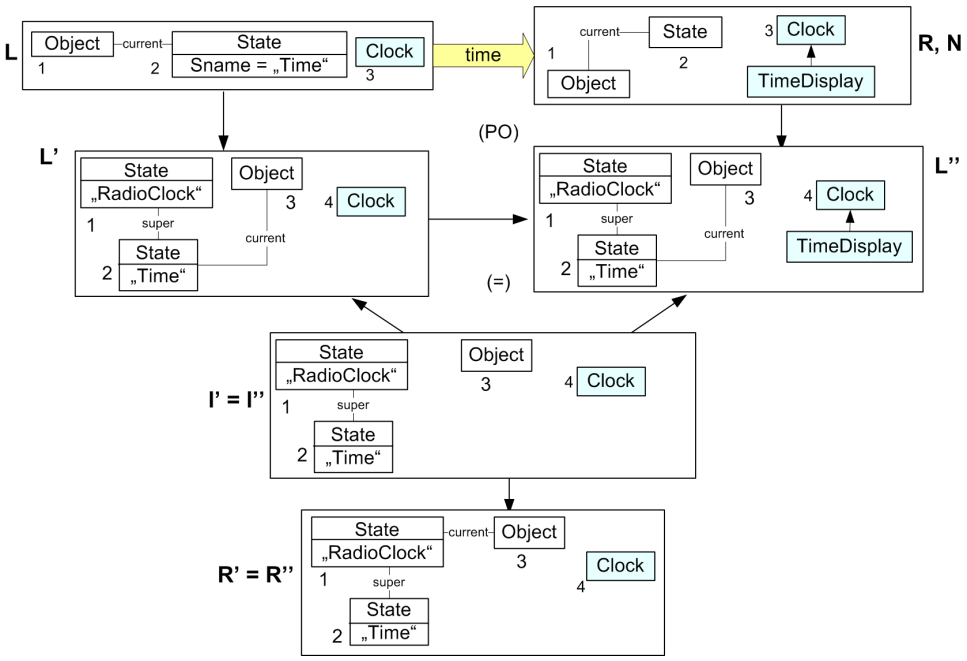


Figure 4.25: Application of $S2A$ Rule time to Simulation Rule up'_{Time}

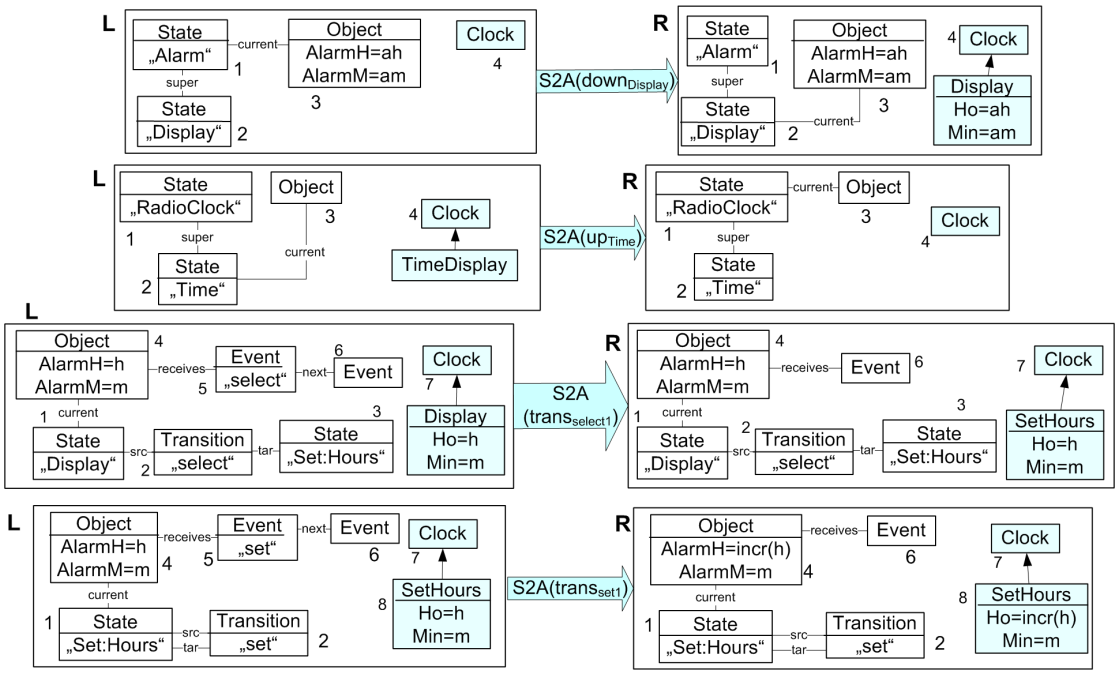


Figure 4.26: Animation Rules for the Radio Clock

Correctness and Completeness of the $S2A$ Transformation

To ensure the semantical correctness of the Radio Clock $S2A$ transformation, we have to check the NAC-compatibility. of the $S2AM$ transformation.

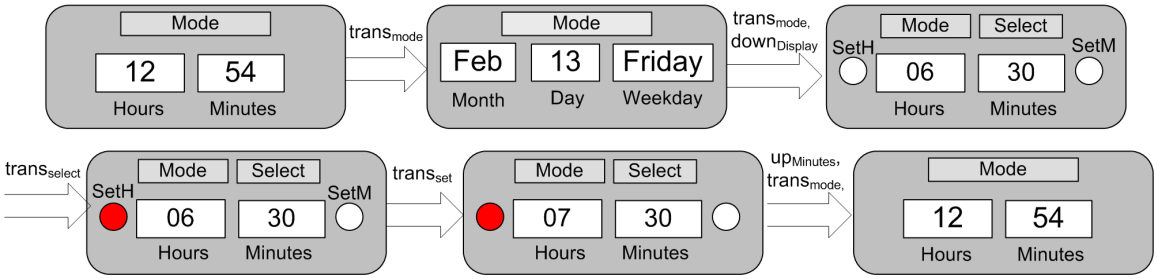


Figure 4.27: Animation Scenario of the Radio Clock Model

Fact 4.7.5 (*NAC-Compatibility of the Radio Clock $S2AM$ Transformation*)
The Radio Clock $S2AM$ transformation is NAC-compatible (see Def. 4.5.2) in the following sense: For all $p_i \overset{q}{\Longrightarrow} p_{i+1}$ with $q = (L_q \overset{q}{\longrightarrow} R_q)$ and $NAC_q = (L_q \overset{q}{\longrightarrow} R_q)$ such that the match from q to p_i satisfies NAC_q, the following $S2AM$ steps also satisfy NAC_q according to the rule transformation cases below:

Case (1): $G_i \overset{q}{\Longrightarrow} G_{i+1}$ and $H_i \overset{q}{\Longrightarrow} H_{i+1}$,
Case (2): $G_i \overset{q}{\Longrightarrow} G_{i+1}$,
Case (3): $H_i \overset{q}{\Longrightarrow} H_{i+1}$. △

Proof: We show for all $q \in Q$ that for a match $L_q \to X$ there is no NAC-morphism $(R_q - L_q) \overset{x}{\longrightarrow} X$. Due the property of all $q \in Q$ being type-increasing, in this case NAC_q is satisfied for this match.

The only $S2A$ rule which can be applied to any rule p_i according to Case (1) is rule clock. As rule clock belongs to rule layer 1, all rules p_i it can be applied to, are the original simulation rules, and do not contain symbols typed over TG_V. Hence, a step involving the application of clock to a rule p_i is always NAC-compatible, since

- the match $L_q \overset{h}{\longrightarrow} I_{p_i} \overset{l_{p_i}}{\longrightarrow} L_{p_i} \overset{m_{p_i}}{\longrightarrow} G_i$ satisfies NAC_q as G_i does not contain TG_V-typed elements, and hence there is no NAC-morphism $(R_q - L_q) \overset{x}{\longrightarrow} G_i$;

- the match $L_q \overset{h}{\longrightarrow} I_{p_i} \overset{r_{p_i}}{\longrightarrow} R_{p_i} \overset{m^*_{p_i}}{\longrightarrow} H_i$ satisfies NAC_q as H_i does not contain TG_V-typed elements, and hence there is no NAC-morphism $(R_q - L_q) \overset{x}{\longrightarrow} H_i$;

All subsequent $S2A$ transformation steps are either according to Case (2) or to Case (3). Note that the right-hand sides of all $S2A$ rules do not overlap in their generated elements, i.e. in $(R_q - L_q)$.

Let us consider a step according to Case (2), first: We assume that q is applicable to p_i, i.e. there is a match $L_q \overset{h}{\longrightarrow} L_i$ satisfying NAC_q. Now, if NAC_q is not satisfied for the match $L_q \overset{h}{\longrightarrow} L_i \overset{m}{\longrightarrow} G_i$, then this means that q must have been applied before to another rule p_j according to Case (2) with $p_j \overset{q}{\Longrightarrow} p_{j+1} \Longrightarrow \cdot\cdot \Longrightarrow p_i$ with $j < i$, since q is the

only $S2A$ rule which could add the elements $(R_q - L_q)$ to G_i. But in this case, we have a NAC-morphism $(R_q - L_q) \to L_{j+1} \to L_i$ which is a contradiction to our assumption that NAC_q is satisfied for the match $L_q \to L_i$. Hence, NAC_q must be satisfied for the match $L_q \xrightarrow{h} L_i \xrightarrow{m} G_i$.

Analogously, we can argue for the Case (3) steps: We assume that q is applicable to p_i, i.e. there is a match $L_q \xrightarrow{h} R_i$ satisfying NAC_q. Now, if NAC_q is not satisfied for the match $L_q \xrightarrow{h} R_i \xrightarrow{m^*} H_i$, then this means that q must have been applied before to another rule p_j according to Case (3) with $p_j \overset{q}{\Longrightarrow} p_{j+1} \Longrightarrow \cdot\cdot \Longrightarrow p_i$ with $j < i$, since q is the only $S2A$ rule which could add the elements $(R_q - L_q)$ to H_i. But in this case, we have a NAC-morphism $(R_q - L_q) \to L_{j+1} \to L_i$ which is a contradiction to our assumption that NAC_q is satisfied for the match $L_q \to R_i$. Hence, NAC_q must be satisfied for the match $L_q \xrightarrow{h} R_i \xrightarrow{m^*} H_i$. □

Fact 4.7.6 (*Rule Compatibility of Radio Clock $S2A$ Transformation*)
The Radio Clock $S2A$ transformation is rule compatible in the sense of Def. 4.5.9, i.e. all p_A and all q are parallel and sequential independent. △

Proof: If p_A is applicable to a graph G, then there is a match $L_A \xrightarrow{m} G$. Therefore, symbols of at least those types from TG_V that are contained in L_A have also to be contained in G. So, in the sequence $G_S \overset{Q*}{\Longrightarrow} G$ there have been applied at least those rules $q \in Q$ which have also been applied in $\overset{Q\ !}{\Longrightarrow} p_S p_A$ according to Case (1) or (2) (i.e. applied to some L_i, $i = 0, .., n$). All those rules q are not applicable anymore to L_A because of their NACs NAC_q. Neither are they applicable to G, due to NAC-compatibility.

So we have to consider only those overlappings L_A/L_q where $h(L_q)$ is not completely included in $m(L_A)$. As the LHS of $S2A$ rule clock is empty, we do not have to consider this rule at all. Moreover, there exists exactly one instance of type Clock in each graph G, in all L_q and in all L_A. Hence, all pairs L_A/L_q overlap in the Clock symbol. This is uncritical, as the Clock symbol is always preserved by all rules. Furthermore, there exists always only one State symbols with a certain name. So all pairs L_A/L_q which both contain a State symbol with the same name, overlap in this node. Again, this is uncritical, as State symbols are always preserved by all rules. The next symbol and link L_A/L_q could overlap at, is a symbol of type Object and a link of type current. The Object symbol is the last node apart from the Clock and State nodes in the LHSs of the $S2A$ transformation rules. As we have argued above, L_A and L_q overlap already in the Clock and State nodes. If they would overlap also in the Object node and the current link, they would overlap completely, i.e. $h(L_q)$ would be included in $m(L_A)$. In this case q would not be applicable as shown above. As there is exactly one Object node and one current link in any graph G, we can conclude that there are no pairs L_A/L_q which do not overlap completely, and in these cases NAC_q forbids the application of q.

Hence, all pairs p_A/q are parallel independent.

Due to the Local Church Rosser Theorem 2.1.19, we know that if $G \overset{p_A}{\Longrightarrow} H$ and $G \overset{q}{\Longrightarrow} G'$ are parallel independent (which was shown above), then $G \overset{p_A}{\Longrightarrow} H$ and $H \overset{q}{\Longrightarrow} H'$ are sequential independent. □

Fact 4.7.7 (*Confluence of Radio Clock $S2A$ Transformation*)
The $S2A$ transformation $S2A = (S2AM, S2AR)$ based on the $S2A$ transformation system (TG_A, Q) for the Radio Clock model, where the $S2A$ transformation rules Q are shown in Fig. 4.24, is confluent. △

Proof: The conflict analysis for the Radio Clock $S2A$ transformation rules yields one potential conflict for each pair q/q. All potential conflicts are produce-forbid-conflicts, i.e. the NAC of the second rule forbids its second application after its first application. Consider the sample critical pair in Fig. 4.28 found for application of rule time and again rule time.

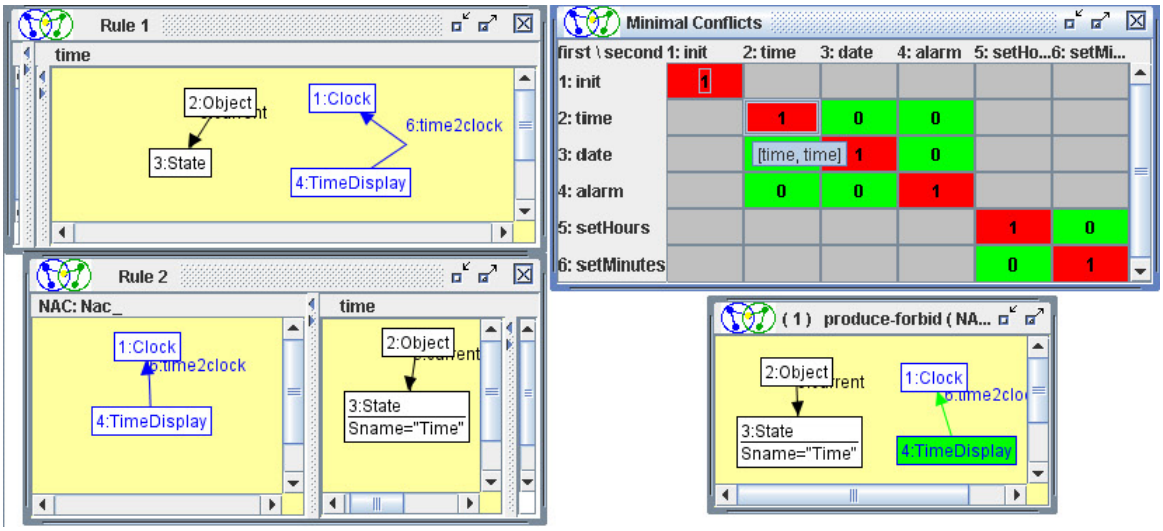


Figure 4.28: Critical Pairs of the Radio Clock $S2AM$ Transformation

The only critical pair (1) shows that the co-match from the first RHS and the NAC-morphism from the second rule overlap completely. This situation is intended because we *do not want* the rule to be applied again at the same essential match. Hence, Critical Pair (1) does not lead to an unwanted conflict. Analogously, the critical pairs found for each of the other rule pairs q/q only model the intended conflict of applying the rule again at the same essential match, but do not yield unwanted conflicts.

Hence, $S2AM$ transformation is confluent.

Regarding $S2AR$ transformation, we have to check what happens if two $S2A$ transformation rules q_1 and q_2 from the same rule layer are applied to the same rule p_i with $i = 0, .., n$

and $p_S = p_0, p_n = p_A$ in $p_S \overset{Q\ !}{\Longrightarrow} p_A$. If they are applied according to the same rule transformation case (see Def. 4.2.12), i.e. they change the same rule graph(s), then we have a conflict which can always be resolved, since q_1 and q_2 are parallel independent by Definition 4.2.8. If they are applicable according to different rule transformation cases, then the match priorities define an application order, such that we will not get conflicts. □

Fact 4.7.8 (*Semantical Equivalence of $SimSpec_{VL_S}$ and $AnimSpec_{VL_A}$*)
The $S2A$ transformation $S2A = (S2AM, S2AR)$ based on the $S2A$ transformation system (TG_A, Q) for the Radio Clock model, where the $S2A$ transformation rules Q are shown in Fig. 4.24, is a semantical equivalence of $SimSpec_{VL_S}$ and $AnimSpec_{VL_A}$. △

Proof: Termination has been shown to be fulfilled for general type-increasing $S2A$ transformation systems in Theorems 4.3.5 and 4.3.6. Since the Radio Clock $S2A$ transformation is NAC-compatible due to Fact 4.7.5, it is hence also semantically correct due to Theorem 4.5.11. Furthermore, due to Theorem 4.6.5, we have a backward transformation $A2S : \mathbf{AnimSpec_{VL_A}} \rightarrow \mathbf{SimSpec_{VL_S}}$, defined by type restrictions, such that the $S2A$ transformation is also semantically complete. In addition to semantical completeness, according to Theorem 4.6.15, the requirements for an $S2A$ transformation being a semantical equivalence, are rule compatibility and confluence. These requirements have been shown for the Radio Clock $S2A$ transformation in Fact 4.7.6 (rule compatibility) and Fact 4.7.7 (confluence). □

4.7.3 Animation View of the UML Model of a Drive-Through

In this section we develop an animation view for the Drive-Through model introduced in Chapter 3.5.3.

Simulation Specification

Fig. 4.29 (a) shows the type graph TG_S and Fig. 4.29 (b) shows the initial instance graph for our Drive-Through simulation, which is typed over TG_S. The type graph consists of the four symbol types Client, DriveThrough, Meal, Offer and Order. Clients may be associated with a drive-through via a visit-link (in this case they are hungry). Every client in a queue is linked with his/her successor provided that he/she is not the last person in a queue. This is reflected in the next link. The first and the last client are also linked to the drive-through they are visiting. Clients may submit orders of meals which are offered by the drive-through, and eat meals that are served by drive-throughs. The names of the meals and the orders are given by attributes. After eating, clients may leave the drive-through.

The initial instance graph for the simulation consists of four clients and one drive-through restaurant together with its two offered meals. Three clients are already queueing at the drive-through, one is lingering nearby. Note that in this example we only have the abstract syntax of the model, which is the basis of the automatic translation from the integrated UML model into the graph transformation system.

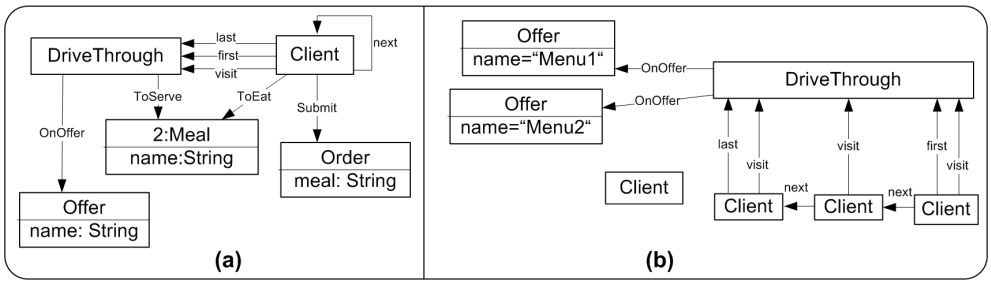


Figure 4.29: Type Graph TG_S (a) and Initial State G_I (b) of a Drive-Through Simulation

The model-specific simulation rules for the Drive-Through model are shown in Fig. 4.30.

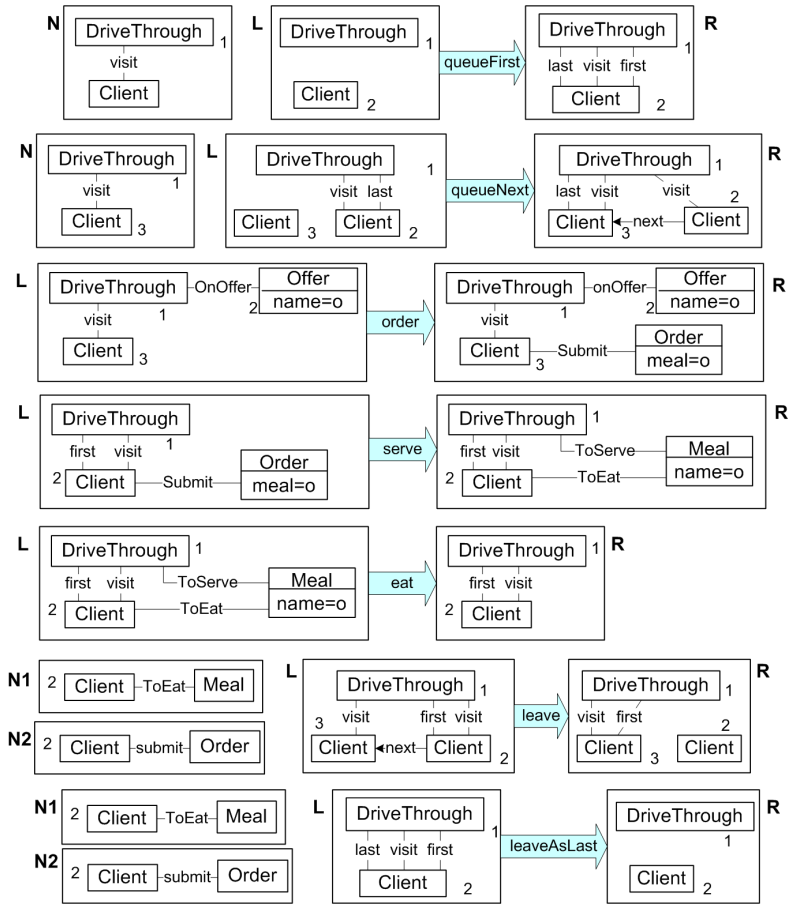


Figure 4.30: Simulation Rules for the Drive-Through Model

The simulation specification $SimSpec_{VL_S} = (VL_S, P_S)$ consists of the simulation language VL_S typed over TG_S, where TG_S is the simulation alphabet depicted in the Fig. 4.29 (a), P_S is the set of simulation rules shown in Fig. 4.30, and VL_S consists of all graphs that can occur in any Drive-Through simulation scenario: $VL_S = \{G_S | \exists G_I \overset{P_S*}{\Longrightarrow} G_S\}$, where G_I is the discrete graph containing one DriveThrough node, and a certain number of Client nodes.

Animation Alphabet

Fig. 4.31 shows the animation alphabet TG_A, which is a disjoint union of the simulation alphabet TG_S and the new visualization alphabet TG_V which is shown in the right part of Fig. 4.23, and which models the elements for a domain-specific visualization of the drive-through behavior. We choose an animation view where the clients are visualized as cars queueing in front of a restaurant building. Their orders are shown as bubbles inscribed by the order attribute meal, e.g. "Menu 1", which are positioned above the car submitting the order. Similarly, served meals are visualized as a burger and a soft-drink, which are positioned besides the car they are served to. Animated actions are the entering of the drive-through, ordering, being served, eating and leaving the queue.

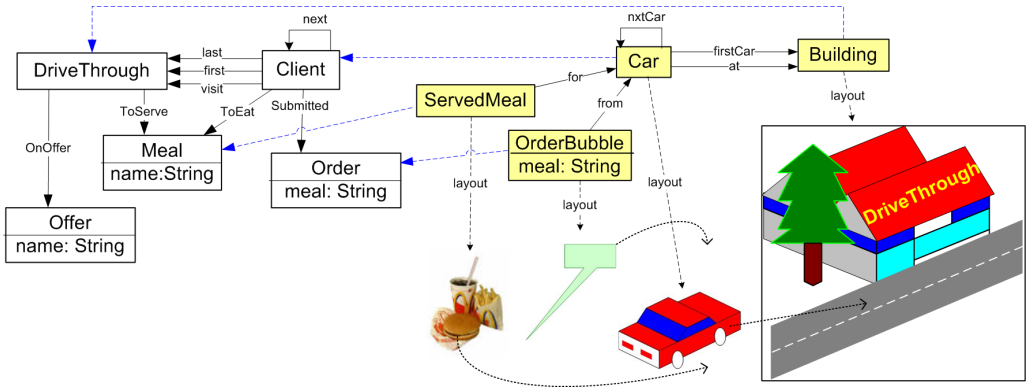


Figure 4.31: Animation Alphabet for the Drive-Through

The symbol Building is the root symbol of the animation, where all other symbols are linked to. Hence, the Building symbol type belongs to type layer 1. Cars are linked to the Building symbol, and hence belong to type layer 2, whereas ServedMeal and OrderBubble symbols are linked to a Car symbol and belong to layer 3. Note that we have reference arcs connecting animation view symbols to their counterparts in the simulation alphabet. These arcs are to control that the correct number of symbols of a certain type is generated in the $S2A$ transformation. Note that the target nodes of the reference arcs all belong to the interface type graph TG_I, and are therefore formally included in the animation view.

The Drive-Through Animation Alphabet is a valid animation alphabet according to Def. 4.2.4.

$S2A$ Transformation

The $S2A$ transformation rules Q, shown in Fig. 4.24, add corresponding visualization elements to the simulation rules and to an initial graph. The visualization concentrates on the cars in the queue, their orders and the meals the get served. We do not visualize the offers of the drive-through, or clients which are not linked to the drive-through.

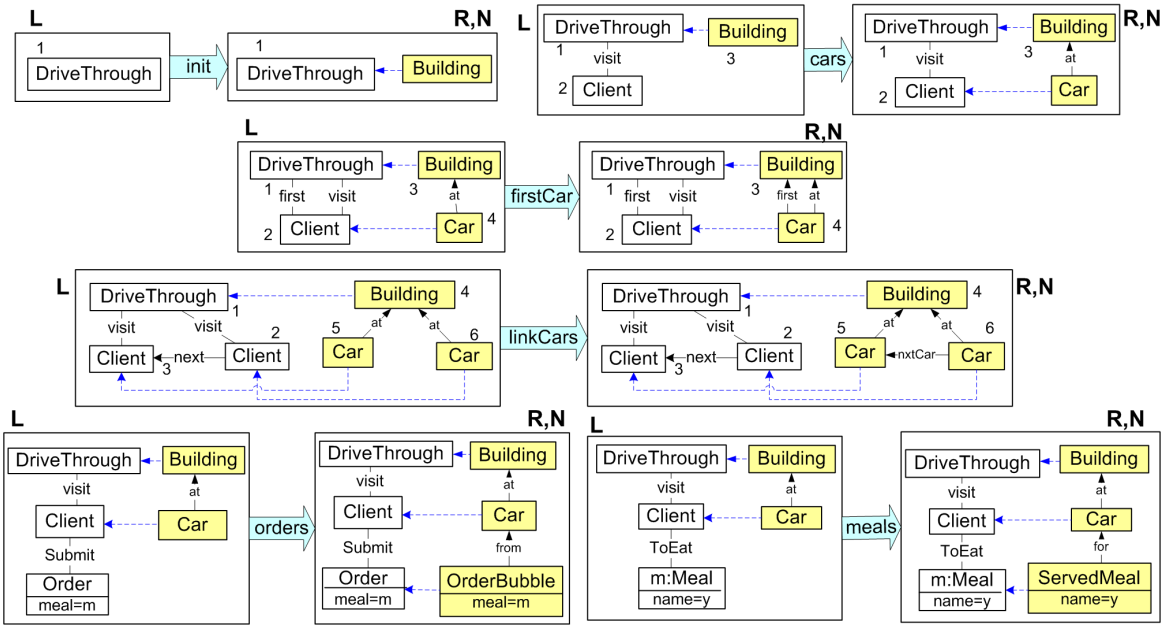


Figure 4.32: $S2A$ Rules for the Drive-Through

The $S2A$ rule init initializes the animation view part by adding a Building symbol to all (rule) graphs it is applied to. This rule belongs to $S2A$ rule layer 0. To this Building symbol, all animation elements will be linked. $S2A$ rules firstCar and nextCars generate a car symbol for each client in the queue. To the firstCar link of the first car, a graphical constraint is associated ensuring that this car is positioned at the rightmost end of the street in front of the drive-through building. The nxtCar link carries a graphical constraint positioning the target car left to the source car, with a small distance between. $S2A$ rules orders and meals generate the OrderBubble and the ServedMeal symbols for each Order and each Meal. Graphical constraints ensure that the new symbols are placed correctly depending of the position of the corresponding Car symbol.

The Drive-Through $S2AM$ transformation $S2AM : VL_S \rightarrow VL_A$ is given by $S2AM = (VL_S, TG_A, Q)$, where the animation language $VL_A = \{G_A | \exists G_S \in VL_S : G_S \overset{Q}{\Longrightarrow}{}^{*} G_A\}$.

We consider a sample $S2AR$ transformation sequence which transforms the simulation rule order to the animation rule order_A In the first $S2A$ transformation step, only the $S2A$ rule init is applicable. It is applied according to Case (1) of Def. 4.2.12 and adds a Building symbol to all three rule graphs of rule order, which results in an intermediate rule order_1.

Secondly, rule car from $S2A$ rule layer 2 is applicable to rule order$_1$, again according to Case (1). The application adds a symbol of type Car to all rule graphs, and links it to the Building symbol by a visit link, resulting in rule order$_2$. At last, the only $S2A$ rule applicable to rule order$_2$ is the $S2A$ rule orders, which is applicable according to Case (3) and adds an OrderBubble symbol in the RHS of rule order$_2$, resulting in rule order$_3$ = order$_A$. This last transformation step is depicted in Fig. 4.33.

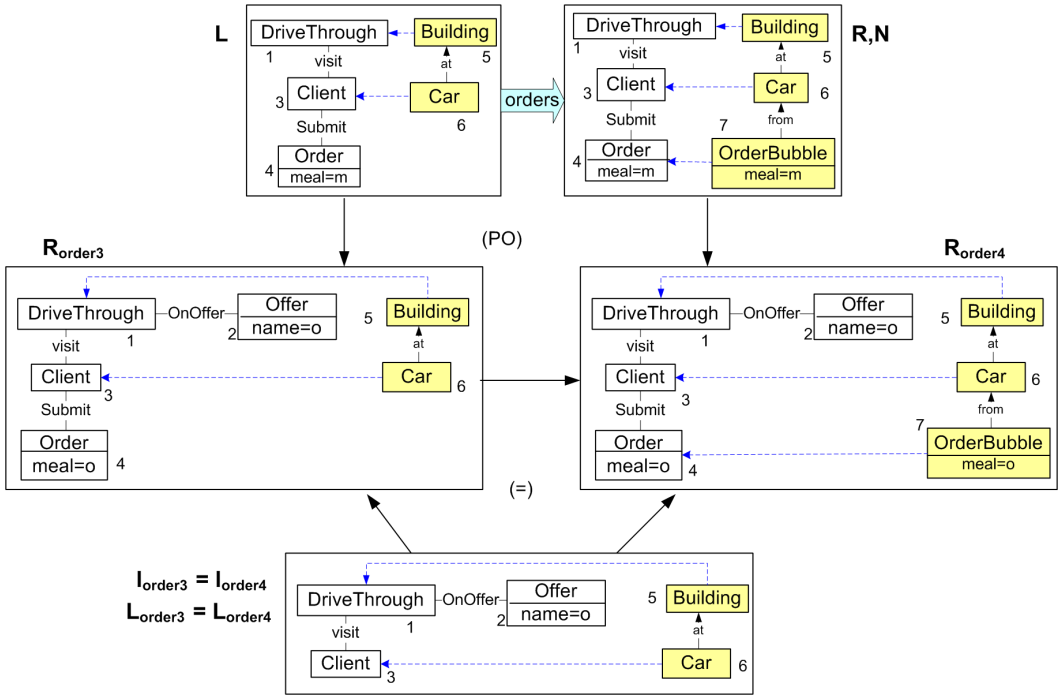


Figure 4.33: Application of $S2A$ Rule orders to Rule order$_3$

The Drive-Through $S2AR$ transformation $S2AR : P_S \rightarrow P_A$ is given by $S2AR = (P_S, TG_A, Q)$, where the animation rules $P_A = \{p_A | \exists p_S \in P_S : p_S \overset{Q\ !}{\Longrightarrow} p_A\}$.

The Drive-Through animation specification $AnimSpec_{VL_A} = S2A(SimSpec_{VL_S})$ based on the $S2A$ transformation $S2A = (S2AM, S2AR)$ is given by $AnimSpec_{VL_A} = (VL_A, P_A)$, where VL_A is the animation language obtained by the Drive-Through $S2AM$ transformation, and P_A are the animation rules obtained by the Drive-Through $S2AR$ transformation, some of which are shown in Fig. 4.34.

The start state of the sample animation scenario shown in Fig. 4.35 in the concrete notation of the animation view, was obtained from the initial state (one DriveThrough symbol and four Client symbols) by applying the simulation rules queueFirst, queueNext, queueNext, addOffer("Menu1"), addOffer("Menu2") adding two menus to the offer list of the drive-through, and putting three cars in its queue. Fig. 4.35 shows some subsequent animation steps in their concrete notation by applying animation rules.

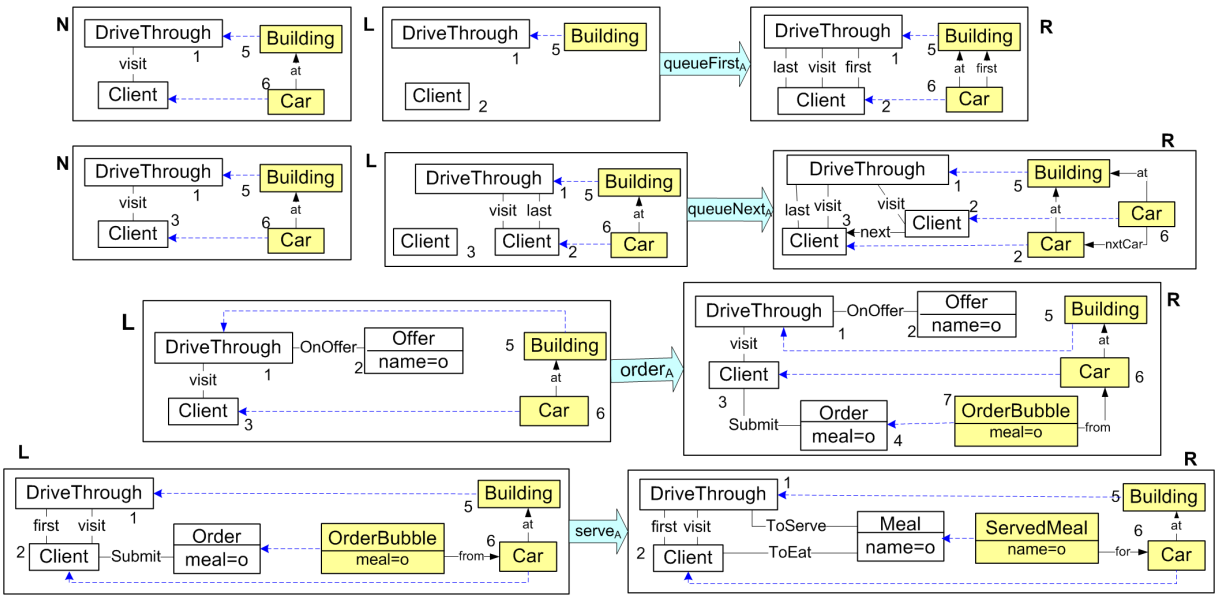


Figure 4.34: Animation Rules for the Drive-Through

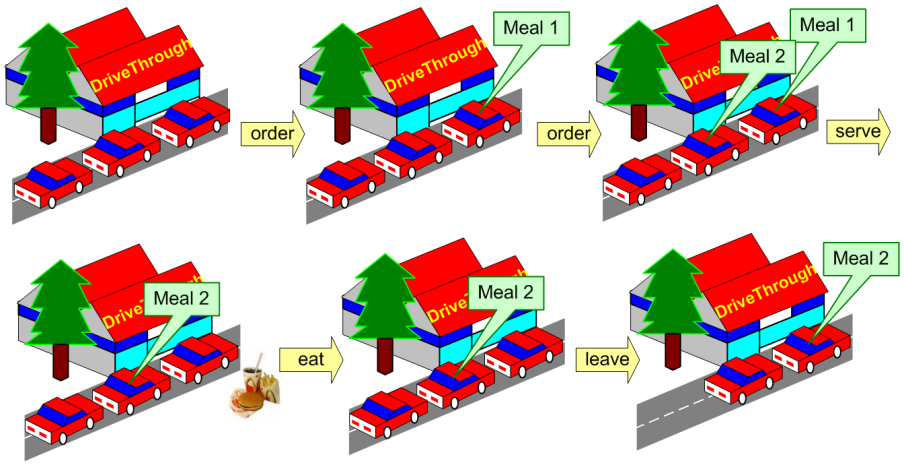


Figure 4.35: Animation Scenario of the Drive-Through Model

Correctness and Completeness of the $S2A$ Transformation

To ensure the semantical correctness of the Drive-Through $S2A$ transformation, we have to check the NAC-compatibility.

Fact 4.7.9 (*NAC-Compatibility of the Drive-Through $S2AM$ Transformation*) The Drive-Through $S2AM$ transformation is NAC-compatible (see Def. 4.5.2) in the following sense: For all $p_i \overset{q}{\Longrightarrow} p_{i+1}$ with $q = (L_q \overset{q}{\longrightarrow} R_q)$ and $NAC_q = (L_q \overset{q}{\longrightarrow} R_q)$ such that the match from q to p_i satisfies NAC_q, the following $S2AM$ steps also satisfy NAC_q according to the rule transformation cases below:

Case (1): $G_i \overset{q}{\Longrightarrow} G_{i+1}$ and $H_i \overset{q}{\Longrightarrow} H_{i+1}$,
Case (2): $G_i \overset{q}{\Longrightarrow} G_{i+1}$,
Case (3): $H_i \overset{q}{\Longrightarrow} H_{i+1}$. △

Proof: We show for all $q \in Q$ that for a match $L_q \to X$ there is no NAC-morphism $(R_q - L_q) \overset{x}{\longrightarrow} X$. Due the property of all $q \in Q$ being type-increasing, in this case NAC_q is satisfied for this match.

The only $S2A$ rules which can be applied to any rule p_i according to Case (1) is rule init. As rule init belongs to rule layer 1, all rules p_i it can be applied to, are the original simulation rules, and do not contain symbols typed over TG_V. Hence, a step involving the application of init to a rule p_S is always NAC-compatible, since

- the match $L_q \overset{h}{\longrightarrow} I_{p_i} \overset{l_{p_i}}{\longrightarrow} L_{p_i} \overset{m_{p_i}}{\longrightarrow} G_i$ satisfies NAC_q as G_i does not contain TG_V-typed elements, and hence there is no NAC-morphism $(R_q - L_q) \overset{x}{\longrightarrow} G_i$;

- the match $L_q \overset{h}{\longrightarrow} I_{p_i} \overset{r_{p_i}}{\longrightarrow} R_{p_i} \overset{m^*_{p_i}}{\longrightarrow} H_i$ satisfies NAC_q as H_i does not contain TG_V-typed elements, and hence there is no NAC-morphism $(R_q - L_q) \overset{x}{\longrightarrow} H_i$;

Let us next consider steps applying an $S2A$ rule q of a higher level according to Case (1) or (2): Since q is applicable to p_i, there is a match $L_q \overset{h}{\longrightarrow} L_i$ satisfying NAC_q. Now, if NAC_q is not satisfied for the match $L_q \overset{h}{\longrightarrow} L_i \overset{m}{\longrightarrow} G_i$, then this means that q must have been applied before to another rule p_j according to Case (2) with $p_j \overset{q}{\Longrightarrow} p_{j+1} \Longrightarrow \cdot\cdot \Longrightarrow p_i$ with $j < i$, since q is the only $S2A$ rule which could add the elements $(R_q - L_q)$ to G_i. But in this case, we have a NAC-morphism $(R_q - L_q) \to L_{j+1} \to L_i$ which is a contradiction to our assumption that NAC_q is satisfied for the match $L_q \to L_i$. Hence, NAC_q must be satisfied for the match $L_q \overset{h}{\longrightarrow} L_i \overset{m}{\longrightarrow} G_i$. In Case (1) we can argue analogously that the match $L_q \overset{h}{\longrightarrow} R_i \overset{m^*}{\longrightarrow} H_i$ also satisfies NAC_q.

Let us consider Case (3) steps next: Since q is applicable to p_i, there is a match $L_q \overset{h}{\longrightarrow} R_i$ satisfying NAC_q. Now, if NAC_q is not satisfied for the match $L_q \overset{h}{\longrightarrow} R_i \overset{m^*}{\longrightarrow} H_i$, then this means that q must have been applied before to another rule p_j according to Case (3) with $p_j \overset{q}{\Longrightarrow} p_{j+1} \Longrightarrow \cdot\cdot \Longrightarrow p_i$ with $j < i$, since q is the only $S2A$ rule which could add the elements $(R_q - L_q)$ to H_i. But in this case, we have a NAC-morphism $(R_q - L_q) \to R_{j+1} \to R_i$ which is a contradiction to our assumption that NAC_q is satisfied for the match $L_q \to R_i$. Hence, NAC_q must be satisfied for the match $L_q \overset{h}{\longrightarrow} R_i \overset{m^*}{\longrightarrow} H_i$. □

Fact 4.7.10 (*Rule Compatibility of Drive-Through $S2A$ Transformation*)
The Drive-Through $S2A$ transformation is rule compatible in the sense of Def. 4.5.9, i.e. all p_A and all q are parallel and sequential independent. △

Proof: If p_A is applicable to a graph G, then there is a match $L_A \xrightarrow{m} G$. Therefore, symbols of at least those types from TG_V that are contained in L_A have also to be contained in G. So, in the sequence $G_S \overset{Q*}{\Longrightarrow} G$ there have been applied at least those rules $q \in Q$ which have also been applied in $p_S \overset{Q\ !}{\Longrightarrow} p_A$ according to Case (1) or (2) (i.e. applied to some L_i, $i = 0, .., n$). All those rules q are not applicable anymore to L_A because of their NACs. Neither are they applicable to G at match $L_q \xrightarrow{h} L_A \xrightarrow{m} G$, due to NAC-compatibility.

So, only those pairs q/p_A are both applicable to a graph G where the image of $L_q \xrightarrow{m_q} G$ is not completely included in $m(L_A)$. The LHS $L_{q_{init}}$ of $S2A$ rule init contains only a DriveThrough symbol. A DriveThrough symbol occurs exactly once in each L_A, and also in G. Hence, for all pairs $L_{q_{init}}/L_A$ we have $m_q(L_q) \subseteq m(L_A)$, which means that due to NAC_{q_init}, rule init and an animation rule are never both applicable to G.

Let us consider $S2A$ rule cars next: L_q overlaps with L_A at least in the symbols DriveThrough and Building (for both symbol types, there exists at most one instance). But they may not overlap in the Client node (if there is a Client node in G to which rule cars has not yet been applied). Yet, this overlapping is uncritical as none of the animation rules p_A delete objects needed by $S2A$ rule cars, nor does any p_A generate objects which are forbidden by NAC_q, and vice versa. Hence, $S2A$ rule cars and any animation rule p_A are parallel independent.

Analogously, we can argue that the remaining $S2A$ rules firstCar, linkCars, orders and meals are parallel independent from all animation rules, as their LHS can overlap only in items which are preserved, and since no rule p_A adds items which are forbidden by NAC_q. For the last two animation rules leave$_A$ and leaveAsLast, we also need the fact that the links of types first and last occur at most once, and that a Client who is not alone in the queue cannot be linked by both the first and the last link to the DriveThrough.

Hence, all pairs p_A/q are parallel independent.

Due to the Local Church Rosser Theorem 2.1.19, we know that if $G \overset{p_A}{\Longrightarrow} H$ and $G \overset{q}{\Longrightarrow} G'$ are parallel independent (which was shown above), then $G \overset{p_A}{\Longrightarrow} H$ and $H \overset{q}{\Longrightarrow} H'$ are sequential independent. □

Fact 4.7.11 (*Confluence of Drive-Through $S2A$ Transformation*)
The $S2A$ transformation $S2A = (S2AM, S2AR)$ based on the $S2A$ transformation system (TG_A, Q) for the Drive-Through model, where the $S2A$ transformation rules Q are shown in Fig. 4.32, is confluent. △

Proof: The conflict analysis for the Drive-Through $S2A$ transformation rules yields one potential conflict for each pair q/q. All potential conflicts are produce-forbid-conflicts, i.e. the NAC of the second rule forbids its second application after its first application.

Consider the sample critical pair in Fig. 4.36 found for application of rule linkCars and again rule linkCars.

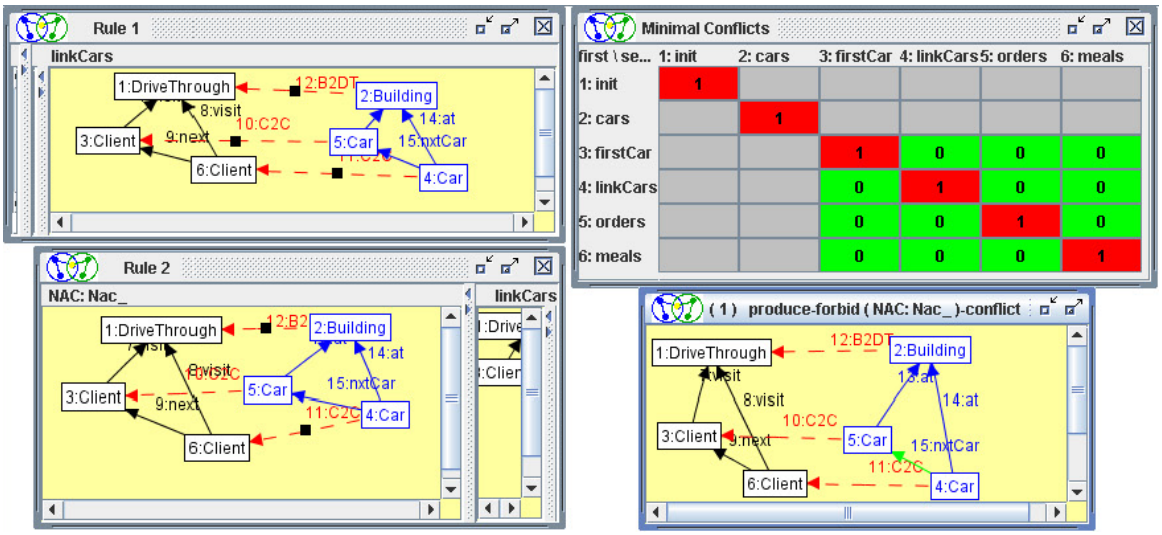


Figure 4.36: Critical Pairs of the Drive-Through $S2AM$ Transformation

The only critical pair (1) shows that the co-match from the first RHS and the NAC-morphism from the second rule overlap completely. This situation is intended because we do not want the rule to be applied again at the same essential match. Hence, Critical Pair (1) does not lead to an unwanted conflict. Analogously, the critical pairs found for each of the other rule pairs q/q only model the intended conflict of applying the rule again at the same essential match, but do not yield unwanted conflicts.

Hence, $S2AM$ transformation is confluent.

Regarding $S2AR$ transformation, we have to check what happens if two $S2A$ transformation rules q_1 and q_2 from the same rule layer are applied to the same rule p_i with $i = 0, .., n$ and $p_S = p_0, p_n = p_A$ in $p_S \overset{Q}{\Longrightarrow}{}^{!} p_A$. If they are applied according to the same rule transformation case (see Def. 4.2.12), i.e. they change the same rule graph(s), then we have a conflict which can always be resolved, since q_1 and q_2 are parallel independent by Definition 4.2.8. If they are applicable according to different rule transformation cases, then the match priorities define an application order, such that we will not get conflicts. □

Fact 4.7.12 (*Semantical Equivalence of* $SimSpec_{VL_S}$ *and* $AnimSpec_{VL_A}$)
The $S2A$ transformation $S2A = (S2AM, S2AR)$ based on the $S2A$ transformation system (TG_A, Q) for the Drive-Through model, where the $S2A$ transformation rules Q are shown in Fig. 4.32, is a semantical equivalence of $\mathbf{SimSpec_{VL_S}}$ and $\mathbf{AnimSpec_{VL_A}}$. △

Proof: Termination has been shown to be fulfilled for general type-increasing $S2A$ transformation systems in Theorems 4.3.5 and 4.3.6). Since the Drive-Through $S2A$ transfor-

mation is NAC-compatible due to Fact 4.7.9, it is hence also semantically correct due to Theorem 4.5.11. Furthermore, due to Theorem 4.6.5, we have a backward transformation $A2S : \mathbf{AnimSpec}_{\mathbf{VL_A}} \rightarrow \mathbf{SimSpec}_{\mathbf{VL_S}}$, defined by type restrictions, such that the $S2A$ transformation is also semantically complete. In addition to semantical completeness, according to Theorem 4.6.15, the requirements for an $S2A$ transformation being a semantical equivalence, are rule compatibility and confluence. These requirements have been shown for the Drive-Through $S2A$ transformation in Fact 4.7.10 (rule compatibility) and Fact 4.7.11 (confluence). □

4.8 Related Work

In the area of UML modeling, the notion of *animation* has been used for example for the following visualizations:

- In [OK99], so-called "filmstrips" are generated from the UML model (sequences of snapshots which are object diagrams over a given class diagram). This approach is closely related to our simulation specification with the difference that we use graph transformation rules instead of OCL constraints.
- Various commercial tools allow to enrich UML behavioral diagrams by graphical means for highlighting process steps, such as colored message arcs in sequence diagrams or colored activity rectangles in activity diagrams (see e.g. the animation add-in for Microsoft Visio [Sys04]).
- Gogolla et al. [RG00, GRR99] enhance the UML diagram syntax, e.g. by three-dimensional layout to make UML diagrams more readable. They use blocks for classes in class diagrams and move message balls along the object lifelines of a sequence diagram whenever a message is sent between two objects.

We prefer to regard the execution of a prototype which is generated from a behavioral model, such as our graph-transformation based simulation specification, as *simulation*, even if the syntax is enhanced by highlighting or 3D-features. Our *animation view*, the basis for animation, is defined using a layout for visualizing the animation which is completely different to the layout of the formal modeling language.

A more similar, tool-oriented approach, where different visual representations are used to visualize a model's behavior, is called *reactive animation* and has been advocated by Harel in [HEC03]. Here, the reactive system behavior is specified with tools like Rhapsody [Rha05] using UML, or the Play-Engine [HM03] using Life-Sequence-Charts, an extension of Statecharts. The animated representation of the system behavior is implemented by linking these tools to pure animation tools like Flash or Director from Macromedia

[Fla04, Dir04]. Here, the mapping from simulation to animation happens at the implementation level and is not specified formally.

Different Petri net tools also offer support for customized Petri net animations, which show, move and hide graphical objects in correlation with transition firing steps. The SimPEP tool [Gra99] allows to define VRML-specifications (Virtual Reality Modeling Language) to visualize transition firings of low-level Petri nets. For Coloured Petri nets, the tool Design/CPN offered a graphical GUI library called MIMIC [LRS95]. Instead of investigating token distributions in a coloured Petri net, the Design/CPN user could inspect the state of the simulator via graphical objects (e.g. a picture of a mobile phone). It is important to realize, however, that such GUI tools are limited, and are mainly useful for obtaining a rough idea of the highly discrete user interface of the system in operation. They are in fact almost static in nature (like the GUI of a mobile phone, a watch or a calculator) and do not supply a general means allowing to animate selected features of a model at a freely chosen level of abstraction adapted to the development stage of the system model. The MIMIC library was implemented specific to the DESIGN-CPN tool environment and was not adapted to the new toolset, CPN TOOLS.

In general, approaches to enhance the front end of CASE tools using GUIS for simulating/animating the behavior of models are restricted to the modeling language implemented by the CASE tool. The approach presented in this thesis, to integrate animation facilities with a generic tool which generates environments for different visual modeling languages based on a formal specification, provides the model designer with more flexibility, as the modeling language to be enhanced by animation features, can be freely chosen.

Chapter 5

Implementation

In this chapter we present the implementation of the concepts for simulation and animation (Chapters 3 and 4) based on visual languages defined by graph transformation (Chapter 2). After an introductory overview in Section 5.1 of different kinds of CASE and Meta-CASE tools, and their abilities to support simulation and animation, we review the GENGED environment [Bar02] in Section 5.2. GENGED is a Meta-CASE tool in the sense that it supports the generation of a specific visual modeling environment from a visual language specification. GENGED is based on the graph transformation engine AGG [Tae04, AGG], which realizes all necessary graph transformations for editing, parsing and simulation (Section 5.2.1). The implementation of our simulation and animation concepts is realized by recent extensions of the GENGED environment [Bar02]. Originally, the generated environment contained a visual editor for the specified VL which could be configured to be syntax-directed (based on a syntax graph grammar) or a free-hand editor (requiring a parse grammar). The most recent GENGED components presented in this chapter comprise the generation of a simulator [BEW03, Wei01] (Section 5.3), based on simulation rules for the specific language or model, and an animation environment [Ehr03] (Section 5.4), based on animation rules which can be extended in an animation editor by continuous animation operations.

5.1 Overview: Tools for Simulation and Animation of Visual Models

In the wide area of visual modeling techniques a large number of CASE and Meta-CASE tools have been developed to define and work with visual modeling techniques. A *CASE tool* is usually dedicated to one individual modeling technique. It supports the editing of models and might offer also support for validation by simulation or animation, for transformation and for code generation. On the other hand, *Meta-CASE tools* offer an automated

or semi-automated support for developing case tools. They support the development (or generation) of (visual) editors, simulators or animation tools, analyzers and testing tools, code generators or model transformation tools.

5.1.1 CASE Tools

Examples for CASE tools supporting visual behavior modeling and simulation (or model execution) are RHAPSODY [Rha05] for UML, STATEMATE [HG97] for Statecharts, or CPN TOOLS (formerly Design/CPN) for Coloured Petri nets [RWL+03, CPN05]. Rhapsody, for example, enables the user to automatically generate code, which can be run allowing the user to check and play with the model. A so-called *diagrammatic animation* can be carried out, meaning that during simulation the user sees the diagrams changing in a way that indicates the dynamic behavior that is being simulated. A diagrammatic animation shows the generation of instances, the switching between states, the events that have been consumed, the events that are about to be consumed, and so on. This representation is detailed and rigorous, and it does help in understanding what is happening in the modeled system, but only if the person running the simulation is used to work with the modeling formalism. Such a diagrammatic animation for UML model is comparable to playing the token game in Petri nets.

To make simulation useful for a broader spectrum of less-technical people, including end-users, CASE tools often allow to use graphical user interfaces (GUIs) to portray the look-and-feel of the modeled system during simulation. GUIs also provide a convenient way of manipulating the model during simulation. Such interfaces are sometimes created using a programming language's visual "forms", or even coded-in while creating the application's code. Other ways to create such a visual interface is to use specific tools, such as Altia [Alt05] or add-ons for existing simulation environments. For Design/CPN, for example, a graphical library called MIMIC was developed in 1995 [LRS95]. During simulations, the MIMIC library allows to show, move and hide graphical objects. Instead of investigating tokens in the CP-net, the user can inspect the state of the simulator via graphical objects (e.g. a picture of a mobile phone). By clicking on button icons or turning knobs, different parameters are conveyed to the running simulation, where the proper operations are performed and the results are sent back to be displayed in the visual interface of the CPN simulator.

It is important to realize, however, that most GUI tools are limited, and are mainly useful for obtaining a rough idea of the highly discrete user interface of the system in operation. They are in fact almost static in nature (like the GUI of a mobile phone). GUI tools are designed to handle a well-characterized set of elements that are found in systems such as watches, phones, cars, calculators, electric appliances etc. They do not supply a general environment allowing to animate selected features of a model at a freely chosen level of

abstraction adapted to the development stage of the system model. Moreover, GUI tools have no general functional abilities or animation techniques. In short, the front end of CASE tools using GUIS for simulating/animating the behavior of models is limited and does not provide the model designer with much animation flexibility.

In this thesis we are interested in a fruitful combination of formal simulation specification on the one hand, and flexible animation techniques enhancing the formal simulation specification on the other hand. Our approach includes the possibility of static GUI interfaces if adequate for the model, but is not limited to it.

5.1.2 Meta-CASE Tools

The use of CASE tools in practice showed that they were not applied as widely as expected. One of the reasons for this is that there are so many different dialects of modeling languages, heterogeneous environments and continuously changing requirements. Thus, many CASE tools do not fit their specific context exactly. This inflexibility of existing CASE systems lead to a a need to build systems which can be adjusted to the individual situation. Meta-CASE tools – tools which are used to build CASE tools – solve this problem. Meta-CASE tools are especially useful when some minor variation of the modeling language is needed for which building a complete tool from scratch would be too expensive. Meta-CASE tools support a graphical, high-level description of the CASE tool to be generated.

Different kinds of Meta-CASE tools are available (see Fig. 5.1).

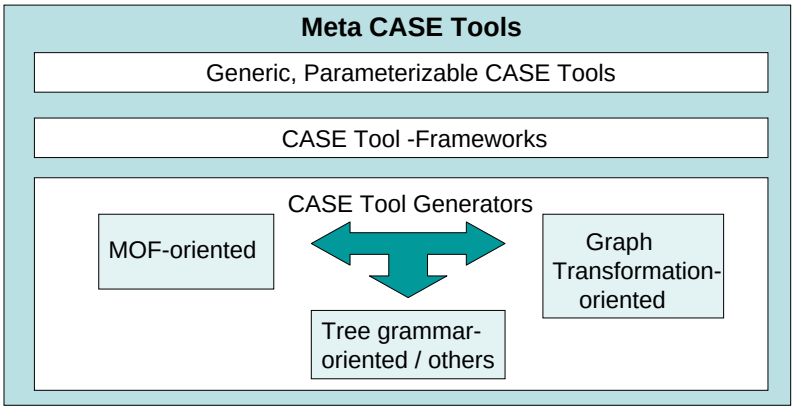


Figure 5.1: Different Kinds of Meta-CASE Tools

Generic, parameterizable CASE tools allow the definition of variants of one main modeling technique (e.g. lots of UML CASE tools offer support for the definition of stereotypes). Early examples of Meta-CASE tools like KOGGE [JE97] and METAEDIT+ [KLR96] belong to this kind of tools. *CASE tool frameworks* (such as ECLIPSE/EMF/GEF) [Ecl04]

can be used to generate reusable, semi-complete code to be extended to a specific CASE tool. *CASE tool generators* offer designer components for the specification of visual modeling environments and their generation from the given specification. Such generators are either *graph transformation-based* and/or *MOF-based*. Using MOF, symbols, links and multiplicity constraints are described by class diagrams, while well-formedness rules define the language syntax (see Section 2.2.1.1). Basing the specification of a visual modeling technique on graph transformation, the visual alphabet is described by type graphs (conf. Section 2.2.2), graph grammars define the language syntax (conf. Section 2.2.4.2), and graph transformation systems can be used for the behavior definition (see Chapter 3). Tools like GENGED [Bar02], TIGER [EEHT04] and DIAGEN [Min01] belong to this group of Meta-CASE tools. Also mixed approaches are possible, where the definition of the visual language is done according to the MOF approach, but components for simulation or model transformation are defined by graph transformation systems. Examples for tools using this mixed approach are ATOM^3 [dLVA04], FUJABA [BGH$^+$05], and VIATRA [CHM$^+$02]. CASE tool generators either generate source code which may be further adapted (e.g. TIGER), or they provide a kernel environment which interprets the language specification and is thus extended to the specific VL environment (e.g. GENGED and ATOM^3).

In this thesis, the implementation of simulation and animation of visual models is realized by extending the GENGED Meta-CASE environment [Bar02] for visual language definition, which is based on the graph transformation engine AGG [Tae04], both developed at the TU Berlin. This allows us to consider various visual behavior modeling languages such as different Petri net and Statechart variants within one unifying VL modeling environment on the basis of typed attributed graph transformation systems. For the concrete syntax of visual languages, GENGED allows to define not only graph-like languages (where symbols are represented as icons and links between symbols are arcs or lines connecting these icons), but covers also more complex layout conditions like an *inside* relation between two symbol types, or flexible size or position dependencies between symbol graphics. Moreover, the import of *gif* or *jpg* images for symbol graphics is possible, which allows the animation designer to use arbitrary symbols for animation, created with arbitrary graphical drawing tools, or even to use photos.

Related Meta-CASE tools supporting the simulation of visual behavior modeling languages based on graph transformation, are DIAGEN [Min01] and ATOM^3 [dLVA04]. DIAGEN also allows to define "animations" in the sense that smooth animations can be assigned to graph objects in the simulation rules. But the animations are restricted to the elements of the specified visual language. In DIAGEN, a Java programming framework allows to implement *Move* animations for objects occurring in the simulation rules in Java [MH01, MG97]. ATOM^3 supports to add time durations to simulation rules, allowing to visualize a simulation run with an adequate lapse of time between each two state

changes. More sophisticated animations could be realized in ATOM^3 theoretically by implementing PYTHON code which has to be executed during the simulation runs.

In contrast to our approach, both tools do not provide an integrated visual environment for visually specifying animation operations in close relation to the simulation rules. Moreover, DIAGEN and ATOM^3 do not consider a layout transformation of the formal simulation language to the layout of the animation domain or to another abstraction level. Both tools implement the visualization of the simulation/animation within the generated simulator (online-animation), whereas we realize a more flexible offline-animation by generating SVG code [WWW03] from an animation specification. SVG is an XML-based format for the animation of scalable vector graphics. To view SVG-animations, various commercial and free viewers exist, and even the newer versions of web browsers (such as the Internet Explorer within Microsoft Windows) can be used.

Running the animation serves as an explanatory tool to the driving simulation. It tells a visual story that comes as close as we want to a real system, limited only by the graphical power of the animation creation tool. In our case, we restrict this power to the use of 2-dimensional graphics, because this is in most cases sufficient to give an impression of the system's functional behavior. Moreover, the design of animations on the basis of simulation specifications should be as simple and straightforward as possible and not be reserved to experts skilled in designing e.g. components in X3D [X3D05], the new Open Standards XML-enabled 3D file format, (following the virtual reality modeling language VRML) or having access to sophisticated 3D-animation software.

5.2 The Basic GENGED Environment for Visual Language Definition

In this section, we briefly present the basic GENGED environment (before the recent extensions for simulation, view definition and animation). The basic GENGED environment is a generator for visual editors based on the visual language specifications. Detailed information about the necessary hardware and software requirements to install and run the GENGED system, as well as a user manual, a system design document, example and all packages for download can be found at the GENGED homepage [Gen].

The basic structures in GENGED are given by a *VL alphabet* (a type graph and a graphical constraint satisfaction problem) and a typed, attributed syntax graph grammar. The *alphabet editor* supports the definition of symbol types and how they are linked, which results in a type system for all instances. Based on an alphabet, the grammar editor allows for defining graph grammars typed over the alphabet. The resulting VL specification (an alphabet and a syntax grammar) configures the aimed VL editor. For the storage of the VL specification, XML [Tae01] is used.

In all GENGED components, we distinguish the *abstract syntax* and the *concrete syntax* of VLs. The abstract syntax level of diagrams is defined by typed attributed graphs, which are manipulated by AGG, a visual interpreter for typed, algebraic graph transformation with integration of Java programs [AGG, Tae03, TER99]. The concrete syntax is given by specific node attributes (defining graphical shapes for symbol types), and by graphical constraints for the correct arrangement of symbol graphics. The constraint satisfaction problems are solved by the integrated graphical constraint solver PARCON [Gri96].

Fig. 5.2 illustrates the architecture of the basic GENGED environment for visual language definition including the tools used (indicated by dashed arrows).

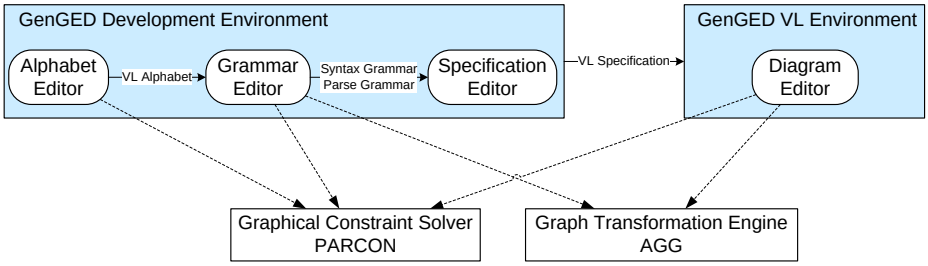


Figure 5.2: Architecture of the GENGED Environment for Visual Language Definition

As AGG is the central graph transformation engine, used on the one hand for the definition of the VL syntax, and on the other hand, for editing and parsing of diagrams in the generated VL editor, we introduce the concepts of AGG before going into details of the definition of VL alphabets and syntax grammars in GENGED.

5.2.1 The AGG Graph Transformation Machine

AGG is a general development environment for algebraic graph transformation systems which follows the interpretative approach [AGG]. Its special power comes from a very flexible attribution concept. AGG graphs are allowed to be attributed by any kind of Java objects. Graph transformations can be equipped with arbitrary computations on these Java objects described by a Java expression. The AGG environment consists of a graphical user interface comprising several visual editors, an interpreter, and a set of validation tools. The interpreter allows the stepwise transformation of graphs as well as rule applications as long as possible. AGG supports several kinds of validations which comprise graph parsing, consistency checking of graphs and conflict detection in concurrent transformations by critical pair analysis of graph rules. Applications of AGG include graph and rule-based modeling of software, validation of system properties by assigning a graph transformation based semantics to some system model, and graph transformation based evolution of software.

On the one hand, AGG comes with its own visual development environment including graphical editors for graphs and graph transformation rules, on the other hand AGG offers

an interface for the use of the graph transformation machine to external tools which have their own graphical user interface (such as GENGED). Right now, AGG version 1.2.1 is integrated in GENGED and supports the definition of visual languages and parsing of visual diagrams in GENGED.

In AGG *typed attributed graphs* consisting of attributed nodes and directed edges are the basis for graph transformation. The attribution of vertices and edges[1] by Java objects and expressions follows the ideas of attributed graph grammars introduced in [LKW93]. Two kinds of transformation concepts can be realized, namely the *Single-Pushout* (SPO) and the *Double-Pushout* (DPO) approach (see Section 2 for an overview of graph transformation approaches). Applying a rule in the SPO approach, all dangling edges are deleted implicitly, whereas in the DPO approach the application of a rule is forbidden if the resulting graph would have dangling edges (cf. [EHK+97] for more details).

For syntax checking, the AGG *graph parser* is included, based on the parse algorithm proposed in [BST00], which uses *Contextual Layered Graph Grammars* (CLGG) and critical pair analysis: Assigning rules as well as node and edge types to layers such that the layering condition in [BST00] is satisfied, the layer-wise application of rules to a given graph always terminates. Roughly speaking, the layering condition is fulfilled if each rule deletes at least one node or edge coming from a lower level (*deletion layer*) and creates graph objects of a higher level (*creation layer*). *Critical pair analysis* [LM95] is used to make parsing by graph transformation more efficient: Decisions between conflicting rule applications are delayed as far as possible. This means to apply non-conflicting rules first and to reduce the graph as much as possible. Afterwards, rule application conflicts are handled by creating decision points for the backtracking part of the parsing algorithm. For critical pair analysis of CLGG rules [BST00], a layer-wise analysis is sufficient, since a rule of an upper layer is not applied as long as rules of lower layers are still applicable.

Amalgamated Graph Transformation in AGG

A recent extension of the AGG tool, initiated from the needs to define amalgamated simulation specifications for certain visual languages (see Section 3.3), has been realized in the master thesis of B. Karwan [Kar05], where amalgamated graph transformation (based on the *locall*-covering construction) is implemented in AGG.

A rule scheme can be defined by drawing a subrule and arbitrary many elementary rules which are added to the navigation tree. Fig. 5.3 shows in the upper half a subrule and in the lower half an extended rule for simultaneous transition processing of Statecharts with AND-States (see Section 3.3.2).

Afterwards, an operation *amalgamate* can be evoked, leading to the computation of an amalgamated rule depending on the current working graph. Negative application condi-

[1] Attributes for edges are not used by GENGED.

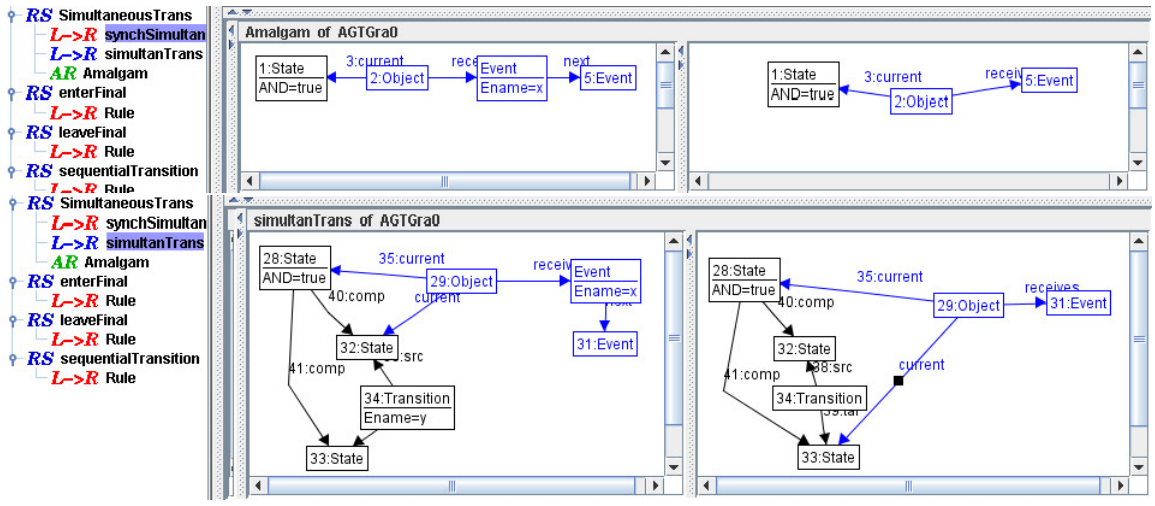


Figure 5.3: Construction of Rule Schemes

tions of the subrule and the elementary rules are adopted by the amalgamated rule. The amalgamated rule is also added to the navigation tree, and the match is set according to the match used for the construction of the amalgamated rule (see Fig. 5.4).

A transformation step using an amalgamated rule leads to the replacement of the working graph by its transformed working graph, analogously to sequential graph transformation in AGG. It is possible to perform a transformation step using a rule scheme without having explicitly computed the amalgamated rule before. In this case, the amalgamation and the transformation are realized in one step, and the amalgamated rule is not added to the navigation tree.

5.2.2 Definition of the Visual Alphabet

Formal Concepts for Visual Alphabets. In GENGED, the abstract syntax of an alphabet is given by an attributed graph structure signature [Bar00]. This means that symbols are defined by sorts S_{GS}, and links by unary operations OP_{GS} of a graph structure signature $GS = (S_{GS}, OP_{GS})$. Attributes are defined by a separate attribute signature, and the attribute sorts are linked to the symbol sorts by attribution operations.

In comparison to the concepts of visual alphabets discussed in Chapter 2, a graph structure signature corresponds to the abstract type graph in Def. 2.2.2. A significant difference is the restriction in GENGED alphabets, that operations (links) have to be *total* (see also Ehrig's comparison of different formal representations of typed attributed graphs [Ehr04a]). Links being defined as total operations in the alphabets imply the following restrictions for visual diagrams of the alphabet (defined formally in GENGED as

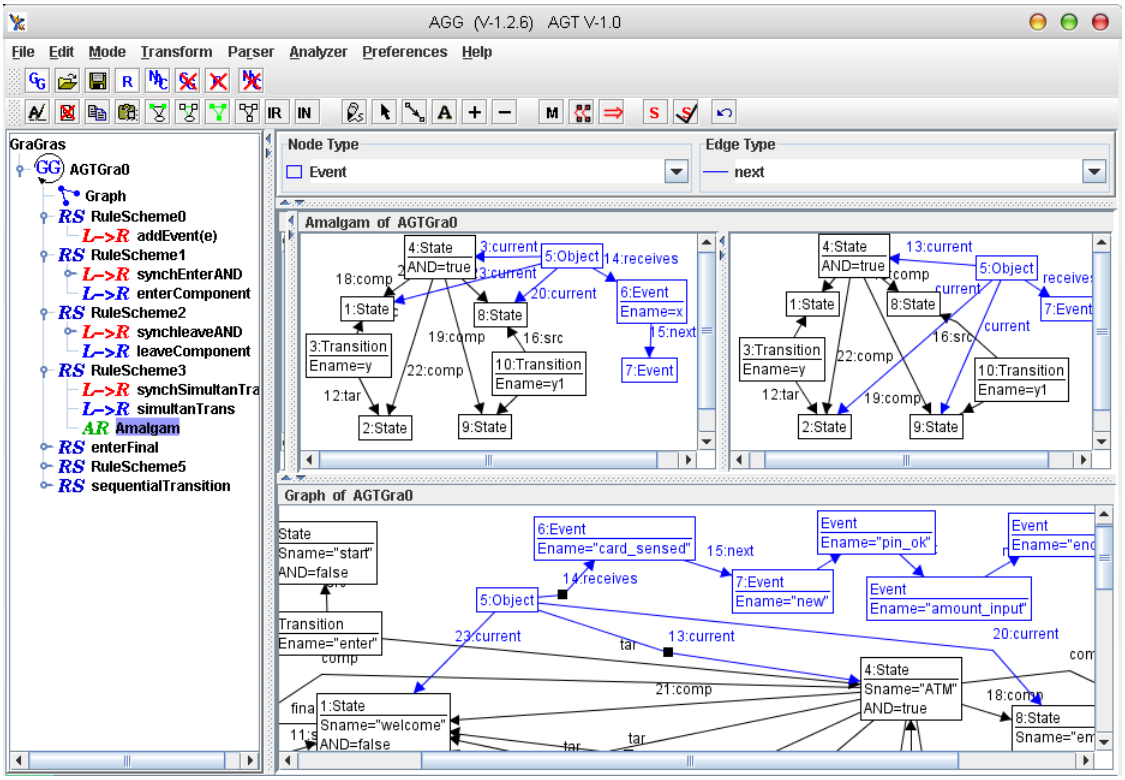


Figure 5.4: Computation of Amalgamated Rules from a Rule Scheme

graph structures with respect to the graph structure signature): Each symbol in the diagram where the symbol type is a source of an operation (a link type) in the alphabet, has to be the source of a corresponding link in the diagram. The advantage of this restriction is that it ensures that edge symbols, such as an association in class diagrams, are always linked by begin and end links to corresponding node symbols. The disadvantage is that alphabets have to be defined in a way that for link types where links do not necessarily have to exist for all nodes of the link type's source node type, auxiliary node types have to be defined.

Let us compare as example the VL alphabets for Statecharts with nested OR-states as defined in Section 2.3.2.1 (Fig. 5.5 (a)), to the corresponding GENGED alphabet (Fig. 5.5 (b)). In the VL alphabet in Fig. 5.5 (a), the hierarchy of nested states is modeled by a link type super as loop at the State symbol type. But as not all states have a super state, this hierarchy relation must be modeled in a slightly different way in GENGED, where links correspond to total operations. Hence, in the GENGED alphabet in Fig. 5.5 (b), a Hierarchy symbol type is defined with links of types super and sub linking it to two states. Such a Hierarchy node is present between two states in a diagram only if one of the states is a super state of the other one. The nesting hierarchy is layouted in GENGED by the help of an invisible box (depicted as dashed rectangle). A link super from a Hierarchy symbol to a state evokes a layout constraint overlap which places the hierarchy box and the shape for

the super state at the same position with the same size. The constraint includes for the sub link from the Hierarchy symbol to the symbol of the substate ensures that the shape for the substate is drawn inside the hierarchy box. Thus, the substate shape is nested in the super state shape.

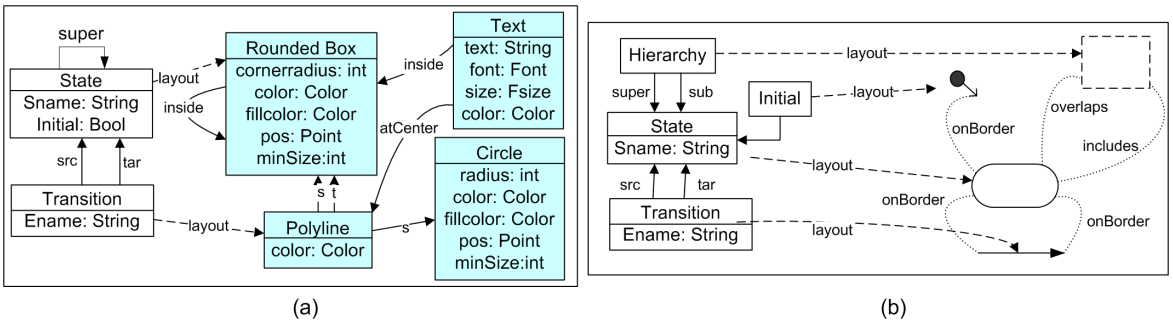
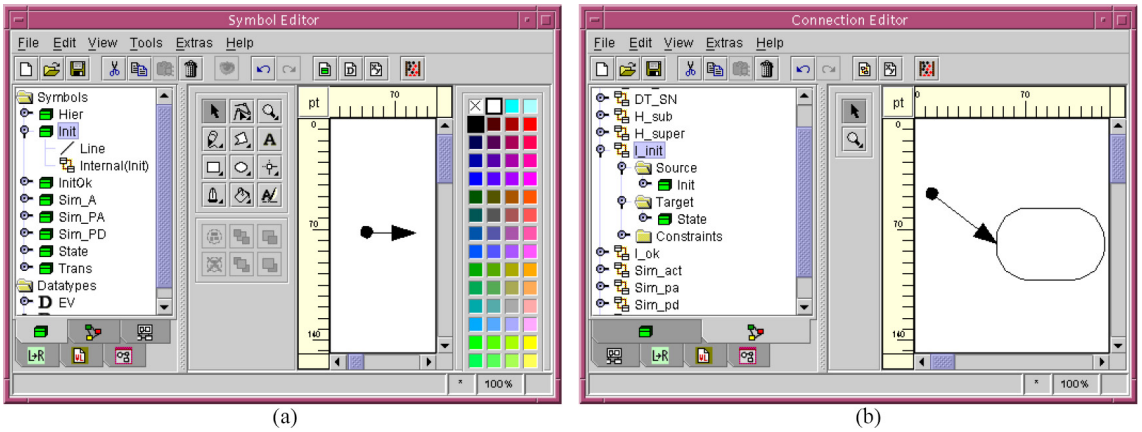


Figure 5.5: VL Alphabets as Type Graph (a), and as GENGED Alphabet (b)

Another restriction imposed by the GENGED concepts concern the definition of the concrete syntax for abstract symbols and links. In GENGED, there is a one-to-one correspondence between abstract symbol types at the abstract syntax level and graphical shapes at the concrete syntax level. This correspondence is again defined by a set of total attribution operations from the symbols in S_{GS} to the sort $Graphic$, where the elements of $Graphic$ are graphical shapes such as rectangle or ellipse. Hence, it is not possible to define a symbol type which may be visualized by different symbol graphics depending of the value of an attribute. For example, in Fig. 5.5 (a), we define a boolean state attribute Initial, which is evaluated to true if the state is an initial state. In this case, it is not only visualized as an ellipse inscribed by the state name, but has additionally a small black circle connected to the ellipse by a small arc (the *initial state* pointer). In the GENGED alphabet in Fig. 5.5 (b), we have to define an extra node symbol type Initial which is linked to state. For this Initial node symbol type, the graphic shape of a small black circle together with a small arc is the corresponding layout. An additional layout constraint onBorder has to ensure that the Initial graphic's arrow touches the border of the corresponding state ellipse.

Alphabet Editor. The alphabet editor supporting the definition of VL alphabets consists of a *symbol editor* and a *connection editor*. The symbol editor (Fig. 5.6 (a)) works similar to well-known graphical editors like xfig, however, the grouping of primitive graphics (in order to define one complex symbol graphic) is realized by graphical constraints. Moreover, according to the abstract syntax, each symbol type (which is represented by a node type) needs a unique symbol name. These symbol names are used in the connection editor (Fig. 5.6 (b)) for defining a connection (or link type), which is represented by an edge between two symbol type nodes. At the concrete syntax level, graphical layout constraints have to be defined in order to connect the corresponding symbol graphics (cf. [BNS00]).



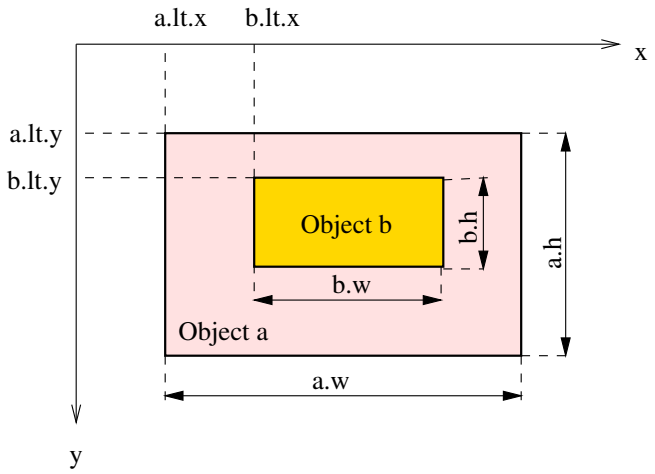
(a) (b)

Figure 5.6: The GENGED Symbol Editor (a), and the Connection Editor (b)

The resulting abstract VL alphabet is represented by a type graph where the symbol types are given by the symbol names, and link types are defined by the connections between the symbols, respectively.

For the concrete VL alphabet, symbol graphics are defined by graphical shapes as attributes of nodes, and layout constraints are defined by graphical constraints which are collected in a constraint satisfaction problem (CSP): An example for a layout constraint is the inside constraint between two graphical shapes (objects), expressing that one object is placed inside the borders of the second object. Such *high-level* constraints are decomposed into *low-level* constraints (equations over position and size variables) in order to be solved by the graphical constraint solver PARCON [Gri96].

Let us consider the decomposition of the high-level layout constraint "*inside(a,b)*" as graphical constraint of the CSP, meaning that object b is placed completely inside the borders of object a. The variable lt indicates the left top corner point of an object, and w and h its width and height. Point coordinates x and y of point P are written $P.x$ and $P.y$. Thus, e.g. $a.lt.x$ denotes the x-coordinate of the left top corner of object a. The constraint is a set of in-equations over these variables: Other layout conditions are formalized by more complex constraints but with the same underlying principle: the scope of possible variable bindings is restricted by equations/in-equations over constraint variables denoting the position or size of graphical objects. The set of constraints defined in the VL alphabet, together with some initial constraints (e.g. all graphical objects are positioned within the editor panel and have a default size) comprise the CSP which has to be satisfied by all visual diagrams of the visual language.



inside(Object a, Object b) {
$a.lt.x < b.lt.x$;
$a.lt.x + a.w > b.lt.x + b.w$
$a.lt.y < b.lt.y$;
$a.lt.y + a.h > b.lt.y + b.h$;
}

Figure 5.7: **Layout condition *inside* as graphical constraint of the CSP**

5.2.3 Definition of the Syntax Grammar

Based on a VL alphabet, the grammar editor allows to define distinguished graph grammars, namely for syntax-directed diagram editing as well as for parsing a diagram. A grammar consists of a (start or stop) diagram and a set of rules, each one comprises a left- and a right-hand side, and optionally a set of *negative application conditions* (NACs), as well as attribute conditions.

Grammar Editor. The grammar editor works in a syntax-directed way: Based on a given VL alphabet so-called *alphabet rules* are generated automatically which function as basic edit commands. Fig. 5.8 shows a screen-shot of a grammar editor with a syntax grammar for the Statechart VL, where the alphabet rule InsertState supporting the insertion of a state symbol is illustrated in the rule panel to the upper right. The left-hand side of this rule is empty, in its right-hand side a state symbol is generated. In the rule editor to the lower right of Fig. 5.8, we defined a new syntax rule S-InsState for inserting a sub-state symbol. On the left-hand side of the syntax rule, we require the existence of a state symbol which is preserved by the rule (indicated by number 1). On the right-hand side, we inserted a further state symbol by applying the alphabet rule to the right-hand side graph of the syntax rule. The navigation tree to the left of Fig. 5.8 holds the names of all defined rules and of the start and/or stop diagrams. A suitable dialog supports the selection of rules and the export of grammars.

The AGG system is used for the transformation of the abstract syntax of diagrams in the grammar editor and in the generated VL environment. The graphical constraints defined with respect to the concrete syntax (the layout) of the VL alphabet are solved by the constraint solver PARCON [Gri96] for VL diagrams after each transformation step so that the computed values for positions and sizes of graph objects yield a valid concrete VL diagram.

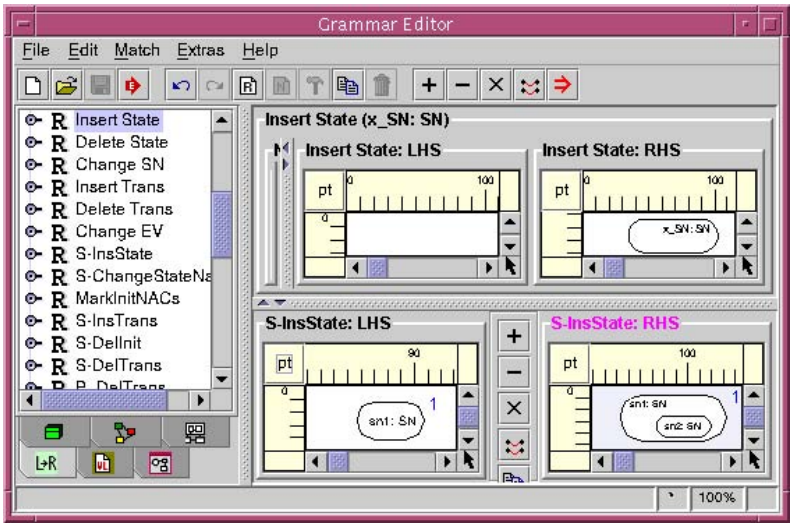
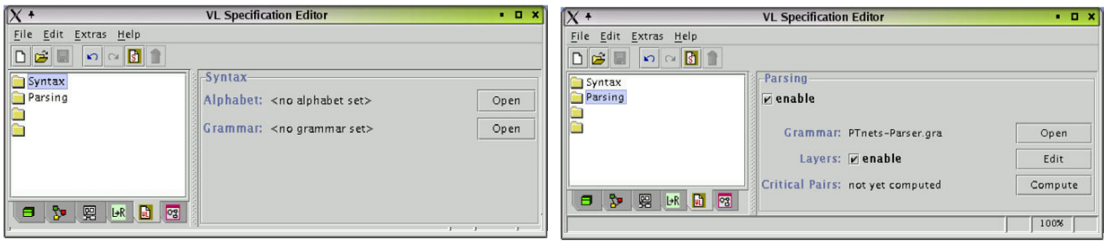


Figure 5.8: The GENGED Grammar Editor

VL Specification Editor. Based on the definitions of the VL alphabet and syntax or parse grammars, the specification editor allows to establish a *VL specification*. Syntax-directed editing can be defined by loading the VL alphabet and a VL syntax grammar (see Fig. 5.9 (a)). A parse grammar may be extended by the definition of a layering function and a critical pair analysis in order to optimize the parsing process (Fig. 5.9 (b)). The parse grammar together with these extensions result in a *parse specification*.



(a) (b)

Figure 5.9: The GENGED Specification Editor for the Specification of Syntax (a), and Parsing (b)

5.2.4 The Generated VL Environment

A VL environment generated by GENGED contains at least a *visual editor*, (which, optionally, may contain a *parser*).

Generated Visual Editor. The generated visual editor shown in Fig. 5.10 supports syntax-directed diagram editing in a way similar to the grammar editor: The navigation

tree on the left holds all the names of rules that can be applied for manipulating a diagram in the lower part of the editor. Similarly to the grammar editor, a selected rule is visualized in the upper part of the diagram editor. A rule can be applied after defining mappings from the symbols in the rule's left-hand side to symbols in the diagram. A parse grammar must be available if free-hand editing should be enabled in a visual editor. For free-hand editing the editing rules are much weaker than for syntax-directed editing and allow to edit intermediate diagrams which are not sentences of the visual language. Only the parsing process (i.e. applying the rules of the parse grammar) decides whether the diagram is syntactically correct or not.

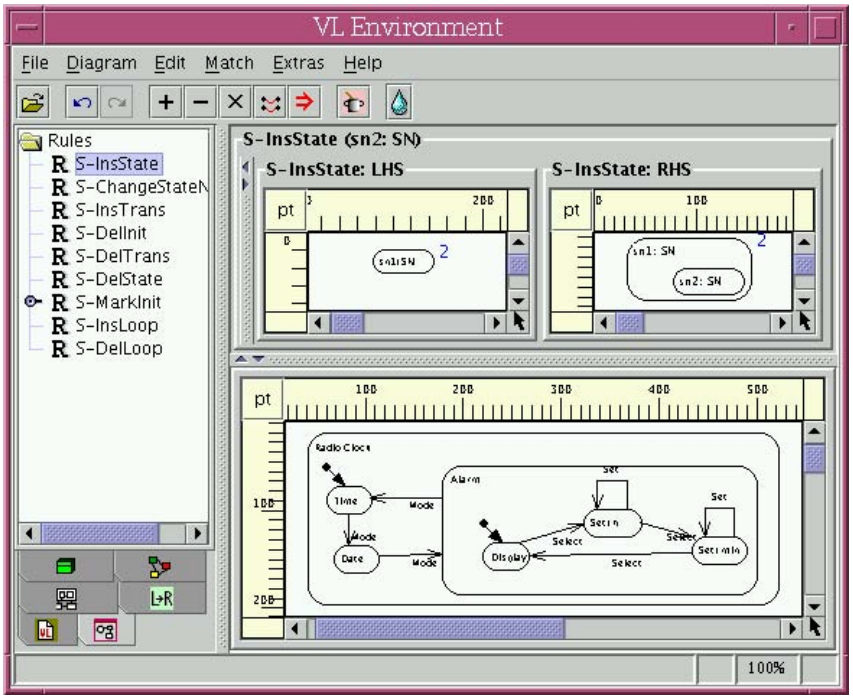


Figure 5.10: The Generated Visual Editor

5.3 The Simulation Environment of GENGED

In this section we describe the extension of GENGED in order to provide tool support for the generation of simulators, and the definition of simulation runs by simulation run specifications (controlled application of simulation rules).

Simulation rules are defined using the grammar editor. A simulation specification may either be specified in a way that the rules are universal, i.e. they can be applied to simulate arbitrary models of the VL (according to Def. 3.1.2 of $SimSpec(VL)$ in Section 3.1), or the simulation rules are specified in a model-specific way, i.e. they can be applied only

to simulate the behavior of one specific model (according to Def. 3.2.2 of $SimSpec(M)$ in Section 3.2). Simulation rules can be grouped in so-called simulation steps (controlled application units of simulation rules) using the specification editor for simulation. From such a *simulation run specification*, a simulator is generated as component of the generated VL environment. The GENGED architecture extending the GENGED environment from Fig. 5.2 by features for simulation run definition and simulator generation, is shown in Fig. 5.11.

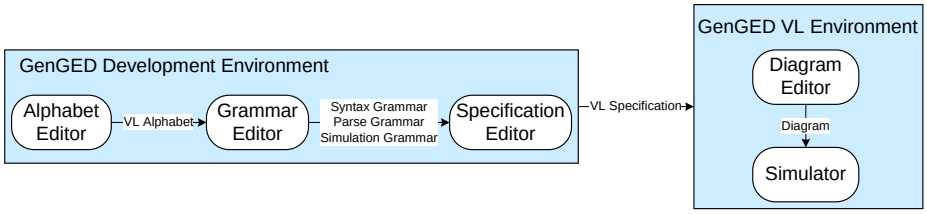


Figure 5.11: Architecture of GENGED with Simulation Environment

Simulation Run Specification. Simulation runs are specified in a *simulation run specification*, consisting of a simulation grammar (i.e. a simulation specification together with a start diagram) and a set of simulation steps, which specify how the simulation rules have to be applied, using branch and loop constructs as control structures.

This is realized in the GENGED environment by an extension of the specification editor, which offers support to edit simulation steps based on an existing simulation specification. A simulation step describes an atomic step to be executed during the simulation process consisting of possibly many rule applications. The core of a simulation step is a simulation expression, a kind of iterative program executed when the step is being applied in the simulator of the VL environment.

Fig. 5.12 shows the GENGED specification editor for the specification of simulation steps for the simulation of Statecharts with OR-states..

The GUI of the simulation step definition dialog contains three parts: the structure view on the left, a button bar in the middle and a working display on the right. The *structure view* holds a list of parameters and the expression tree of the current simulation step. The *button bar* offers functions to manipulate these structures. If there is a parameter or expression selected in the structure view, the *working display* shows the objects necessary for manipulating the selected parameter or expression. The button bar contains three different groups of buttons. The first group Insert allows to add expressions to the structure of the simulation step, the second group Move supports sorting of expressions, and the third group Remove contains only one button for deleting an expression from the simulation step.

The syntax and semantics of expressions is defined as follows: Basic expressions are rules, and $true$ and $false$. Complex expression are defined over expressions $e_i, i \in$

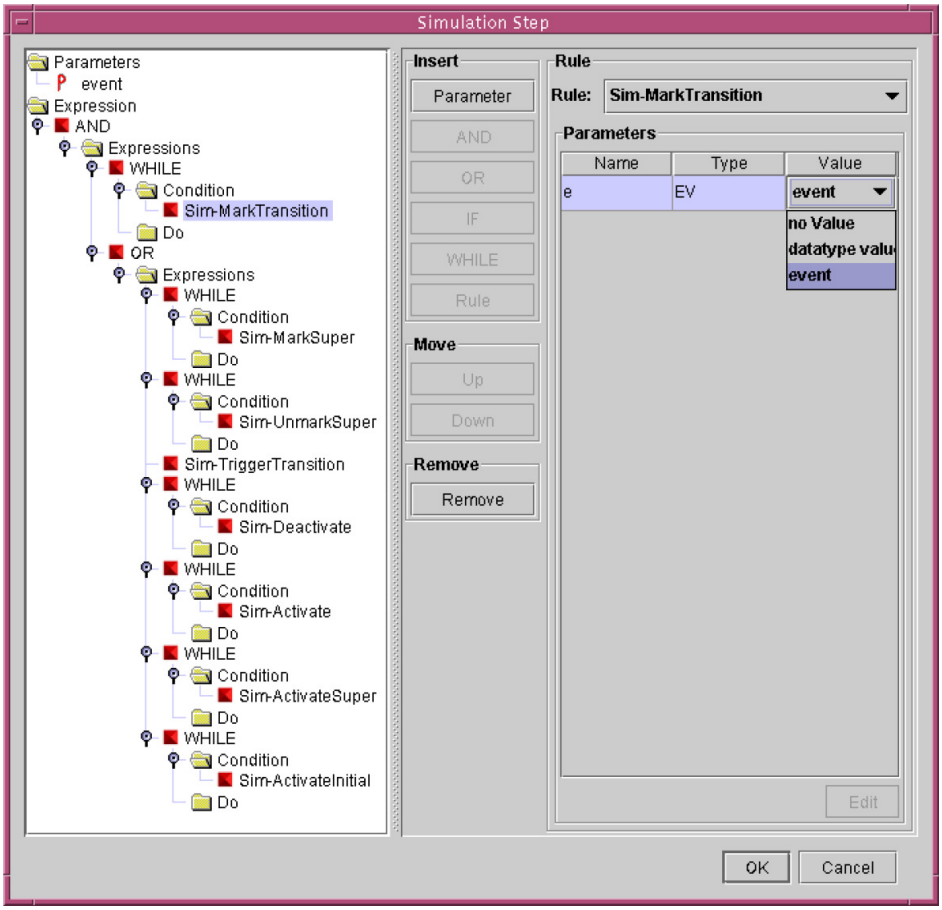


Figure 5.12: The GENGED Specification Editor for the Specification of Simulation Steps

$\{1, .., n\}$ by $AND(e_1, .., e_n), OR(e_1, .., e_n), IF(e_1, e_2, e_3)$ and $WHILE(e_1, e_2)$. The evaluation of an expression yields, besides a transformed diagram, a boolean value, which can be used as the condition subexpression e_1 of the $IF(e_1, e_2, e_3)$ and $WHILE(e_1, e_2)$ expressions. How this value is calculated is obvious for $true$ and $false$. Given a rule as expression, the evaluation yields $true$ if the rule can be applied, otherwise $false$. For an AND expression, the sequential list of expressions $e_1, .., e_n$ is evaluated until an expression yields $false$ or the end of the list is reached. In the first case, the AND expression is evaluated to $false$, otherwise to $true$. For an OR expression, the sequential list of expressions $e_1, .., e_n$ is evaluated until an expression yields $true$ or the end of the list is reached. In the first case, the OR expression is evaluated to $true$, otherwise to $false$. For an $IF(e_1, e_2, e_3)$ expression, first the condition subexpression e_1 is evaluated. If it yields true, the $THEN$-subexpression e_2 is evaluated, otherwise the $ELSE$-subexpression e_3. The IF expression yields the value from the evaluation of the chosen subexpression (e_2 or e_3). For a $WHILE(e_1, e_2)$ expression, first the condition subexpression e_1 is evaluated. If it yields $true$, the loop body subexpression e_2 is evaluated. The $WHILE$ expression

yields $true$ if e_2 is evaluated at least once, otherwise $false$. (For a formal definition of the semantics of expressions, see [BEW02b].)

The buttons from the button bar have the following functionalities:

Parameter: Adds a parameter at the end of the parameter list and allows to define predefined values for rule parameters.

AND: Adds an AND-expression at the selected position to the structure.

OR: Adds an OR-expression at the selected position to the structure.

IF: Adds an IF-expression at the selected position to the structure.

WHILE: Adds a WHILE-expression at the selected position to the structure.

Rule: Adds a rule (basic expression) at the selected position to the structure.

Up: Moves the selected expression from a list one position higher in the list.

Down: Moves the selected expression from a list one position lower in the list.

Remove: Removes the selected parameter or expression.

If a parameter is selected, in the working display a text field is shown where the parameter name can be changed and its type can be chosen. If a rule is selected, all rule parameters must be bound to data values or to simulation step parameters. A data value can be changed in a separate dialog after selecting the "Edit" button.

Generated Simulator. A simulator (see Fig. 5.13) is generated from a given simulation run specification.

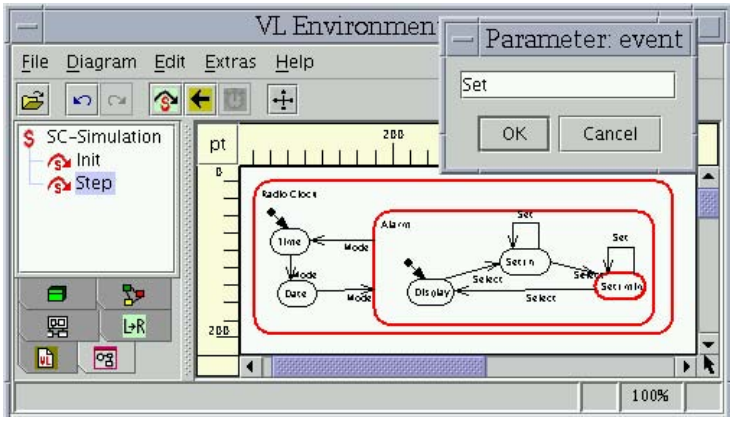


Figure 5.13: The Generated Simulator

The simulator directly shows the model to be simulated, and the names of specified simulation steps in the navigation tree. Selecting a name, the corresponding simulation step is activated. In Fig. 5.13 we can see an advanced simulation of the Radio Clock Statechart (see Section 3.1.2, Fig. 3.2). All active states are highlighted by a fat, colored border. Selecting a further Step effects the dialog asking the user to define a value for the parameter *event*, whereupon the respective successor states are marked by fat borders.

5.4 The Animation Environment of GENGED

In this section we describe the recent extensions of GENGED in order to provide tool support for the generation of animation tools (animators). To this end, GENGED is extended by means to allow the *integration* and *restriction* of alphabets on the basis of existing alphabets (type graphs), as described in Section 4.2.1. Moreover, *meta transformation* is supported by defining and applying the rules of a so-called *meta grammar* to the rules and the start graph of an existing grammar, based on Def. 4.2.12 of rewriting rules by rules in Section 4.2.2. This enables the animation view designer to define $S2A$ transformation rules as meta grammar. By applying them to a simulation grammar, $S2A$ *transformation* is realized, resulting in a set of animation rules and a start graph for the animation. The addition of animation operations to animation rules can be done using the visual *animation rule editor*. The application of the extended animation rules can be visualized as smooth movements and movie-like state changes instead of discrete execution steps. Animation rule sequences can be compiled by defining a control structure on the application of animation rules, similarly to the control of simulation steps. Resulting animations can be exported to the SVG (scalable vector graphics) format and thus viewed by any SVG-viewer or SVG-capable browser.

The GENGED architecture extending the GENGED environment from Fig. 5.11 by features for animation run definition and animator generation, is shown in Fig. 5.11

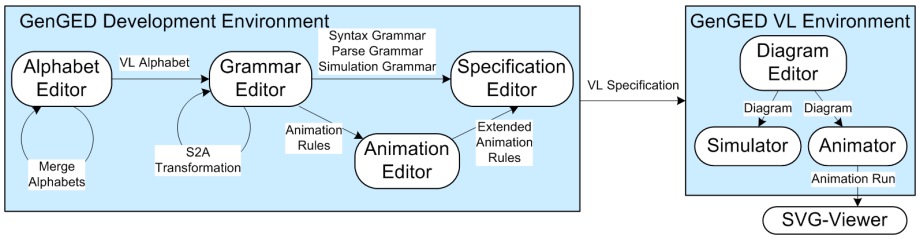


Figure 5.14: Architecture of GENGED with Simulation and Animation Environments

Summarizing, we describe the implementation of the following extensions of the GENGED environment allowing to specify animations for visual behavior models:

- *Merge Alphabets* functionality: allows the definition of animation alphabets in the GENGED alphabet editor, where two alphabet are merged via a common interface,
- *Define View* functionality: allows the definition of restrictions to a sub-alphabet (a view) in the generated simulation or animation environment of GENGED. Thus, the transformations of a scenario can be viewed (and exported to SVG) either in the layout of the animation domain or in the layout of the formal behavior model.
- *Apply Meta Grammar* functionality: allows the realization of a $S2A$ transformation by defining and applying a meta grammar, containing the $S2A$ transformation rules, to a simulation grammar,
- *Animation Rule Editor*: allows to visually specify animation operations enhancing animation rules.
- *Animation Specification*: an animation specification consists of a VL alphabet, a VL syntax grammar, an animation grammar (generated from a simulation grammar by $S2A$ transformation), and additional animation operations, which have been specified using the animation rule editor.
- *SVG export* functionality: allows to export animation scenarios into the XML-based SVG format [WWW03].

The functionalities listed above are presented in detail in the following sections.

5.4.1 Animation View Definition

Merging alphabets over a common interface alphabet. In order to support the definition of animation alphabets, the alphabet editor has been extended by a *Merge Alphabet* action allowing to define a new alphabet as integration of two different basic alphabets, called *views*. One alphabet has to be loaded in the alphabet editor (the first basic alphabet). In Fig. 5.15, the alphabet for marked P/T nets has been loaded in the alphabet editor in Window 1. Evoking the menu item *Merge Alphabet* from the File menu, the user selects the file name of the file containing the second basic alphabet in the pop-up dialog (Window 2 of Fig. 5.15). The resulting integrated alphabet contains all symbols and links, including their concrete layout, from both basic alphabets, as can be seen in Window 3 of Fig. 5.15. The common symbols and links (comprising the *interface type graph* of the two alphabets, defined formally in Def. 4.2.2) are identified by equal names in the basic alphabets. These common objects are glued and appear only once in the integrated alphabet. The concrete syntax of glued items (if different in the basic alphabets) is defined by the concrete syntax of the first basic alphabet, i.e. the alphabet from which the *Merge Alphabet* functionality

is evoked. The information about the original basic alphabet(s) (views) is stored for each symbol of the integrated alphabet, and shown in the navigation tree of the alphabet editor. In Fig. 5.15, Window 3, for example, each symbol belongs either to the view PTnet, or to the view Echo or to both (symbol type Net). This allows later to define a restriction of models of the integrated alphabets to a selected view. To integrate more than two basic alphabets, two alphabets have to be merged first, and the resulting integrated alphabet can be integrated again with a third alphabet, etc.

The feature *Merge Alphabets* is used for the definition of the animation alphabet. Fig. 5.15 shows the definition of the animation alphabet for the Echo model (see Fig. 4.3) by merging the simulation alphabet (e.g. the alphabet PTnet for marked Petri nets) with the alphabet Echo of the animation domain (the view showing the host nodes and message arcs of the modeled network).

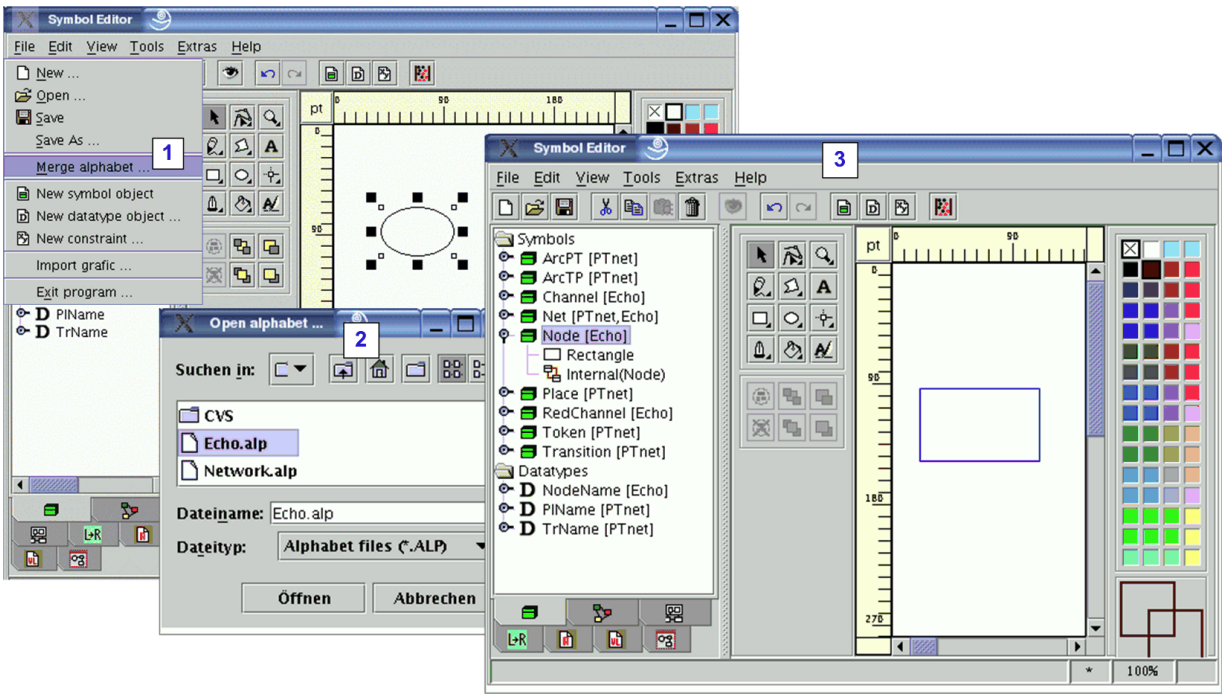


Figure 5.15: Merging Alphabets in GENGED

Restriction of models and scenarios to selected views. In the generated GENGED components, i.e. the visual editor, the simulator or the animator, models can now be restricted to a selected view (a basic alphabet) if the underlying VL alphabet has been defined as integration of two or more basic alphabets. Fig. 5.16 shows the generated animator for the Echo model. A view may be activated by selecting the *Alphabet Views* item from the *File* menu. (see *File* menu in Window 1 of Fig. 5.16). An alphabet views dialog appears,

where all possible views are listed, and one or more views can be selected. By default, all potential views are active (see Window 2 in Fig. 5.16). The case that all views are selected, corresponds to the selection of the complete underlying integrated alphabet. In this case, no restriction takes place, and all model elements typed over the integrated alphabet are shown. Otherwise, only the model elements are shown which are typed over one or more of the selected basic alphabets.

The feature *Alphabet Views* is used for the definition of the animation view in the generated animator as restriction of the integrated models, which are defined as diagrams over the integrated simulation/animation alphabet, to the alphabet of the animation domain. For our Echo model in Fig. 5.16, the view PetriAlphabet is deactivated, which leads to the restriction of all presented diagrams to the Echo alphabet (the animation view). Window 3 in Fig. 5.16 shows a snapshot of the resulting presentation of the Echo model's behavior in the animation view, where only the network host nodes are shown, and the message arcs connecting them.

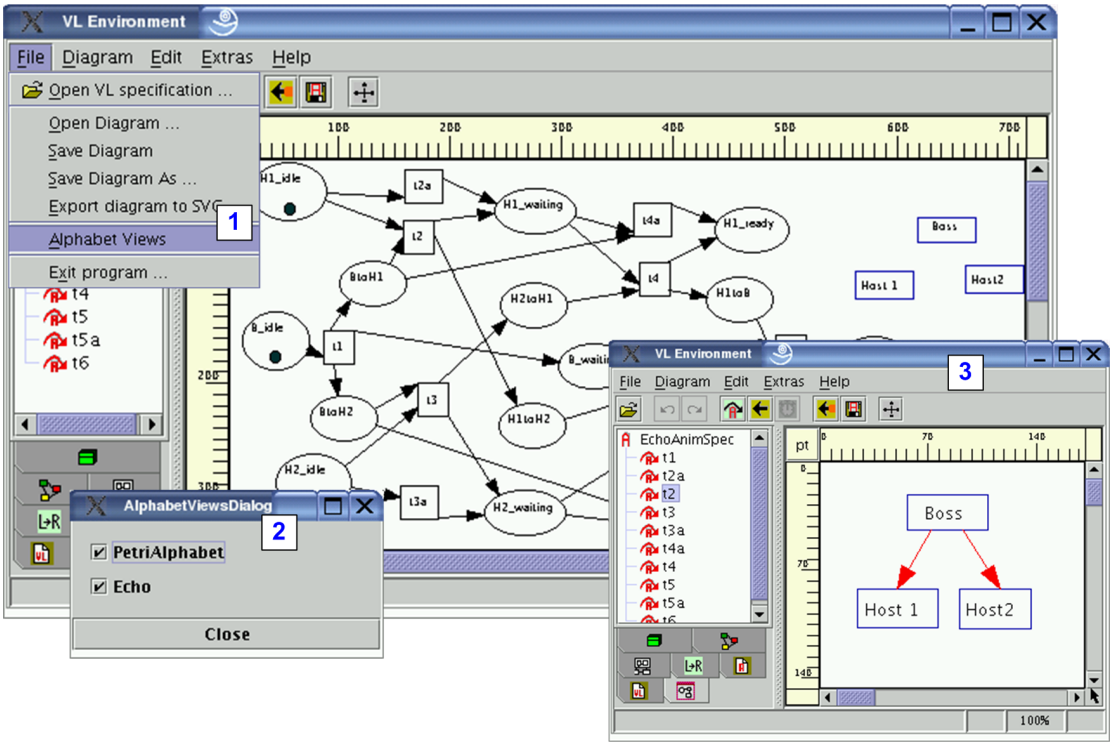


Figure 5.16: Restriction to Selected Alphabet Views in GENGED

5.4.2 Meta Transformation

To support rewriting of rules by rules, so-called *meta grammars* may be defined, using the Grammar Editor in the usual way to define rules, but leaving the start graph of the meta

grammar empty. Having defined a set of meta rules, an *Apply Meta Grammar* action can be evoked from the grammar editor where the currently loaded grammar is the grammar to be transformed. Selecting the Menu item File > Apply Meta Grammar (Window 1 in Fig. 5.17) applies the selected meta grammar rules to the current grammar according to Def. 4.2.12 in Section 4.2.2. Starting with the first meta rule, each meta rule is applied as often as possible to each rule. The meta grammar must be designed such that the application of the meta rules terminates (see Section 4.3 for formal termination criteria of $S2A$ transformation). One way to ensure this is to define a NAC equal to the RHS for each meta rule, such that each meta rule can be applied at most once at each match.

The feature *Apply Meta Grammar* is used for $S2A$ transformation to apply the $S2A$ transformation rules to the simulation rules and to the VL diagram defined as start graph of the simulation grammar (e.g. the simulation grammar of the Echo model in Window 2 of Fig. 5.17). This application results in the corresponding animation grammar (Window 3 of Fig. 5.17). In order to be able to apply a meta grammar, the meta transformation rules, as well as the simulation grammar have to be typed over the animation alphabet, usually constructed before by merging two alphabets.

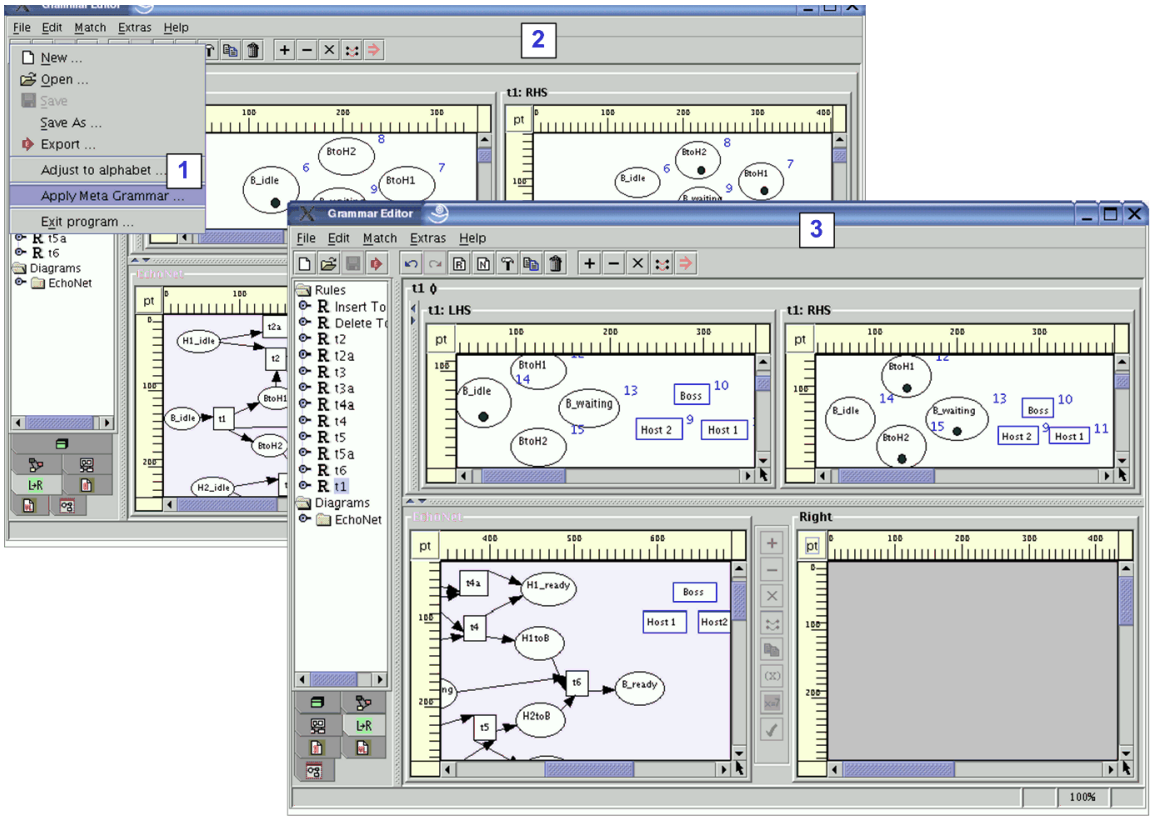


Figure 5.17: Applying Meta Transformation in GENGED

5.4.3 Generator for Animation Environments

The major extension of GENGED regarding support for animation based on simulation specifications is the integrated component allowing to generate animation environments, implemented by Karsten Ehrig in his master thesis [Ehr03]. The visual animation rule editor allows to extend simulation rules by adding animation operations which describe continuous changes of the graphical objects occurring in the rule. On the basis of the animation rules, an animation specification can be defined, based on which an *animation environment* is generated allowing to define animation scenarios and generating animation data in SVG format [WWW03].

The detailed architecture of the extended GENGED environment including the generator for animation environments is depicted in Fig. 5.18.

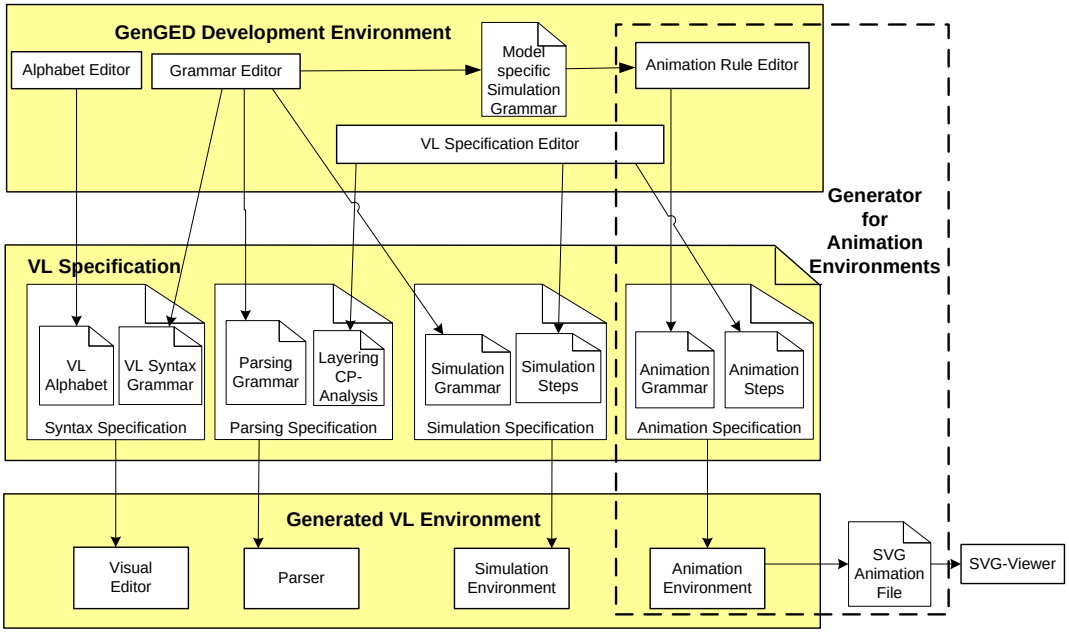


Figure 5.18: Animation Environment in the GENGED Environment

The basis for animation is a model-specific simulation specification, which is defined using the GENGED grammar editor. In the *animation rule editor*, this grammar is extended by animation operations. The consisting VL specification editor is extended by dialog elements for the definition of *animation steps* where the repeated and controlled application of animation rules can be defined, in a way similar to the definition of simulation steps by using expressions which define in which order and how often single animation rules are applied. In this way, several animation rule applications can be combined to an animation step. A complete animation step definition is stored as additional VL specification file (the *animation specification*), which can be loaded in the generated VL environment. In addition to the VL-specific visual editor, an optional parser and a simulation environment, now

a fourth component is generated in the VL environment, namely the *animation environment*. Here, an animation scenario can be executed based on animation step definitions, evaluating the animation operations specified for each animation rule, and generating the respective animation data (i.e. SVG operations). The resulting generated SVG file contains all SVG object data and animation elements for the specified animation scenario. This realizes an off-line-animation, as the generated SVG file can be viewed independently of the GENGED environment by any adequate SVG viewer.

The Animation Rule Editor. The animation rule editor allows to enrich animation rules by continuous animation operations. The GUI of the animation rule editor is depicted in Fig. 5.19, showing the animation rules from the *Dining Philosophers* example.

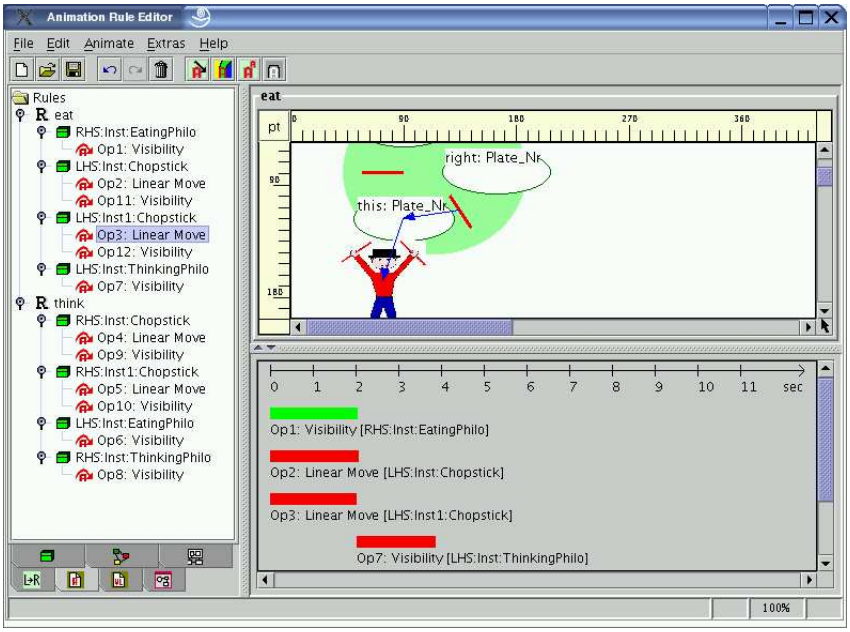


Figure 5.19: Animation Rule Editor of GENGED

Three functional areas can be distinguished:

1. The navigation tree to the left of Fig. 5.19 shows the names of the animation rules and their respective animation operations defined so far (e.g. *Linear Move*).

2. The editor panel in the upper right of Fig. 5.19 displays the selected animation rule in one picture: graph objects which are deleted by the rule are highlighted by red color, and graph objects which are generated by the rule are highlighted by green color. Graph objects which are preserved (the context of the animation), are not highlighted. Animation operations can be defined only for highlighted graph objects. In Fig. 5.19, move-operations for the two chopstick icons are defined. In general, more than one animation operation can be defined for one rule.

3. The time line in the lower right of Fig. 5.19 allows to specify conveniently the starting time and duration of each animation operation defined so far for the current animation rule. If the time bar for an animation operation is moved, a later or ealier start and end time is specified; if the time bar is resized, the duration of the operation is changed. In this way it is possible e.g. to make movements faster by defining a shorter duration for a move-operation, or to define sequences of different movements, or even define animation operations to be visualized in parallel.

The File menu of the animation rule editor allows to define a new animation rule specification by loading a simulation grammar, or to open an existing animation rule specification, or to save the current one to disk. The Edit menu contains the usual editor operations for undo/redo and remove (the currently selected animation operation). The Animate menu allows to select one of the available animation operations *Linear Move, Change Color, Resize Object* or *Visibility*. In the Extras menu, the user may define that all parameters are stored for the next session when GENGED is closed. Most of the menu items can be evoked directly via tool-bar buttons.

The animation operations selected from the Animate menu are specified as follows:

- *Linear Move*: In the editor panel, a linear move-operation is defined by drawing a path with clicks of the left mouse button, from the graphical object to be moved to its source or target position. In case that the object is deleted by the rule, a path from the object to a target position is defined, and in case that the object is generated by the rule, a backward path from the object to a source position is defined. A double-click finishes the path selection. The selected path is shown in the editor panel by blue arrows.

- *Change Color*: A dialog pops up, where a color has to be selected. The animation then starts from the original color of the selected object, and then changes it in a linear way so that after the time defined for the operation has passed, the object has the new selected color.

- *Resize*: A dialog pops up, where the changed size has to be given as percentage of the original size. The animation then starts from the original size of the selected object, and then changes it in a linear way so that after the time defined for the operation has passed, the object has the new size (with the upper left corner remaining fixed).

- *Visibility*: The operation *Visibility* allows to hide an object for a certain time, and then suddenly show it.

Please note that in an animation scenario, the time for a rule application is the time from the beginning of its first animation operation until the end of its last animation operation. Layout attributes that are changed by animation operations are reset to values computed by

the graphical constraint solver for the diagram after the rule application. Thus, the concrete layout of a diagram between two rule applications remains the same, with or without animation operations. Otherwise, the behavior of the system would have been changed by the animation operations, as the derived graphs in an animation sequence *with* animation operations would differ with respect to their concrete layout from the corresponding derived graphs *without* animation operations.

The VL Specification Editor. The existing VL specification editor has been extended by a component to define animation steps, i.e. animation rules coupled by branch and loop control structures. The GUI of the VL specification editor for animation is shown in Fig. 5.20.

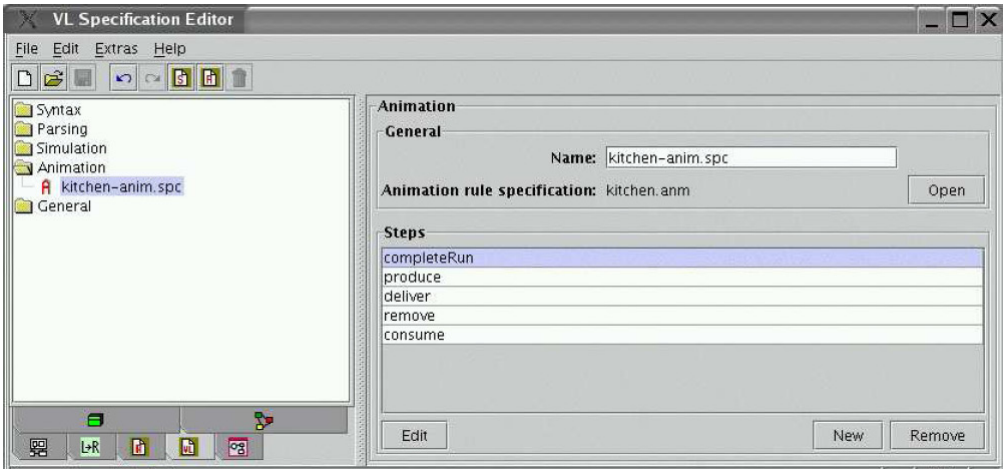


Figure 5.20: The VL Specification Editor for Animation Specification

After a VL alphabet has been defined in the Syntax component of the VL specification editor, a new animation step specification can be defined in the Animation component. The Animation Step Editor is evoked by pressing the New button in Fig. 5.20 (or the Edit button if the user wants to edit an existing animation step specification). In the animation step editor (shown in Fig. 5.21), animation steps are defined as expressions over animation rules in a way similar to the definition of simulation steps.

The Generated Animation Environment. The generated VL environment is extended by a generated animation component (animator). The animator offers an extra editor integrated in the generated VL environment in addition to the generated visual editor and the simulator. The GUI of the animator is shown in Fig. 5.22.

After a VL specification has been loaded in the VL environment, the user can switch to the animator by selecting the button Switch to animation mode. Here, animation steps are shown in the navigation tree in the left part of Fig. 5.22. Choosing the Open Diagram item

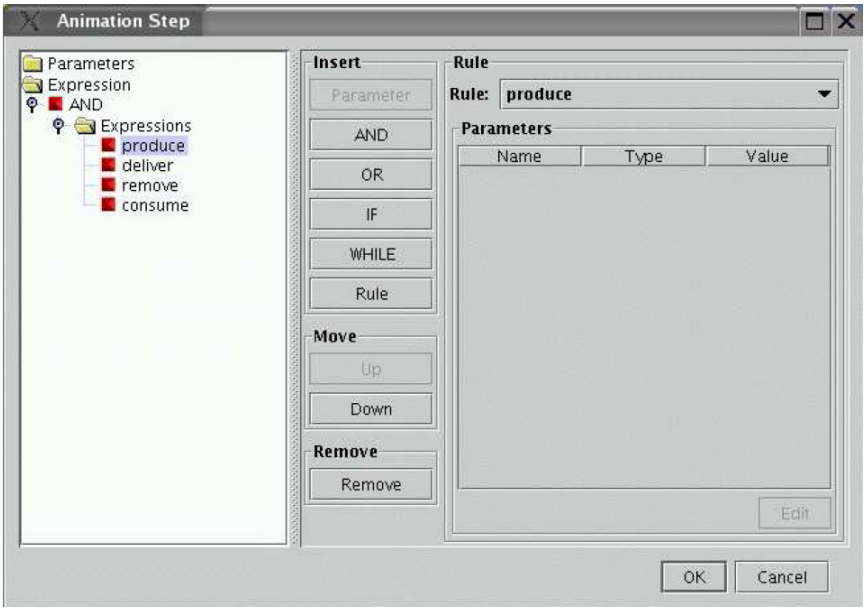


Figure 5.21: The Animation Step Editor

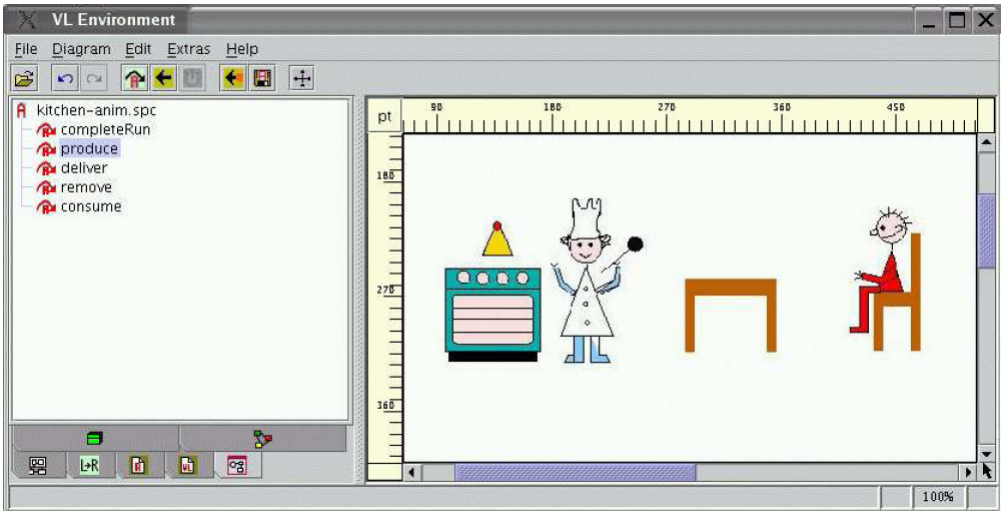


Figure 5.22: The Generated Animation Environment

from the File menu, a start diagram can be loaded which is visualized in the editor panel. To execute an animation scenario, an animation step is selected first. Then, the tool button Execute animation step is pressed. In the editor panel, the diagrams after each rule application are visualized, thus realizing a discrete simulation in the layout of the animation view. As the animator realizes an *off-line-animation*, the animation operations are not visualized in GENGED. They are evaluated in the background and stored in an SVG file, if the user presses the tool button Export animation step to SVG. The exported animation scenario including the continuous animation operations, can be viewed afterwards, by loading the

SVG file in an external SVG viewer.

By pressing the tool button Reset animation, the diagram in the editor panel is replaced by the original one, and by pressing the button Stop animation execution, the transformation is interrupted. The last button, Reset SVG animation protocol deletes the recorded animation steps, thus allowing to record a new animation scenario.

SVG Export. As existing animation tools and viewers in the last few years have been improving rapidly to satisfy a wide variety of growing applications and needs, we decided to profit from this variety and to implement an XML-based export from the GENGED animation environment to the standard language SVG instead of building our own GENGED-specific animation viewer tool.

SVG [WWW03] is a language for describing two-dimensional graphics and graphical applications in XML. The current version, SVG 1.1, is a W3C Recommendation, an open standard developed under the W3C Process. SVG has two parts: an XML-based file format (which we make use of) and a programming API for graphical applications (which we do not use). Key features include shapes, text and embedded raster graphics, with many different painting styles. SVG builds upon many other successful standards such as XML (SVG graphics are text-based and thus easy to create), JPEG, GIF and PNG for image formats, DOM for scripting and interactivity, SMIL for animation and CSS for styling.

SVG is used in many business areas including Web graphics, animation, user interfaces, graphics interchange, pretty printing, mobile applications and high-quality design.

SVG supports the animation of primitive graphic objects such as rectangle, ellipse, poly line, text, polygon and images in JPEG and GIF format. The Document Object Model (DOM) allows the definition of animation elements that attach dynamical behavior to graphical objects. The existing animation elements have been designed in cooperation with the W3C Synchronized Multimedia Working Group (SYMM), resulting in the *Synchronized Multimedia Integration Language (SMIL)* [Hos98], a part of which, called *SMIL Animation Specification* [SC00], specifies universal animation properties for XML based documents.

SVG offers four animation elements, which are also specified in the SMIL Animation Specification:

- *animate* allows to attach values to scalar attributes and properties over time. For example, the movement of an object is realized by changing the values of the object's X and Y coordinates over time.

- *set* is a simplified *animate* element allowing to change animation values on non-numerical attributes and properties. An example is the visibility of objects, which can be turned on or off.

- *animateMotion* moves an object along a path, which has to be specified according to the SVG path definition. Using path expressions, object could be moved for instance along a Bezier curve.
- *animateColor* modifies the color value of objects over time.

Fig. 5.23 shows an animation file in SVG format illustrating the use of the *animate* element where the attributes of a rectangle object are changed over time. The SVG file starts with a reference to the Document Type Definition (DTD) of the used SVG document type definition. The element svg has to attributes, width and height, which define the size of the display area. The attribute viewBox contains the area of user coordinates situated within the display area. The next element rect contains the rectangle to be animated, where the attributes x,y define its start position, width, height its start size, and fill the fill color. Starting from the time begin="0s" over a duration of dur="9s", the values of the rectangle attributes x,y, width and height are now changed in a linear way from their respective start values from to the end values to. In doing so, the rectangle changes its size, so that in the end it fills the complete visual display area (the viewBox). The attribute fill=freeze defines that the end values of the changed attributes are kept after the end of the animation.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w2.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg width="8cm" height="3cm" viewBox"0 0 800 300"
  xmlns="http://www.w3.org/2000/svg">
  <!-- The following illustrates the use of the 'animate' element
       to animate a rectangle x, y, and width attributes so that
       the rectangle grows to ultimately fill the viewport. -->
  <rect id="RectElement" x="300" y="100" width="300" height="100"
     fill="rgb(255,255,0)">
     <animate attributeName="x" attributeType"XML"
        begin="0s" dur="9s" fill="freeze" from="300" to="0"/>
     <animate attributeName="y" attributeType"XML"
        begin="0s" dur="9s" fill="freeze" from="100" to="0"/>
     <animate attributeName="width" attributeType"XML"
        begin="0s" dur="9s" fill="freeze" from="300" to="800"/>
     <animate attributeName="height" attributeType"XML"
        begin="0s" dur="9s" fill="freeze" from="100" to="300"/>
  </rect>
</svg>
```

Figure 5.23: Example of an SVG Animation File

In order to view the generated animations, a number of stand-alone SVG Viewers exist. These can be thought of as SVG-only browsers, in the same way as older browsers were HTML-only browsers. These SVG Viewers include an XML parser; a CSS parser; a CSS cascading, specificity and inheritance engine; and an SVG rendering engine to draw the graphics. They may offer print capabilities, in addition to display on screen.

A detailed overview of commercial and free SVG viewers is given by the official W3C list of implementations of SVG at [SVG04]. For our examples we used the SVG Toolkit

[SVG02], which is for free download according to a public license, and supports the animation of graphic objects. The source code is completely available in Java.

5.4.4 Workflow for the Definition of Animation Views and the Generation of Animation Scenarios in GENGED.

Fig. 5.24 presents a workflow using the GENGED environment (the architecture of which is shown in Fig. 5.18) for the generation of a scenario animation.

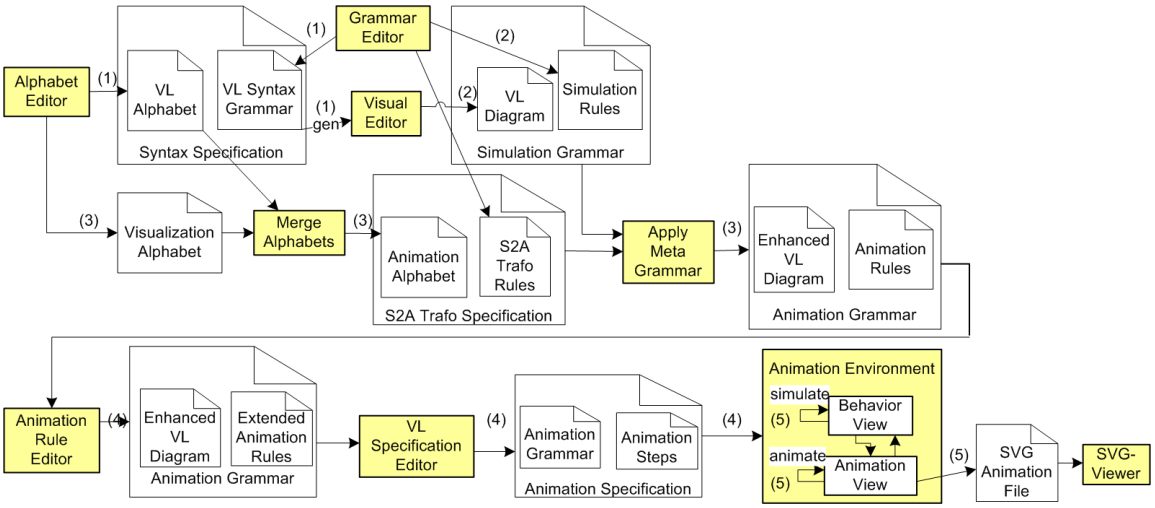


Figure 5.24: Defining Animation Views and Generating Animation Scenarios in GENGED

We explain Fig. 5.24 by adding to the workflow different roles for users of the extended GENGED environment (different roles need not necessarily be taken by different persons) and describing who is doing what. Note that the model of the system is built without necessarily taking into account the fact that we plan to later represent it in animation. Thus, the first two steps (1) and (2) where the visual language and the visual model are defined, use the components of the old GENGED environment.

(1) The *language designer* defines the Syntax Specification by using the Alphabet Editor to define the VL Alphabet and using the Grammar Editor to define the VL Syntax Grammar. Now a VL specific Visual Editor is generated allowing to edit VL diagrams.

(2) The *model designer* uses the generated Visual Editor to edit a VL Diagram (the start graph of the simulation grammar) and defines Simulation Rules using the Grammar Editor.

(3) The *view designer* specifies a Visualization Alphabet in the Alphabet Editor which represents the application domain, and integrates it with the VL Alphabet to an Animation Alphabet using the Merge Alphabets functionality of the Alphabet Editor. Now, the *view designer* defines the *S2A* Transformation Rules over the Animation Alphabet. The *S2A* Transformation Rules extend the parts of the model that should appear in the animation by their respective animation symbols. Applying the *S2A* Transformation Rules in the Grammar Editor to the Simulation Grammar (to the VL Diagram and to the Simulation Rules), the view designer generates an Animation Grammar, consisting of the Enhanced VL Diagram and of Animation Rules, both typed over the animation alphabet.

(4) The *animation designer* identifies the changes in the simulated system that would be best described by animation (e.g. the movement of some objects or changes of their attributes). He uses the visual Animation Rule Editor to produce Extended Animation Rules extending the Animation Rules by *animation operations* realizing continuous changes of graphics such as moving, or changing the size or the color. Additionally, the animation designer defines different animation steps by specifying a sequence of animation rule applications (scenarios) using the VL Specification Editor. In specifying such a sequence, all possible rule parameters (e.g. names of events etc.) must be given. The animation rules and steps are saved as Animation Specification, from which the Animation Environment can be generated.

(5) the *model validator* works in the generated Animation Environment. He or she can switch between the different views for one model. Thus, the model can be shown in the original layout of the behavior model, e.g. as AHL net, or in the animation view, where the model behavior is shown in the layout of the application domain. If more than one view is selected, then the simulation or animation steps are visualized in all selected views at once, e.g. it is possible to see a Petri net and its animation view together. The model validator simulates or animates the behavior of a Model by applying the previously defined Animation Steps to the current model state. The state transitions of the model are visualized as discrete steps in the GENGED animation environment. Continuous animation sequences can be generated and recorded. Initially, the start diagram (the initial state of the model) is saved as SVG file. Then the animation operations of each animation rule applied in the step are performed on the corresponding animated objects in the model diagram the animation rule is applied to, according to the match of the animation rule into the model. Each performed operation leads to the generation of the corresponding SVG animation elements (see Section 5.4.3), which are added to the SVG file. The resulting SVG file can be loaded into an external SVG viewer tool or SVG-capable browser, where the continuous animation can be viewed.

5.5 Related Work

In the last twenty years there has been much work on topics related to the ideas presented here. The first is *algorithm animation* (later also called *specification animation*). Various researchers developed this area into a notable sub-field of computer science. Algorithm animation began as a visual abstraction of program operation and its data and dynamics. The main motivation for the algorithm animation tools of the early 1990's was to find different ways of representing and abstracting the text-based formulation of the dynamics of algorithms with a visual environment that would provide an intuitive visualization of the logic behind the algorithm. Thus, tools like BALSA [Bro88], Tango [Sta90] and Polka [TS95] provide the user with the ability to construct a front-end and associate it with the behavior of the animated algorithm. Other tools enhance the power of a diagrammatic language to animation [Mey97]. Most approaches shared the view of the algorithm as a sequence of events occurring in time, and the most interesting events were chosen to be reflected in the animation. Much of the effort in building these tools was the development of viewers for the animation. The main focus of using such tools was associated with animation as a teaching tool for computer programming [HAS02, CDFP00].

Over the years, the developers of such tools were able to handle distributed technologies [CIFP02, BCLT96], and three-dimensional representation [SW93]. Other tools, which we will not review here, dealt with the complexity of software engineering and visualized the connection between different classes, different files of different behaviors at run-time (giving rise to the area of *software visualization* [SDBP98]).

The concepts behind these previous works lead naturally to our present work, namely the process of identifying important events at one level and showing them dynamically at another. Our implementation, presented in this chapter, is not another, more powerful tool that claims to make the connection between models and their animation quicker or easier. Rather, it provides a generic methodology and a specific implementation to use state-of-the art tools of one kind to view the animations (SVG-viewers and browsers), and, in a separate effort, to use formally-based specification tools to build and analyze simulation specifications (graph transformation based Meta-CASE tools according to the classification given in Section 5.1.2) and to define a formal, behavior-preserving extension towards the animation of system behavior. This methodology allows to be kept up to date on both kinds of tools. Graph transformation based tools are constantly evolving today and getting more and more powerful concerning formal model analysis, while animation tools and viewers have been improving rapidly to satisfy a wide variety of growing applications and needs. By dividing these tasks, and isolating the link between these two worlds, we are able to stay updated as both of them develop, resulting in powerful and detailed "inner parts" of the simulation specification that also "look nice" on the outside.

Chapter 6

Conclusion

We first summarize the main achievements of this thesis, and then outline some open problems and directions for future work.

Summary and Main Results

In this thesis we have presented a formal framework for simulation and animation of visual behavioral models, based on graph transformation. Basically, we have formalized a translation of visual models which conform to a visual language (VL) definition, into so-called *animation views*, where the model behavior is visualized in an adequate layout of the application domain (see Fig. 6.1 (a)).

The framework is based on the double-pushout approach to typed algebraic graph transformation [EEPT06]. The alphabet of the diagrammatic modeling VL is defined by type graph. For the animation view definition, this VL type graph is extended by elements for visualizing the application domain. A model-and-rule simulation-to-animation transformation ($S2A$ transformation) is defined to transform model-specific simulation specifications into animation specifications The typing relationships and specifications formalizing the relations in Fig. 6.1 (a) are shown in more detail in Fig. 6.1 (b).

The formal basis of graph transformation allows to reason about semantical correctness of simulation specifications, as well as the correctness of animation with respect to simulation.

The approach has been implemented by extending the GENGED environment for visual language definition and visual modeling [Gen, Bar02]. In the new GENGED animation environment, animation specifications may be enriched by continuous animation operations to observe the model behavior not only as discrete-event steps, but as a continuously animated change of the scene.

In contrast to related approaches and tools, e.g. for the animation of Petri net behav-

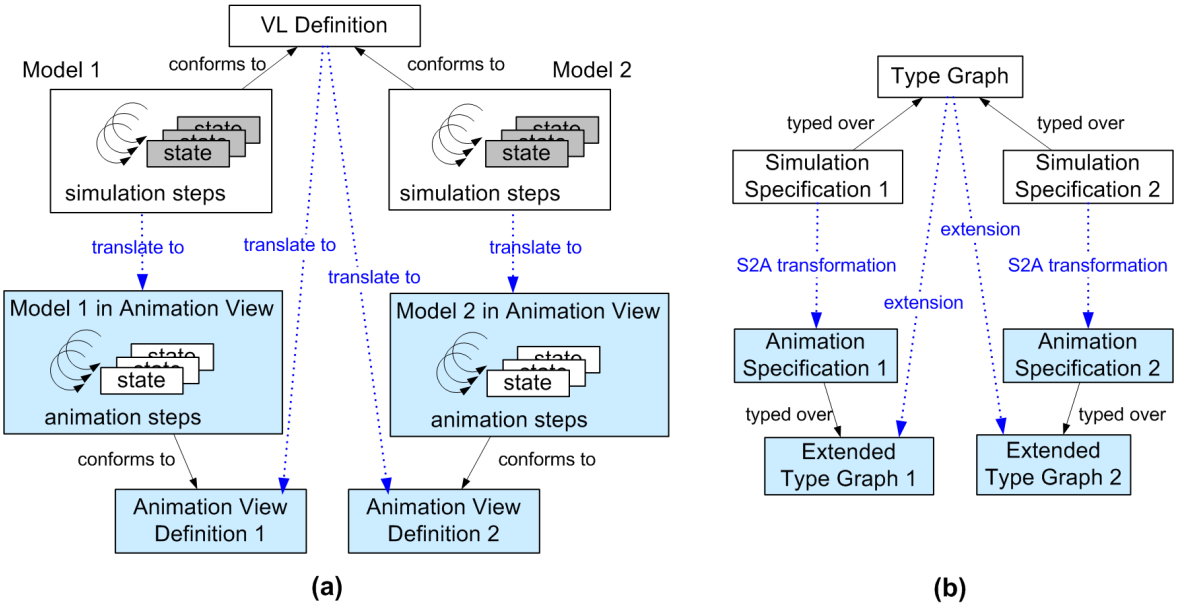


Figure 6.1: From Visual Models to Animation Views: Concepts (a) and Formalization by Typed Graph Transformation Systems (b)

ior [Gra99], the graph transformation framework offers a basis for a more general formalization of model behavior which is applicable to various visual behavior modeling languages.

In the following, we summarize the main results achieved in this thesis. Concerning the two conceptual chapters *Simulation* and *From Simulation to Animation*, we distinguish between general theoretical results and application-specific results achieved for specific visual modeling languages or specific applications.

Simulation

- *General theoretical results*:
  We formalized three different approaches to simulation based on graph transformation, namely the *interpreter semantics* (Def. 3.1.4), the *GTS-compiler semantics* (Def. 3.2.3), and the *amalgamation semantics* (Def. 3.3.5). In Theorem 3.4.2, we have shown the general semantical compatibility of universal simulation specifications defining interpreter or amalgamation semantics for visual languages, and their translation into model-specific simulation specifications defining GTS-compiler semantics for visual models.

- *Application-specific results*:
  For formal behavior modeling languages such as Petri nets, the semantical correctness of the corresponding simulation specification is crucial. We have shown the

semantical compatibility of the firing behavior of AHL nets and their compiler and amalgamation semantics (Theorem 3.5.10 and Theorem 3.5.18).

From Simulation to Animation

- *General theoretical results*:
  We have defined an *animation view* of a visual model by extending the alphabet of the underlying visual modeling language (Def. 4.2.2). Based on this definition, a model-and-rule simulation-to-animation transformation ($S2A$ transformation) was defined to transform model-specific simulation specifications into animation specifications (Def. 4.2.16). The termination of $S2A$ transformation was shown in Theorems 4.3.5 and 4.3.6. Criteria for the semantical correctness and completeness of $S2A$ transformation have been stated in Theorem 4.5.11 and Theorem 4.6.9, leading to the semantical equivalence of simulation and animation specifications in Theorem 4.6.15.

- *Application-specific results*:
  The criteria from Theorem 4.6.15 have been applied to show the semantical equivalence of simulation and animation specifications for the *Echo model* Petri net (Fact 4.6.16), the *Dining Philosophers* AHL net (Fact 4.7.4), the *Radio Clock* Statechart (Fact 4.7.8), and the *Drive-Through* UML model (Fact 4.7.12). Moreover, confluence of the respective $S2A$ transformations was proven for all sample applications (Facts 4.4.1, 4.7.3, 4.7.7, 4.7.11).

Tool Support

The visual language environment GENGED has been extended by the following components: A new *simulation environment* allows to specify simulation scenarios by defining units of controlled applications of simulation rules to be applied in the generated *simulator* to simulate the model behavior (Section 5.3). The alphabet editor has been extended to define new VL alphabets as *integration* of two or more existing VL alphabets (called *views*); In a VL environment which has been generated over such an integrated VL alphabet, it is now possible to *restrict* diagrams to one or more selected views (Section 5.4.1). The grammar editor has been extended to support *meta transformations* by defining and applying the rules of a so-called *meta grammar* to the rules and the start graph of an existing grammar (Section 5.4.2). A new *animation environment* allows to generate animation rules from simulation rules by $S2A$ transformation and to enrich animation rules in an *animation editor* by additional continuous animation operations. In analogy to the simulation environment, units of controlled applications of one or more animation rules can be defined, to be applied in the generated *animator* where the model behavior of is animated in the layout

of a selected animation view. Animation scenarios are exported into the XML-based SVG format [WWW03] (Section 5.4.3).

Open Problems and Future Work

Most of the presented results (including the implementation in GENGED) are available also for typed *attributed* graph transformation systems (TAGTS) [EEPT06]. In fact, the sample applications all use attributes for naming or numbering of symbols. Except of the retyping functor, which is defined and proven to be adjoint in the context of typed graph transformation systems only, and which is part of the proofs of semantical completeness, the concepts and results also hold for TAGTS.

Hence, one line of future work concerning the formal framework for simulation and animation presented in this thesis, is to make the theoretical results available for the more general framework of adhesive HLR categories [EEPT05].

Moreover, we have defined *animation views* on the basis of type graph extensions and restrictions. This approach fits well in the general notion of *views* (or viewpoints) in software modeling [FKN+92], where a view of a system is a projection of the system on one of its relevant aspects. Such a projection emphasizes certain aspects and ignores others. The UML [UML04a] therefore uses multiple visual languages that focus on different aspects, therefore exhibiting different *views* of a system (i.e. the structural, behavioral, data and interface view [BGH+98]). UML's many loosely connected kinds of diagrams make it so difficult or even impossible to define a rigorous semantics for full UML.

Hence, one line of future work is to give a formal definition not only for animation views, but for views in general, including the formalization of interaction, integration and consistency of views in the domain of typed, (attributed) graph transformation systems. First ideas of a view-based specification based on graph transformation are presented by Heckel [EEHT97, Hec98] using open graph transformation systems.

As mentioned in Section 5.2.2, VL alphabet specifications in GENGED are based on graph structure signatures which poses some restrictions on the way to define alphabets. Moreover, the concrete syntax is computed by the graphical constraint solver PARCON [Gri96] which has the disadvantages that on the one hand more complex diagrams lead to large constraint satisfaction problems and cause performance problems for layout computation; on the other hand, PARCON is available for Linux and Solaris only, so despite of the implementation of GENGED in Java, it cannot be run under the Windows platform. Furthermore, the generated environments are not meant to be integrated into other existing tool environments. As stand-alone applications they do not always offer the standard look-and-feel of modern modeling/editing features, like e.g. zooming or undo/redo.

To overcome these shortcomings, we recently decided to develop a new tool environment supporting the generation of visual environments, based on the one hand on recent MDA

development tools integrated in the development environment ECLIPSE [Ecl04], and on the other hand on typed attributed graph transformation to support syntax-directed editing, simulation and animation. ECLIPSE offers rich support for graphical editor development in form of a number of plug-ins (e.g. EMF [EMF06] and the Graphical Editor Framework GEF [GEF06]) . The new tool environment TIGER [EEHT05, EEHT04] (Transformation-based generation of modeling environments) combines the advantages of precise VL specification techniques (offered by AGG [Tae04]) with sophisticated graphical editor and layout features (offered by GEF). Graph transformation is used on the abstract syntax level to realize the editing operations. TIGER extends the AGG engine by a concrete visual syntax definition. From the VL definition, the TIGER *generator* generates Java [Jav04] source code of an ECLIPSE visual editor plug-in which makes use of a variety of GEF's predefined editor functionalities. Hence, the generated editor plug-in appears in a timely fashion and the generated editor code may easily be extended by further functionalities.

Instead of using attributed graph structures as VL diagrams as done in GENGED, typed attributed graphs are used in TIGER, which comprise a simpler and more compact approach to VL definition. TIGER uses the default GEF graph layouter to compute the layout of the symbols and links in the generated editor. Up to now, TIGER can handle only graph-like languages [Min02], such as Petri nets or class diagrams, and does not yet need a graphical constraint solver.

Future work is planned to extend the TIGER environment to handle more complex visual modeling languages (e.g. allowing nesting of elements), and to extend the generated environment by simulation and animation components, based on the approach presented in this thesis. An open tool environment based on ECLIPSE would also be available for more potential users than GENGED, and be a basis to extend the use of animation views to more and bigger case studies using diverse modeling techniques.

Establishing a connection between TIGER and existing domain-specific graphic *libraries* for animation (e.g. the animation library for biology [Lab06]) would also further assist the user in defining adequate animation views.

Moreover, the animation rules provide a good basis to allow *user interaction* with the animations by directly manipulating elements in the animation view when defining animation scenarios (i.e. controlling the order, the matches and the frequency of animation rule applications). Here lies the central advantage of coding the animation information into the rules instead of translating directly the simulation steps into the animation layout (as realized e.g. in [HEC03]).

Further tool support would be helpful for the choice of relevant (critical) scenarios for simulation and animation. This would require to future work combining research directions from the area of *test case generation* and model-based testing (e.g. [Mue05, BJK+05, Gra94]) with the field of graph transformation.

Appendix A

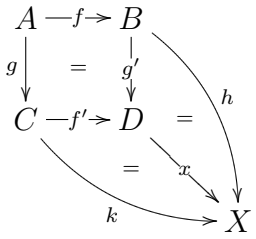
Pushouts and Pullbacks

The idea of a pushout is that of an object, that emerges from gluing two objects along a common subobject.

Definition A.1 (*Pushout*)
Given morphisms $f : A \to B$ and $g : A \to C$ in a category $\mathbf{C}$. A *pushout* (D, f', g') over f and g is defined by

- a pushout object D and
- morphisms $f' : C \to D$ and $g' : B \to D$ with $f' \circ g = g' \circ f$

such that the following universal property is fulfilled:
For all objects X and morphisms $h : B \to X$ and $k : C \to X$ with $k \circ g = h \circ f$ there is a unique morphism $x : D \to X$ such that $x \circ g' = h$ and $x \circ f' = k$.



△

Property A.2 (*Pushouts in* $\mathbf{Sets}$, $\mathbf{Graphs}$, $\mathbf{Graphs_{TG}}$)
In $\mathbf{Sets}$, the pushout object over morphisms $f : A \to B$ and $g : A \to C$ can be constructed as the quotient $B \dot{\cup} C|_{\equiv}$, where $\equiv$ is the smallest equivalence relation with $(f(a), g(a)) \in \equiv$ for all $a \in A$. The morphisms f' and g' are defined by $f'(c) = [c]$ for all $c \in C$ and $g'(b) = [b]$ for all $b \in B$.

Moreover we have the following properties:

1. If f is injective (surjective), then also f' is injective (surjective).
2. The pair (f', g') is jointly surjective, i.e. for each $x \in D$ there is a preimage $b \in B$ with $g'(b) = x$ or $c \in C$ with $f'(c) = x$.
3. If $x \in D$ has preimages $b \in B$ and $c \in C$ with $g'(b) = f'(c) = x$ and f is injective then there is a unique preimage $a \in A$ with $f(a) = b$ and $g(a) = c$.
4. The pushout object D is unique up to isomorphism.

In $\mathbf{Graphs}$ and $\mathbf{Graphs_{TG}}$, pushouts can be constructed componentwise in $\mathbf{Sets}$. The mappings for the nodes are uniquely determined by those of the edges. Moreover the properties 1.-4. hold componentwise. △

Proof: see [EEPT06]. □

More properties concerning the composition of pushouts are stated in Property A.7, 1 and 2.

For various situations we need a reverse construction of a pushout. This is called the pushout complement.

Definition A.3 (*Pushout Complement*)

Given morphisms $f : A \to B$ and $n : B \to D$, then $A \xrightarrow{m} C \xrightarrow{g} D$ is the *pushout complement* of f and n, if (1) is a pushout.

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ {\scriptstyle m}\downarrow & (1) & \downarrow{\scriptstyle n} \\ C & \xrightarrow{g} & D \end{array}$$

△

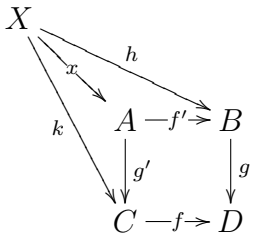
The dual construction for a pushout is that of a pullback. Pullbacks can be seen as a generalized intersection of objects over a common object.

Definition A.4 (*Pullback*)
Given morphisms $f : C \to D$ and $g : B \to D$. A *pullback* (A, f', g') over f and g is defined by

- a pullback object A and
- morphisms $f' : A \to B$ and $g' : A \to C$ with $g \circ f' = f \circ g'$

such that the following universal property is fulfilled:
For all objects X with morphisms $h : X \to B$ and $k : X \to C$ with $f \circ k = g \circ h$ there is

a unique morphism $x : X \to A$ such that $f' \circ x = h$ and $g' \circ x = k$.



△

Property A.5 (*Pullbacks in* $\mathbf{Sets}$, $\mathbf{Graphs}$, *and* $\mathbf{Graphs_{TG}}$)
In $\mathbf{Sets}$, the pullback $C \xleftarrow{g'} A \xrightarrow{f'} B$ over the morphisms $f : C \to D$ and $g : B \to D$ is constructed by $A = \bigcup_{d \in D} f^{-1}(d) \times g^{-1}(d) = \{(c, b) \mid f(c) = g(b)\} \subseteq C \times B$ with morphisms $f' : A \to B : (c, b) \mapsto b$ and $g' : A \to C : (c, b) \mapsto c$. Moreover, we have the following properties:

1. If f is injective (or surjective), then f' is also injective (or surjective, respectively).
2. f' and g' are jointly injective, i.e. for all $a_1, a_2 \in A$, $f'(a_1) = f'(a_2)$ and $g'(a_1) = g'(a_2)$ implies $a_1 = a_2$.
3. A commutative square, as given in Definition A.4, is a pullback in $\mathbf{Sets}$ iff, for all $b \in B$, $c \in C$ with $g(b) = f(c)$, there is a unique $a \in A$ with $f'(a) = b$ and $g'(a) = c$.

In $\mathbf{Graphs}$ and $\mathbf{Graphs_{TG}}$, pullbacks can be constructed componentwise in $\mathbf{Sets}$. The mappings for the nodes are uniquely determined by those of the edges. △

More properties concerning the composition of pullbacks are stated in Property A.7, 3 and 4.

We will state the following pushout / pullback properties without proofs (for proofs, see e.g. [EEPT06, EHPP04]).

Property A.6 (*Trivial Pushouts and Pullbacks*)

1. For arbitrary morphisms $k : A \to B$ the diagram (1) is a pushout and also a pullback.
2. The diagram (2) is a pullback iff $m : A \to B$ is monomorphism,

3. Pushouts along M-morphisms are also pullbacks. A pushout (3) is called "pushout along M-morphisms" if m is a monomorphism.

$$\begin{array}{ccc} A & \xrightarrow{k} & B \\ {\scriptstyle id}\downarrow & (1) & \downarrow{\scriptstyle id} \\ A & \xrightarrow{k} & B \end{array} \qquad \begin{array}{ccc} A & \xrightarrow{id} & A \\ {\scriptstyle id}\downarrow & (2) & \downarrow{\scriptstyle m} \\ A & \xrightarrow{m} & B \end{array} \qquad \begin{array}{ccc} A & \xrightarrow{f} & C \\ {\scriptstyle m}\downarrow & (3) & \downarrow{\scriptstyle n} \\ B & \xrightarrow{g} & D \end{array}$$

△

Property A.7 (*Pushout/Pullback Composition and Decomposition*)

Given the commutative diagram to the right (where properties 5 and 6 additionally require that l and w are monomorphisms).

$$\begin{array}{ccccc} A & \longrightarrow & B & \longrightarrow & E \\ {\scriptstyle l}\downarrow & (1) & \downarrow & (2) & \downarrow \\ C & \longrightarrow & D & \xrightarrow[w]{} & F \end{array}$$

Then we have

1. If the square (1) and (2) are pushouts, the outer square $(1+2)$ is pushout as well.
2. If square (1) and square $(1+2)$ are pushouts, then square (2) is pushout as well.
3. If squares (1) and (2) are pullbacks, square $(1+2)$ is pullback as well.
4. If square (2) and square $(1+2)$ are pullbacks, then square (1) is pullback as well.
5. If square (1) is pushout and square $(1+2)$ is pullback, then squares (1) and (2) are pullbacks.
6. If square (2) is pullback and square $(1+2)$ is pushout, then squares (1) and (2) are pushouts and also pullbacks.

△

Property A.8 (*Special Pushout/Pullback Composition and Decomposition*)

Given the commutative diagram to the right with square (1) pushout and sqare $(1+2)$ pullback, and f_1, f_2, f_3 monomorphisms. Then, square (1) and (2) are pullbacks.

$$\begin{array}{ccccc} A_1 & \xrightarrow{g_1} & A_2 & \xrightarrow{g_2} & A_3 \\ {\scriptstyle f_1}\downarrow & (1) & \downarrow{\scriptstyle f_2} & (2) & \downarrow{\scriptstyle f_3} \\ B_1 & \xrightarrow[h_1]{} & B_2 & \xrightarrow[h_2]{} & B_3 \end{array}$$

△

Proof: It suffices to show that (2) is pullback, because $(1+2)$ and (2) pullbacks, implies that (1) is pullback as well (Property A.7, 4). Since f_2 is monomorphism, ist suffices to show for $b_2 \in B_2, a_3 \in A_3$ with $h_2(b_2) = b_3 = f_3(a_3)$ the existence of $a_2 \in A_2$ with $f_2(a_2) = b_2$ and $g_2(a_2) = a_3$.

Given a_3, b_2 and b_3 as above, square (1) pushout implies that either

(*Case 1*) $\exists a_2 \in A_2 : f_2(a_2) = b_2$, or

(*Case 2*) $\exists b_1 \in B_1 : h_1(b_1) = b_2$.

In *Case 1* we have $f_3 \circ g_2(a_2) = h_2 \circ f_2(a_2) = h_2(b_2) = b_3$ and $f_3(a_3) = b_3$, and f_3 monomorphism implies that $g_2(a_2) = a_3$.

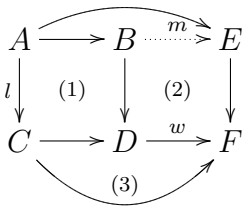
In *Case 2*, square $(1+2)$ pullback implies $\exists a_1 \in A_1 : f_1(a_1) = b_1$ and $g_2 \circ g_1(a_1) = a_3$. Let $a_2 = g_1(a_1) \in A_2$, then $f_2(a_2) = f_2 \circ g_1(a_1) = h_1 \circ f_1(a_1) = h_1(b_1) = b_2$. Now, $f_3 \circ g_2(a_2) = h_2 \circ f_2(a_2) = h_2(b_2) = f_3(a_3)$. Now f_3 monomorphism implies that $g_2(a_2) = a_3$. □

Property A.9 (*Uniqueness of Pushout Complements*)
Pushout complements of M-morphisms (if they exist) are unique up to isomorphism: Let $A \xrightarrow{m} B$ and $C \xrightarrow{g} D$ be two morphisms as shown in Diagram (1) with $m \in M$. Then there exists a unique graph C and unique morphisms $A \xrightarrow{f} C$ and $C \xrightarrow{n} D$ such that square (2) is a pushout.

△

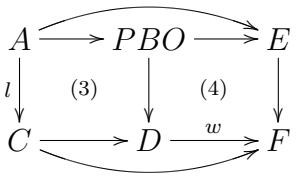
Property A.10 (*Missing Pushout Morphism 1*)

If in the diagram to the right, morphisms l and w are monomorphisms, and square (1) and the surrounding square are pushouts, then there exists an injective morphism $m : B \to E$ (the dotted arrow), such that square (2) is pushout and pullback, and (3) commutes.



△

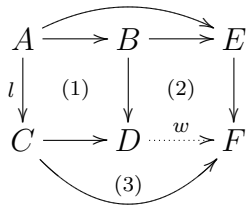
Proof:
We first construct the pullback (4) in the diagram to the right. Due to the surrounding square being a pushout, we get a unique morphism $A \to PBO$ by the universal pullback property (Def. A.4). Now, square (4) is a pullback, and the surrounding square is a pushout. By pushout-pullback decomposition (Property A.7, 6), we get square (3) is a pushout and pullback. This pushout differs from pushout (1) in the diagram above only in the pushout complement.



As pushout complements are unique (Property A.9), we conclude that $PBO \cong B$. By pushout decomposition (Property A.7), we conclude that square (2) is a pushout. Moreover, the morphism $B \to E$ is injective, because (2) is a pushout and w is injective (see Property A.2, 1). □

Property A.11 (*Missing Pushout Morphism 2*)

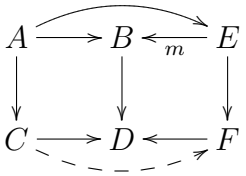
If in the diagram to the right, square (1) and the surrounding square are pushouts, then there exists a morphism $w : D \to F$ (the dotted arrow), such that square (2) is a pushout and (3) commutes.



△

Property A.12 (*Special Pushout Decomposition*)

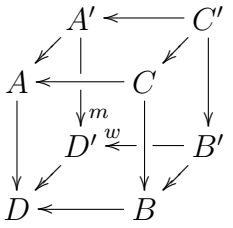
If in the diagram to the right squares (1) and (2) are both pushouts with $m \in M$, and there exists a morphism $A \to E$, then there exists a morphism $C \to F$ such that the diagram commutes and the outer square is a pushout.



△

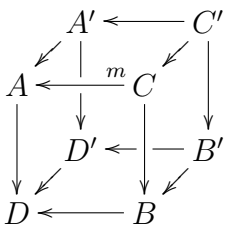
Property A.13 (*Cube Pushout-Pullback Property*)

Given a commutative cube as shown in the diagram to the right, where the back square is pullback with $m : A' \to D' \in M$ and $w : B' \to D' \in M$, and the left square and the bottom square are pushouts. Then, the front square is pullback $\Leftrightarrow$ the right square and the top square are pushouts.



△

Property A.14 (*Van Kampen Property*)
Given a commutative cube as shown in the diagram to the right, with $m : C \to A \in M$, where the front square is a pushout and the right and top faces are pullbacks. Then, we have: the back face is pushout $\Leftrightarrow$ the left and the bottom squares are pullbacks.



△

Bibliography

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

[AK02] D. Akehurst and S. Kent. A relational approach to defining transformations in a metamodel. In *Proc. 5th Intern. Conf. on the Unified Modeling Language (UML 2002)*, volume 2460 of *LNCS*, pages 243 – 258. Springer, 9 2002.

[Alt05] Altia. *Embedded Systems Graphics Software*, 2005. http://www.altia.com.

[Bar97] L. Baresi. *Formal Customization of Graphical Notations*. PhD thesis, Politecnico di Milano, 1997. in Italian.

[Bar00] R. Bardohl. GENGED – *Visual Definition of Visual Languages based on Algebraic Graph Transformation*. Verlag Dr. Kovac, 2000. PhD thesis, Technical University of Berlin, Dept. of Computer Science, 1999.

[Bar02] R. Bardohl. A Visual Environment for Visual Languages. *Science of Computer Programming (SCP)*, 44(2):181–203, 2002.

[BCLT96] J. E. Baker, I. F. Cruz, G. Liotta, and R. Tamassia. Algorithm animation over the World Wide Web. In *Proc. Workshop on Advanced Visual Interfaces (AVI'96)*, pages 203 –212, 1996.

[BE01] R. Bardohl and C. Ermel. Visual Specification and Parsing of a Statechart Variant using GENGED. In *Statechart Modeling Contest at IEEE Symposium on Visual Languages and Formal Methods (VLFM'01)*, Stresa, Italy, September 5–7 2001.

[BE03] R. Bardohl and C. Ermel. Scenario Animation for Visual Behavior Models: A Generic Approach Applied to Petri Nets. In G. Juhas and J. Desel, editors, *Proc. 10th Workshop on Algorithms and Tools for Petri Nets (AWPN'03)*. GI Special Interest Group on Petri Nets and Related System Models, 2003.

[BEdLT04] R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer. Integrating Meta Modelling with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In M. Wermelinger and T. Margaria-Steffens, editors, *Proc. Fundamental Aspects of Software Engineering (FASE)'04*, volume 2984 of *LNCS*. Springer, 2004.

[BEE01] R. Bardohl, C. Ermel, and H. Ehrig. Generic Description of Syntax, Behavior and Animation of Visual Models. Technical Report 2001/19, Technische Universität Berlin, 2001.

[BEE+02] R. Bardohl, K. Ehrig, C. Ermel, A. Qemali, and I. Weinhold. GENGED – Specifying Visual Environments based on Visual Languages. In H.-J. Kreowski, editor, *Proc. of APPLIGRAPH Workshop on Applied Graph Transformation (AGT 2002)*, pages 71–82, 2002.

[BEP02a] R. Bardohl, C. Ermel, and J. Padberg. Formal Relationship between Petri Nets and Graph Grammars as Basis for Animation Views in GenGED. In *Proc. IDPT 2002: Sixth World Conference on Integrated Design and Process Technology*. Society for Design and Process Science (SDPS), 2002.

[BEP02b] R. Bardohl, C. Ermel, and J. Padberg. Transforming Specification Architectures by GENGED. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. First Int. Conf. on Graph Transformation (ICGT'02)*, volume 2505 of *LNCS*, pages 30–44. Springer, 2002.

[BEW02a] R. Bardohl, C. Ermel, and I. Weinhold. AGG and GENGED: Graph Transformation-Based Analysis Techniques for Efficient Visual Language Validation. In T. Mens, A. Schürr, and G. Taentzer, editors, *Proc. Graph Transformation-Based Tools (GraBaTs'02), Satellite Event of ICGT'02*, pages 120–130, 2002.

[BEW02b] R. Bardohl, C. Ermel, and I. Weinhold. Specification and Analysis Techniques for Visual Languages with GENGED. Technical Report 2002–13, Technical University Berlin, Dept. of Computer Science, September 2002.

[BEW03] R. Bardohl, C. Ermel, and I. Weinhold. GenGED - A Visual Definition Tool for Visual Modeling Environments. In J. Pfaltz and M. Nagl, editors, *Proc. Application of Graph Transformations with Industrial Relevance (AGTIVE'03)*, Charlottesville/Virgina, USA, September 2003.

[BGH+98] R. Breu, R. Grosu, F. Huber, B. Rumpe, and W. Schwerin. Systems, Views and Models of UML. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language, Technical Aspects and Applications*. Physica Verlag, Heidelberg, 1998.

[BGH+05] S. Burmester, H. Giese, M. Hirsch, D. Schilling, and M. Tichy. The Fujaba real-time tool suite: Model-driven development of safety-critical, real-time systems. In *Proc. 27th Intern. Conf. on Software Engineering (ICSE), St. Louis, Missouri, USA*, May 2005.

[BJK+05] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[BK06] B. Braatz and A. R. Kniep. Integration of Object-Oriented Modelling Techniques. Internal Report, TU Berlin, 2006.

[BNS00] R. Bardohl, M. Niemann, and M. Schwarze. GENGED – A Development Environment for Visual Languages. In *Int. Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE'99), LNCS 1779*, pages 233–240. Springer, 2000.

[Bot03] P. Bottoni. Dynamic aspects of visual modelling languages. In R. Bardohl and H. Ehrig, editors, *Proc. Uniform Approaches to Graphical Specification Techniques (UniGra 2003), Warsaw, Poland*, volume 82/7, pages 131–145. Elsevier Science Direct, 2003.

[BP02] L. Baresi and M. Pezze. A Toolbox for Automating Visual Software Engineering. In R. Kutsche and H. Weber, editors, *Proc. Fundamental Approaches to Software Engineering (FASE'02), Grenoble, April 2002*, pages 189 – 202. Springer LNCS 2306, 2002.

[BRJ98] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[Bro88] M. H. Brown. Exploring Algorithms using BALSA II. *IEEE Computer*, 21(15), 1988.

[BST00] P. Bottoni, A. Schürr, and G. Taentzer. Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation. In *Proc. IEEE Symposium on Visual Languages*, September 2000.

[BTMS99] R. Bardohl, G. Taentzer, M. Minas, and A. Schürr. Application of Graph Transformation to Visual Languages. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[CC91] G. Costagliola and S.-K. Chang. Parsing 2-D languages with positional grammars. In *Proc. of 1991 International Workshop on Parsing Technologies (IWPT'91)*, pages 218–224, Cancun, Mexico, 1991. Association for Computational Linguistics.

[CDFP00] P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible Execution and Visualization of Programs with Leonardo. *Journal of Visual Languages and Computing*, 11(2):125–150, 2000.

[CDLOT98] G. Costagliola, A. De Lucia, S. Orefice, and G. Totora. Positional Grammars: A Formalism for LR-Like Parsing of Visual Languages. In K. Marriott and B. Meyer, editors, *Visual Language Theory*, pages 171–192. Springer, 1998.

[CEL+96] A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and J. Padberg. The category of typed graph grammars and their adjunction with categories of derivations. In *5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg '94, LNCS 1073*, pages 56–74, 1996.

[CHK04] B. Cordes, K. Hölscher, and H.-J. Kreowski. UML interaction diagrams: Correct translation of sequence diagrams into collaboration diagrams. In M. Nagl, J. Pfaltz, and B. Böhlen, editors, *AGTIVE'03 Proceedings*, volume 3062 of *Lecture Notes in Computer Science*, pages 275–291. Springer, 2004.

[CHM00] A. Corradini, R. Heckel, and U. Montanari. Graphical Operational Semantics. In *Proc. ICALP 2000 Workshop on Graph Transformation and Visual Modelling Techniques*. Carleton Scientific, 2000.

[CHM+02] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA: Visual automated transformations for formal verification and validation of UML models. In J. Richardson, W. Emmerich, and D. Wile, editors, *Proc. 17th IEEE Intern. Conf. on Automated Software Engineering (ASE'02)*, pages 267–270, Edinburgh, UK, September 23–27 2002. IEEE Press.

[CIFP02] G. Cattaneo, G.F. Italiano, and V. Ferraro-Petrillo. CATAI: Concurrent Algorithms and Data Types Animation over the Internet. *Journal of Visual Languages and Computing*, 13(4):391–419, August 2002.

[CM95] A. Corradini and U. Montanari. Specification of Concurrent Systems: From Petri Nets to Graph Grammars. In G. Hommel, editor, *Proc. Workshop on Quality of Communication-Based Systems, Berlin, Germany*. Kluwer Academic Publishers, 1995.

[CPN05] CPN Group, University of Aarhus, Denmark. *CPN Tools: Computer Tool for Coloured Petri Nets*, 2005. http://wiki.daimi.au.dk/cpntools/cpntools.wiki.

[DD04] C-Latitude Ltd. Web Site Design and Application Development. Glossary of technical terms, 2004. http://www.c-latitude.com/.

[Die01] S. Diehl. *Software Visualization*, volume 2269 of *LNCS*. Springer Verlag, 2001.

[Dir04] Macromedia, Inc. *Macromedia Director MX 2004*, 2004. http://www.macromedia.com/software/director/.

[dL03] J. de Lara. Meta-Modelling and Graph Transformation for the Simulation of Systems. *EATCS Bulletin No. 81*, October 2003.

[dLETE04] J. de Lara, C. Ermel, G. Taentzer, and K. Ehrig. Parallel Graph Transformation for Model Simulation applied to Timed Transition Petri Nets. In *Proc. Graph Transformation and Visual Modelling Techniques (GTVMT) 2004*, 2004.

[dLV02] J. de Lara and H. Vangheluwe. Computer Aided Multi-Paradigm Modelling to Process Petri-Nets and Statecharts. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. 1st Int. Conf. on Graph Transformation (ICGT 2002)*, volume 2505 of *LNCS*, pages 239–253. Springer, 2002.

[dLVA04] J. de Lara, H. Vangheluwe, and M. Alfonseca. Meta-Modelling and Graph Grammars for Multi-Paradigm Modelling in AToM^3. *Software and System Modeling: Special Section on Graph Transformations and Visual Modeling Techniques*, 3(3):194–209, 2004.

[DÜ95] T.B. Dinesh and S. M. Üsküdarlı. Visual Object Definition Language. In *Proceedings of the fifth Eurographics Workshop on Programming Paradigms in Graphics*, pages 109–124, 1995.

[EB02] C. Ermel and R. Bardohl. Scenario Views for Visual Behavior Models in GENGED. In *Proc. Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'02), Satellite Event of ICGT'02*, pages 71–83, Barcelona, Spain, October 2002.

[EB04] C. Ermel and R. Bardohl. Scenario Animation for Visual Behavior Models: A Generic Approach. *Software and System Modeling: Special Section on Graph Transformations and Visual Modeling Techniques*, 3(2):164–177, 2004.

[EBE03] C. Ermel, R. Bardohl, and H. Ehrig. Generation of Animation Views for Petri Nets in GENGED. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Advances in Petri Nets: Petri Net Technology for Communication Based Systems*, volume 2472 of *LNCS*. Springer, 2003.

[EBP01] C. Ermel, R. Bardohl, and J. Padberg. Visual Design of Software Architecture and Evolution based on Graph Transformation. In C. Ermel H. Ehrig and J. Padberg, editors, *Proc. Workshop on Uniform Approaches to Graphical Process Specification Techniques (UNIGRA'01)*, volume 44 (4) of *ENTCS*, Genova, Italy, March 31st – April 1st 2001. Elsevier Science Publishers.

[Ecl04] Eclipse Consortium. *Eclipse – Version 2.1.3*, 2004. http://www.eclipse.org.

[EE05a] H. Ehrig and K. Ehrig. Overview of Formal Concepts for Model Transformations based on Typed Attributed Graph Transformation. In *Proc. International Workshop on Graph and Model Transformation (GraMoT'05)*, ENTCS, Tallinn, Estonia, September 2005. Elsevier Science.

[EE05b] C. Ermel and K. Ehrig. View transformation in visual environments applied to Petri nets. In G. Rozenberg, H. Ehrig, and J. Padberg, editors, *Proc. Workshop on Petri Nets and Graph Transformation (PNGT), Satellite Event of ICGT'04*, volume 127(2) of *ENTCS*, pages 61–86. Elsevier Science, 2005.

[EEdL$^+$05] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. Termination criteria for model transformation. In M. Wermelinger and T. Margaria-Steffen, editors, *Proc. Fundamental Approaches to Software Engineering (FASE)*, volume 2984 of *LNCS*, pages 214–228. Springer, 2005.

[EEEP06] H. Ehrig, K. Ehrig, C. Ermel, and J. Padberg. Construction and Correctness Analysis of a Model Transformation from Activity Diagrams to Petri Nets. In I. Troch and F. Breitenecker, editors, *Proc. Intern. IMCAS Symposium on Mathematical Modelling (MathMod)*. ARGESIM-Reports, 2006.

[EEHT97] G. Engels, H. Ehrig, R. Heckel, and G. Taentzer. A combined reference model- and view-based approach to system specification. *Int. Journal of Software and Knowledge Engineering*, 7(4):457–477, 1997.

[EEHT04] K. Ehrig, C. Ermel, S. Hänsgen, and G. Taentzer. Towards Graph Transformation based Generation of Visual Editors using Eclipse. In M. Minas, editor, *Visual Languages and Formal Methods*, volume 127/4 of *ENTCS*, pages 127–143. Elsevier Science, 2004.

[EEHT05] K. Ehrig, C. Ermel, S. Hänsgen, and G. Taentzer. Generation of visual editors as eclipse plug-ins. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, IEEE Computer Society, Long Beach, California, USA, 2005.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[EEPT05] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Fundamental Theory for Typed Attributed Graphs and Graph Transformation based on Adhesive HLR Categories. *Fundamenta Informaticae*, XX:1–31, 2005.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer, 2006.

[EET06] C. Ermel, H. Ehrig, and G. Taentzer. Simulation and Animation of Visual Models of Embedded Systems: A Graph-Transformation-Based Approach Applied to Petri Nets. In G. Hommel, and S. Huanye, editors, *Proc. Workshop on Embedded Systems: Modeling, Technology and Applications*, Berlin, Germany, 2006. Springer.

[EGHK02] G. Engels, L. Goenewegen, R. Heckel, and J. M. Küster. Consistency-preserving Model Evolution through Transformations. In J.-M. Jezequel, H. Hussmann, and S. Cook, editors, *Proc. UML 2002*, volume 2460 of *LNCS*, Dresden, Germany, 2002. Springer.

[EHC05] G. Engels, R. Heckel, and A. Cherchago. Flexible Interconnection of Graph Transformation Modules: A Systematic Approach. In H.-J. Kreowski, U. Montanari, F. Orejas, G. Rozenberg, and G. Taentzer, editors, *Formal Methods in Software and Systems Modeling: Essays Dedicated to Hartmut Ehrig on the Occasion of His 60th Birthday*, volume 3393 of *LNCS*. Springer, 2005.

[EHHS00] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioural diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language*, volume 1939 of *LNCS*, pages 323 – 337. Springer, 2000.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation II: Single Pushout Approach and Comparison with Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, chapter 4, pages 247–312. World Scientific, 1997.

[EHKP91] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. Parallelism and concurrency in high-level replacement systems. *Math. Struct. in Comp. Science*, 1:361–404, 1991.

[EHKZ05] C. Ermel, K. Hölscher, S. Kuske, and P. Ziemann. Animated Simulation of Integrated UML Behavioral Models based on Graph Transformation. In M. Erwig and A. Schürr, editors, *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, Dallas, Texas, USA, September 2005. IEEE Computer Society.

[EHPP04] H. Ehrig, A. Habel, J. Padberg, and U. Prange. Adhesive high-level replacement categories and systems. In F. Parisi-Presicce, P. Bottoni, and G. Engels, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04)*, volume 3256 of *LNCS*, pages 144–160, Rome, Italy, October 2004. Springer.

[Ehr79] H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars (A Survey). In *Graph Grammars and their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1–69. Springer, 1979.

[Ehr03] K. Ehrig. Konzeption und Implementierung eines Generators für Animationsumgebungen für visuelle Modellierungssprachen. Technical Report 2003-17, TUB, 2003.

[Ehr04a] H. Ehrig. Attributed Graphs and Typing: Relationship between Different Representations. *EATCS Bulletin, vol 82*, pages 175–190, 2004.

[Ehr04b] H. Ehrig. Behaviour and Instantiation of High-Level Petri Net Processes. *Fundamenta Informaticae*, 64:1–37, 2004.

[EK76] H. Ehrig and H.-J. Kreowski. Parallel graph grammars. In A. Lindenmayer and G. Rozenberg, editors, *Automata, Languages, Development*, pages 425–447. Amsterdam: North Holland, 1976.

[EKGH01] G. Engels, J. M. Küster, L. Groenewegen, and R. Heckel. A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models. In V. Gruhn, editor, *Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, pages 327–343, Vienna, Austria, September 2001. ACM Press.

[EKMR99] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol 3: Concurrency, Parallelism and Distribution*. World Scientific, 1999.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, Berlin, 1985.

[EMC+98] H. Ehrig, B. Mahr, F. Cornelius, M. Grosse-Rhode, and P. Zeitz. *Mathematisch Strukturelle Grundlagen der Informatik*. Springer Verlag, 1998.

[EMF06] Eclipse Consortium. *Eclipse Modeling Framework (EMF) – Version 2.2.0*, 2006. http://www.eclipse.org/emf.

[EPT04] H. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graph transformation. In F. Parisi-Presicce, P. Bottoni, and G. Engels, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04), Rome, Italy*, volume 3256 of *LNCS*. Springer, 2004.

[ER76] H. Ehrig and B.K. Rosen. Commutativity of independent transformations on complex objects. Research Report RC 6251, IBM T. J. Watson Research Center, Yorktown Heights, 1976.

[Erw98] M. Erwig. Abstract Syntax and Semantics of Visual Languages. *Journal of Visual Languages and Computing*, 9:461–483, 1998.

[ETB05] C. Ermel, G. Taentzer, and R. Bardohl. Simulating Algebraic High-Level Nets by Parallel Attributed Graph Transformation: Long Version. Technical Report 2004-21, Technische Universität Berlin, 2005.

[EW05] K. Ehrig and J. Winkelmann. Model Transformation from VisualOCL to OCL using Graph Transformation. In *Proc. International Workshop on Graph and Model Transformation (GraMoT'05)*, ENTCS, Tallinn, Estonia, September 2005. Elsevier Science.

[FKN+92] A. Finkelstein, J. Kramer, B. Nuseibeh, M. Goedicke, and L. Finkelstein. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–58, March 1992.

[Fla04] Macromedia, Inc. *Macromedia Flash MX 2004*, 2004. http://www.macromedia.com/software/flash/.

[GEF06] Eclipse Consortium. *Eclipse Graphical Editing Framework (GEF) – Version 3.2*, 2006. http://www.eclipse.org/gef.

[Gen] GenGED Homepage. http://tfs.cs.tu-berlin.de/genged.

[Gip75] J. Gips. *Shape Grammars and Their Uses, Artificial Perception, Shape Generation and Computer Aesthetics*. Birkhäuser Verlag, Basel, 1975.

[Gog00] M. Gogolla. Graph Transformations on the UML Metamodel. In A. Corradini and R. Heckel, editors, *Proc. ICALP Workshop Graph Transformations and Visual Modeling Techniques*. Carleton Scientific, 2000.

[Gol91a] E.J. Golin. *A Method for the Specification and Parsing of Visual Languages*. PhD thesis, Brown University, 1991.

[Gol91b] E.J. Golin. Parsing Visual Languages with Picture Layout Grammars. *Journal of Visual Languages and Computing*, 2(4):371–394, 1991.

[GPP98] M. Gogolla and F. Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In M. Broy, D. Coleman, T. Maibaum, and B. Rumpe, editors, *Proc. ICSE'98 Workshop Precise Semantics of Modeling Techniques*, pages 55–72, 1998.

[GPS99] M. Große–Rhode, F. Parisi Presicce, and M. Simeoni. Refinements and Modules for Typed Graph Transformation Systems. In J. L. Fiadeiro, editor, *Workshop on Algebraic Development Techniques (WADT'98), at ETAPS'98, Lisbon, April 1998*, pages 137–151. Springer LNCS 1589, 1999.

[GPS00] M. Große–Rhode, F. Parisi Presicce, and M. Simeoni. Refinements of Graph Transformation Systems via Rule Expressions. In *6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), LNCS 1764*, pages 368–382. Springer, 2000.

[Gra94] J. Grabowski. *Test Case Generation and Test Case Specification Based on Message Sequence Charts*. PhD thesis, Dissertation, Universität Bern, Institut für Informatik, February 1994.

[Gra99] B. Grahlmann. The State of PEP. In M. Haeberer A. editor, *Proc. of AMAST'98 (Algebraic Methodology and Software Technology)*, volume 1548 of *Lecture Notes in Computer Science*. Springer-Verlag, January 1999.

[Gra01] A. Grau. *Computer-Aided Validation of Formal Conceptual Models*. PhD thesis, Technical University of Braunschweig, 2001.

[Gri96] P. Griebel. *Paralleles Lösen von grafischen Constraints*. PhD thesis, University of Paderborn, Germany, February 1996.

[GRPS97a] M. Große-Rhode, F. Parisi Presicce, and M. Simeoni. Concrete Spatial Refinement Constructions for Graph Transformation Systems. Technical Report 97–10, Università di Roma *La Sapienza*, Dip. Scienze dell'Informazione, 1997.

[GRPS97b] M. Große-Rhode, F. Parisi Presicce, and M. Simeoni. Spatial and Temporal Refinement of Typed Graph Transformation Systems. Technical Report 97–11, Università di Roma *La Sapienza*, Dip. Scienze dell'Informazione, 1997.

[GRR99] M. Gogolla, O. Radfelder, and M. Richters. Towards three-dimensional representation and animation of UML diagrams. In R. France and B. Rumpe, editors, *Proc. 2nd Int. Conf. Unified Modeling Language (UML'99)*, volume 1723 of *LNCS*, pages 489–502. Springer, 1999.

[GS00] J. Gray and S. Schach. Constraint Animation Using an Object-Oriented Declarative Language. In *Proc. 38th Annual ACM Software Engineering Conference, Clemson, South Carolina, USA*, pages 1–10, 2000.

[Har87] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HAS02] C. D. Hundhausen, Douglas S. A., and J.T. Stasko. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290, 2002.

[HCEL96] R. Heckel, A. Corradini, H. Ehrig, and M. Löwe. Horizontal and vertical structuring of typed graph transformation systems. *Math. Struc. in Comp. Science*, 6(6):613–648, 1996. Also as techn. report no 96-22, TU Berlin.

[Hec98] R. Heckel. *Open Graph Transformation Systems: A New Approach to the Compositional Modelling of Concurrent and Reactive Systems*. PhD thesis, Technical University of Berlin, Dep. of Comp. Sci., 1998.

[HEC03] D. Harel, S. Efroni, and I.R. Cohen. Reactive Animation. In F.S. de Boer, M.M. Bonsangue, and S. Graf et al., editors, *Proc. First Int. Symposium on Formal Methods for Components and Objects (FMCO'02)*, volume 2852 of *LNCS*, pages 136–153. Springer, 2003.

[HEET99] R. Heckel, H. Ehrig, G. Engels, and G. Taentzer. A View-Based Approach to System Modeling based on Open Graph Transformation Systems. In H. Ehrig, G. Engels, J.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[HG97] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[HHS04] J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling with time: Specifying the semantics of multimedia sequence diagrams. *Journal of Software and Systems Modelling*, 3(3):181–193, 2004.

[HHT96] A. Habel, R. Heckel, and G. Taentzer. Graph Grammars with Negative Application Conditions. *Special issue of Fundamenta Informaticae*, 26(3,4):287–313, 1996.

[HKT02a] R. Heckel, J. Küster, and G. Taentzer. Confluence of Typed Attributed Graph Transformation with Constraints. In A. Corradini, H. Ehrig, H.-J. Kreowski, and Rozenberg. G., editors, *Proc. of 1st Int. Conference on Graph Transformation*, volume 2505 of *LNCS*. Springer, 2002.

[HKT02b] R. Heckel, J. Küster, and G. Taentzer. Towards Automatic Translation of UML Models into Semantic Domains . In H.-J. Kreowski, editor, *Proc. of APPLIGRAPH Workshop on Applied Graph Transformation (AGT 2002)*, pages 11 – 22, 2002.

[HM90] R. Helm and K. Marriott. Declarative Specification of Visual Languages. In *Proc. IEEE Symp. on Visual Languages*, pages 98–103, Skokie, Illinois, October, 4-6 1990.

[HM03] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[Hos98] P. Hoschka. *Synchronized Multimedia Integration Language (SMIL) Specification – Version 1.0*, 1998. Available at http://www.w3.org/TR/REC-smil/.

[ISO93] ISO/IEC2382-1:1993. Iso standard on information technology – vocabulary – part 1: Fundamental terms, 1993.

[Jav04] Sun Microsystems. *Java – Version 1.5*, 2004. http://java.sun.com.

[JE97] I. Uhe J. Ebert, R. Sttenbach. Meta-case in practice: a case for Kogge. In J. A. Pastor A. Olive, editor, *Advanced Information Systems Engineering, Proceedings of the 9th International Conference, CAiSE'97, Barcelona, Catalonia, Spain, June 16-20, 1997*, number 1250 in LNCS, pages 203–216, Berlin, 1997.

[Jia95] X. Jia. An approach to animating Z specifications. In *COMPSAC 95*, 1995.

[JMBH93] S. Jawed, I. Morrey, G. Buckberry, and R. Hibberd. Towards CASE Tools for Prototyping Z Specifications. In *Proc. CASE'93*, pages 166–173, 1993.

[Kar05] B. Karwan. Entwurf und Implementierung eines Interpreters fr amalgamierte Graphtransformation. Master's thesis, Technische Universitt Berlin, Fak. Elektrotechnik/Informatik, 2005.

[KC95] M. Keil and E. Carmel. Customer-developer links in software development. *Communications of the ACM*, pages 33–34, May 1995.

[KGKK02] S. Kuske, M. Gogolla, H.-J. Kreowski, and R. Kollmann. An integrated semantics for UML class, object and state diagrams based on graph transformation. In M. Butler, L. Petre, and K. Sere, editors, *Proc. 3rd Int. Conf. on Integrated Formal Methods (IFM 2002)*, volume 2335 of *LNCS*, pages 11–28. Springer, 2002.

[KHK06] H.-J. Kreowski, K. Hölscher, and P. Knirsch. Semantics of visual models in a rule-based setting. *Electr. Notes Theor. Comput. Sci.*, 148(1):75–88, 2006.

[KK93] P. R. Keller and M. M. Keller. *Visual Clues - Practical Data Visualization*. IEEE Computer Society Press, 1993.

[KK99] H.-J. Kreowski and S. Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11:690–723, 1999.

[KLR96] S. Kelly, K. Lyytinen, and M. Rossi. MetaEdit+ A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In *Proc. Eighth International Conference CAiSE 1996*, volume 996 of *LNCS*. Springer-Verlag, 1996.

[Kre78] H.-J. Kreowski. *Manipulation von Graphmanipulationen*. PhD thesis, Technical University of Berlin, Dep. of Comp. Sci., 1978.

[Kre81] H.-J. Kreowski. A Comparison between Petri Nets and Graph Grammars. In *5th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 1–19. LNCS 100, Springer, 1981.

[Kre93] H.-J. Kreowski. Translations into the Graph Grammar Machine. In R. Sleep, R. Plasmeijer, and M. van Eekelen, editors, *Term Rewriting: Theory and Practice*, pages 171–183. John Wiley, 1993.

[Kus00] S. Kuske. *Transformation Units – A Structuring Principle for Graph Transformation Systems*. PhD thesis, University of Bremen, 2000.

[Kus01] S. Kuske. A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 – The Unified Modeling Language. Modeling languages, Concepts, and Tools*, volume 2185 of *LNCS*, pages 241–256. Springer, 2001.

[KW87] H.-J. Kreowski and A. Wilharm. Is Parallelism already Concurrency? Part 2: Non-Sequential Processes in Graph Grammars. In *3rd Int. Workshop on Graph Grammars and their Application to Computer Science, LNCS 291*, pages 361–377. Springer, 1987.

[Lab06] Cold Spring Harbor Laboratory. Biology animation library, 2006. [Online; accessed 18-March-2006].

[LKW93] M. Löwe, M. Korff, and A. Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In M.R. Sleep, M.J. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley & Sons Ltd, 1993.

[LM95] M. Löwe and J. Müller. Critical Pair Analysis in Single-Pushout Graph Rewriting. In G. Valiente Feruglio and F. Rosello Llompart, editors, *Proc. Colloquium on Graph*

Transformation and its Application in Computer Science. Technical Report B-19, Universitat de les Illes Balears, 1995.

[LRS95] J. Linneberg Rasmussen and M. Singh. *Mimic/CPN: A Graphic Animation Utility for Design/CPN. User's Manual, Version 1.5*, 1995. http://www.daimi.au.dk/designCPN/libs/mimic/.

[Mac71] S. MacLane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer, New York, 1971.

[Mar94] K. Marriott. Constraint Multiset Grammars. In *Proc. IEEE Symp. on Visual Languages*, pages 118–125, St. Louis, Missouri, October, 4-7 1994.

[Mey97] B. Meyer. Formalization of Visual Mathematical Notations. In *Proc. of AAAI Symposium on Diagrammatic Reasoning*, Boston, USA, 1997.

[MG97] M. Minas and J. Gottschall. Specifying Animated Diagram Languages. In *Proc. Workshop on Theory of Visual Languages*, pages 51–59, Capri, Italy, September 1997.

[MH01] M. Minas and B. Hoffmann. Specifying and implementing visual process modeling languages with d¡small¿¡small¿iagen¡/small¿¡/small¿. *Electronic Notes in Theoretical Computer Science*, 44(4), 2001.

[Min01] M. Minas. *Spezifikation und Generierung graphischer Diagrammeditoren*. PhD thesis, Universitt, Erlangen, 2001.

[Min02] M. Minas. Specifying graph-like diagrams with diagen. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.

[MM98] K. Marriott and B. Meyer. *Visual Language Theory*. Springer, 1998.

[MOF05] Object Management Group. *Meta-Object Facility (MOF), Version 1.4*, 2005. http://www.omg.org/technology/documents/formal/mof.htm.

[MSHB98] I. Morrea, J. Siddiqi, R. Hibberd, and G. Buckberry. A toolset to support the construction and animation of formal specifications. *Systems and Software*, 41:147–160, 1998.

[MSP96] A. Maggiolo-Schettini and A. Peron. A Graph Rewriting Framework for Statecharts Semantics. In J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 1073 of *LNCS*, pages 107–121. Springer, 1996.

[Mue05] R. A. Mueller. Test case generation using symbolic execution. In *Proc. VVSS 2005, Eindhoven, Niederlande*, Nov 2005.

[OCL03] Object Management Group. *UML 2.0 OCL Specification*, 2003. http://www.omg.org/docs/ptc/03-10-14.pdf.

[OK99] I. Oliver and S. Kent. Validation of object-oriented models using animation. In *Proc. of EuroMicro'99, Milan, Italy*, 1999.

[Owe99] G. S. Owen. Hypervis – teaching scientific visualization using hypermedia. Project of the ACM SIGGRAPH Education Committee, the National Science Foundation (DUE-9752398), and the Hypermedia and Visualization Laboratory, Georgia State University, 1999. http://www.siggraph.org/education/materials/HyperVis/hypervis.htm.

[PER95] J. Padberg, H. Ehrig, and L. Ribeiro. Algebraic high-level net transformation systems. *Mathematical Structures in Computer Science*, 5:217–256, 1995.

[PP96] F. Parisi-Presicce. Transformation of Graph Grammars. In *5th Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 1073 of *LNCS*. Springer, 1996.

[PPEM87] F. Parisi-Presicce, H. Ehrig, and U. Montanari. Graph Rewriting with Unification and Composition. In *3rd Int. Workshop on Graph Grammars and their Application to Computer Science, LNCS 291*, pages 496–514, Berlin, 1987. Springer Verlag.

[PSS90] C. D. Pegden, R. E. Shannon, and R. P. Sadowski. *Introduction to Simulation Using SIMAN*. McGraw-Hill, 1990.

[Rei85] W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, 1985.

[RG00] O. Radfelder and M. Gogolla. On better understanding UML diagrams through interactive three-dimensional visualization and animation. In Vito Di Gesu, Stefano Levialdi, and Laura Tarantino, editors, *Proc. Advanced Visual Interfaces (AVI 2000)*, pages 292–295. ACM Press, New York, 2000.

[Rha05] I-Logix. *Rhapsody: Model-Driven Development with UML 2.0, SysML and Beyond*, 2005. http://www.ilogix.com/rhapsody/rhapsody.cfm.

[Rib96] L. Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars*. PhD thesis, TU Berlin, 1996.

[Ros76] A. Rosenfeld. Array and Web Languages: An Overwiew. In A. Lindenmayer and G. Rozenberg, editors, *Automata, Languages, Development*, pages 517–529, North Holland, Amsterdam, 1976.

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[RWL+03] A. V. Ratzer, L. Wells, H. M. Lassen, M. Laursen, J. Qvortrup, M. Stissing, M. Westergaard, S. Christensen, and K. Jensen. Cpn tools for editing, simulating, and analysing coloured petri nets. In Mark Minas, editor, *Proc. 24th Intern. Conf. on Applications*

and Theory of Petri Nets (ICATPN 2003), Eindhoven, The Netherlands, volume 2679, pages 450–462. LNCS, 2003.

[SC00] P. Schmitz and A. Cohen. *SMIL Animation*, 2000. Available at http://www.w3.org/TR/2001/REC-smil-animation-20010904/.

[Sch05] L. Schaps. Entwurf und Implementierung eines Systems zur Ausführung von UML- und OCL-Spezifikationen auf der Basis von Graphtransformation. Master's thesis, Universität Bremen, 2005.

[SDBP98] J.T. Stasko, J. B. Domingue, M.H. Brown, and B.A. Price, editors. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, 1998.

[Sta90] J. T. Stasko. Tango: A Framework and System for Algorithm Animation. *IEEE Computer*, 23(9), 1990.

[SVG02] *SVG Toolkit. CSIRO SVG Toolkit – Release March 12th 2002*, 2002. Available at http://www.cmis.csiro.au/svg/.

[SVG04] WWW Consortium (W3C). *SVG Implementations*, 2004. http://www.w3.org/Graphics/SVG/SVG-Implementations.htm8.

[SW93] J. T. Stasko and J.F. Wehrli. Three-dimensional Computation Visualization. In *Proc. IEEE Symposium on Visual Languages*, pages 100–107, August 1993.

[SWZ99] A. Schürr, A. Winter, and A. Zündorf. The PROGRES-approach: Language and environment. In H. Ehrig, G. Engels, J.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[Sys04] Sysoft. *Visio Diagram Animation with Amarcos*, 2004. http://www.sysoft-fr.com/en/Amarcos.

[Tae96] G. Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. PhD thesis, TU Berlin, 1996. Shaker Verlag.

[Tae01] G. Taentzer. Towards Common Exchange Formats for Graphs and Graph Transformation Systems. In J. Padberg, editor, *Proc. Uniform Approaches to Graphical Process Specification Techniques (UNIGRA'01)*, volume 44 (4) of *ENTCS*, 2001.

[Tae03] G. Taentzer. AGG: A Graph Transformation Environment for System Modeling and Validation. In T. Margaria, editor, *Proc. Tool Exihibition at 'Formal Methods 2003'*, Pisa, Italy, September 2003.

[Tae04] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In J. Pfaltz, M. Nagl, and B. Boehlen, editors, *Application of Graph Transformations with Industrial Relevance (AGTIVE'03)*, volume 3062 of *LNCS*, pages 446 – 456. Springer, 2004.

[Tae06] G. Taentzer. Graph Transformation in Software Engineering. Internal Report, TU Berlin, 2006.

[TE00] A. Tsiolakis and H. Ehrig. Consistency Analysis of UML Class and Sequence Diagrams using Attributed Graph Grammars. In H. Ehrig and G. Taentzer, editors, *Proc. of Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation (GRATRA 2000)*, pages 77–86. Technical Report No. 2000-2, FB Informatik, TU Berlin, 2000.

[TEG+05] G. Taentzer, K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovsky, U. Prange, D. Varro, and S. Varro-Gyapay. Model Transformation by Graph Transformation: A Comparative Study. In *Proc. Workshop Model Transformation in Practice*, Montego Bay, Jamaica, October 2005.

[TER99] G. Taentzer, C. Ermel, and M. Rudolf. The AGG-Approach: Language and Tool Environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools*, pages 551–603. World Scientific, 1999.

[TS95] B. Topol and J. T. Stasko. Integrating Visualization Support into Distributed Computing Systems. In *Proc. 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 19–27, Vancouver, Canada, 1995.

[UML04a] *Unified Modeling Language – version 1.5*, 2004. http://www.omg.org/technology/documents/formal/uml.htm.

[UML04b] *Unified Modeling Language: Superstructure – Version 2.0*, 2004. Revised Final Adopted Specification, ptc/04-10-02, http://www.omg.org/cgi-bin/doc?ptc/2004-10-02.

[UML05] UML 2.0 Semantics Project. School of Computing, Queen's University Kingston, Canada, 2005. http://www.cs.queensu.ca/home/stl/internal/uml2/.

[Utt00] M. Utting. Data Structures for Z Testing and Animation Tools. In *Proc. Formal Methods Tools*. University of Ulm, Germany, 2000.

[Var02] D. Varró. A formal semantics of UML Statecharts by model transition systems. In A. Corradini, H. Ehrig, H. J. Kreowski, and G. Rozenberg, editors, *Proc. ICGT 2002: 1st Int. Conf. on Graph Transformation*, volume 2505 of *LNCS*, pages 378–392. Springer, 2002.

[Wal95] R. Walter. *Petrinetzmodelle verteilter Algorithmen – Intuition und Beweistechnik*. Bertz Verlag, VERSAL edition, 1995.

[Wan94] D. Wang. *Studies on the formal semantics of pictures*. PhD thesis, University of Amsterdam, 1994. ILLC dissertation series 1995-4.

[Wei01] I. Weinhold. Konzeption und Implementierung eines Generators für Simulationsumgebungen. Master's thesis, Technische Universität Berlin, FB Informatik, 2001.

[Wik06] Wikipedia. Educational animation — wikipedia, the free encyclopedia, 2006. [Online; accessed 15-March-2006].

[Wit91] K. Wittenburg. Parsing with relational unification grammars. In *Proc. of 1991 International Workshop on Parsing Technologies (IWPT'91)*, pages 225–234, Cancun, Mexico, 1991. Association for Computational Linguistics.

[WW96] K. Wittenburg and L. Weitzman. Relational grammars: Theory and practice in a visual language interface for process modeling. In *Proc. of the AVI'96 Workshop Theory of Visual Languages*, 1996.

[WWW03] WWW Consortium (W3C). *Scalable Vector Graphics (SVG) 1.1 Specification.*, 2003. http://www.w3.org/TR/svg11/.

[X3D05] Web3D Consortium: Open Standards for Real-Time 3D Communication. *X3D ISO Standard for Real-Time 3D Content and Applications Running on a Network*, 2005. http://www.web3d.org/x3d/.

[ZHG04a] P. Ziemann, K. Hölscher, and M. Gogolla. Coherently explaining UML Statechart and collaboration diagrams by graph transformations. In A. Moura and A. Mota, editors, *Proc. of the Brazilian Symposium on Formal Methods (SBMF 2004)*. Electronic Notes in Theoretical Computer Science (ENTCS), Elsevier, 2004.

[ZHG04b] P. Ziemann, K. Hölscher, and M. Gogolla. From UML models to graph transformation systems. In Mark Minas, editor, *Proc. Workshop on Visual Languages and Formal Methods (VLFM 2004)*. ENTCS, 2004.

Index

I

L

M

N

P

R

S

T

V