Christian-Albrechts-Universität zu Kiel

Diploma Thesis

On Integrating Graphical and Textual Modeling

Christian Schneider

February 9th 2011

Department of Computer Science
Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

Dipl.-Inf. Miro Spönemann
Dipl.-Inf. Hauke Fuhrmann

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

____

Abstract

Designing and implementing software systems by means of modeling techniques has gained much popularity in the past decades. In practice, however, composing models is often rather cumbersome and tedious, since current modeling tools do not exploit the full potential of this kind of "writing down" content. This issue is addressed by this thesis. More specifically, methods for extracting chosen information from arbitrarily shaped models and its manipulation are developed.

This work is motivated by the desire to seamlessly integrate different textual and graphical DSLs in the domain of railway signaling systems, which is part of the MENGES (*Modellgetriebene Entwurfsmethoden für eine neue Generation elektronischer Stellwerke*) project. To address this problem, this thesis proposes the *transient view approach*, which allows to decouple models and their representation. The concept has been prototypically implemented within the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER), an academic testbed targeting the investigation of advanced methods for treating models.

This thesis also presents a number of use-cases within the design of railway signaling applications that take advantage of the transient view approach.

Contents

1 Introduction **1**
1.1 Towards Model-Driven Software Development 1
1.2 Problem statement and approach to a solution 6
1.3 Outline of this work 8

2 Related work **11**
2.1 Graphical Modeling 12
2.2 Textual Modeling 14
2.3 Combination of Graphical and Textual Modeling 18

3 Used Technologies **23**
3.1 The Eclipse Project 23
3.2 Eclipse Modeling Framework (EMF) 26
3.3 EMF Compare 28
3.4 Xtend 30
3.5 Graphical Modeling Framework (GMF) 32
3.6 Papyrus 33
3.7 Xtext 35

4 Towards Multiform Model Editing **39**
4.1 Repository-Based Model Management 39
4.2 Lightweight View Construction Methods 40
4.3 Model Representations based on Transient Views 45

5 Multiform Modeling in Kieler **49**
5.1 The Thin KIELER SyncCharts Editor (ThinKCharts) 50
5.2 A textual SyncCharts language 57
5.3 Integration of Transient Textual Views 67
5.4 Remarks on the Implementation 69
5.5 Open Issues 74

6 On Integrating Modeling Language Families - The MENGES Example **77**
6.1 The behavior specification language *GALogic* 79

6.2 Providing Graphical Views 83
6.3 Assessment 89
6.4 Towards Interactive Model Browsing & Editing 90

7 Conclusion **99**
Future Work 100

Bibliography **105**

A Grammars of KIELER Textual SyncCharts (KITS) **111**
A.1 Grammar of KITS 111
A.2 Grammar of actions 113
A.3 Grammar of expressions 114
A.4 Grammar of annotations 117

List of Figures

1.1 Various UML Diagrams. 2
1.2 The Meta-Object Facility (MOF) architecture. 4
1.3 Reorganization of software systems. 5

2.1 Model-View-Controller (MVC)-based implementation of GEF. 12
2.2 A simple GMF-based mindmap editor. 13
2.3 Specification of abstract and concrete syntax of a GMF-based editor. 14
2.4 TCS and its context (Jouault et al. [24]). 15
2.5 Overview on the artifacts of EMFText (Heidenreich et al. [21]). . . . 17
2.6 Projectional editor of the Meta Programming System (MPS). 18
2.7 Model representations in the KIEL tool (Wischer [55]). 19
2.8 Hovering textual editor provided by TEF. 20
2.9 Multi-View-DSLs in AToM3 (Guerra & de Lara [19]). 20
2.10 AToM3 equipped with textual views (Andrés et al. [3]). 21

3.1 Relations of Eclipse plugins (Fuhrmann [11]). 24
3.2 The Eclipse Workbench. 25
3.3 Components of Eclipse (Fuhrmann [11]). 25
3.4 EMF's tree-based and graphical Ecore editors. 27
3.5 Eclipse's Compare tool. 28
3.6 The process of comparing models by means of EMF Compare. 29
3.7 Data structures employed by GMF-based editors. 33
3.8 The Papyrus modeling environment. 34
3.9 A Xtext-based editor tailored to a Domain-Specific Language (DSL). 35

4.1 Repository-based model management in AToM3 (Guerra & de Lara [19]). 40
4.2 Obtaining model representations by means of model transformations. 41
4.3 Obtaining model representations by means of dedicated model viewers. 42
4.4 File-based synchronization of multiple model editors. 43
4.5 *Difference-and-merge*-based synchronization of multiple model editors. 45
4.6 *Single source/multiple view*-based model representations. 46
4.7 Cross-metamodel-based supply of model representations. 47

5.1 Components of KIELER. 50
5.2 The “hello world!” example of *SyncCharts*, called ABRO. 51
5.3 Types of states in SyncCharts. 52
5.4 Regions with initial states. 52
5.5 Types of transitions. 52
5.6 Signals and variables attached to a state. 53
5.7 Usage of signals and variables in ThinKCharts. 53
5.8 Usage of state actions in ThinKCharts. 54
5.9 Graphical notes provided by GMF. 55
5.10 The SyncCharts meta model. 56
5.11 The expressions meta model. 56
5.12 The annotations meta model. 63
5.13 ABRO with annotations. 66
5.14 Integrated graphical and textual views in ThinKCharts. 68
5.15 (Re-)Initialization of the KITS view. 71
5.16 Multiple embedded text editors stacked on each other. 72
5.17 Transmitting modifications to ThinKCharts. 73
5.18 Exemplary glitch occuring while modeling with the KITS view. 74

6.1 Kinds of specifications of an interlocking block design. 78
6.2 Rule graph of rule block defined in Listing 6.4 laid out manually. . . 83
6.3 Invocation of the behavior diagram generation. 85
6.4 State machine defined in Listing 6.2 arranged with *KLay Layered*. . . 86
6.5 State machine defined in Listing 6.2 arranged with *Graphviz Circo*. . 88
6.6 View on the rule block defined in Listing 6.5 with layout defects. . . 89
6.7 Graphical view of a rule block employing predicates and procedures. 91
6.8 Graphical view after the expansion of predicates and procedures. . . 91
6.9 Exploring hidden details in a model diagram. 92
6.10 Graphical view supplemented with auxiliary information. 92
6.11 Navigation handles in graphical representations in Topcased. 93
6.12 Principle of automatically updating opened views. 94
6.13 Interactive model exploration. 95
6.14 Multiform model editing by means of model views. 96
6.15 Overview on field elements. 97

7.1 Validation and quickfix proposals in Xtext-based editors. 101
7.2 Graphical visualization of the emit-test-relation of signals (Müller [34]). 103

List of Listings

3.1 Comparing and merging models programmatically. 30

5.1 Grammar excerpt of action declarations in ThinKCharts. 54
5.2 `ABRO` in the textual KITS language. 57
5.3 Grammar of KITS's `state` statement. 58
5.4 Grammar of KITS's `region` statement. 59
5.5 Grammar of the transition definition in KITS. 60
5.6 Grammar of the `signal` declaration in KITS. 61
5.7 Grammar of the variable declaration in KITS. 62
5.8 Grammar definition of effects in KITS. 62
5.9 Grammar definition of `TextualCode` in KITS. 62
5.10 Grammar definition of comment annotations. 64
5.11 Grammar definition of key-value-annotations. 64
5.12 Grammar definition of `StringAnnotation`. 65
5.13 `ABRO` with annotations. 65

6.1 Top level rule of the grammar of *GALogic*. 79
6.2 Example of a state machine in *GALogic*. 80
6.3 Grammar of state machines in *GALogic*. 80
6.4 Example of a rule block in *GALogic*. 81
6.5 A rule block containing reference alternatives. 82
6.6 Grammar of rule blocks in *GALogic*. 82

A.1 Grammar of KITS. 111
A.2 Actions grammar used by KITS. 113
A.3 Expressions grammar used by KITS. 114
A.4 Annotations grammar used by KITS. 117

List of Abbreviations

AMMA	Atlas Model Management Architecture
ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AST	Abstract Syntax Tree
ATL	Atlas Transformation Language
AToM3	A Tool for Multi-formalism and Meta-modelling
CASE	Computer-Aided Software Engineering
DAG	directed acyclic graph
DSL	Domain-Specific Language
EBNF	Extended Backus-Naur Form
EMF	Eclipse Modeling Framework
EMOF	Essential MOF
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
HUTN	Human-Usable Textual Notation
IDE	Integrated Development Environment
JaMoPP	Java Model Parser and Printer
JVM	Java Virtual Machine
KEI	KIELER Execution Infrastructure

KIEL	Kiel Integrated Environment for Layout
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KIML	KIELER Infrastructure for Meta Layout
KiVi	KIELER View Management
KITS	KIELER Textual SyncCharts
KM3	Kernel MetaMetaModel
KSBasE	KIELER Structure-Based Editing Framework
LDK	Language Development Kit
M2T	Model To Text
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MDE	Model-Driven Engineering
MDSD	Model-Driven Software Development
MDSE	Model-Driven Software Engineering
MDT	Model Development Tools
MENGES	Modellgetriebene Entwurfsmethoden für eine neue Generation elektronischer Stellwerke
MOF	Meta-Object Facility
MPS	Meta Programming System
MVC	Model-View-Controller
MWE	Modeling Workflow Engine
oAW	open Architecture Ware
OCL	Object Constraint Language
OMG	Object Modeling Group
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
QVT	Query/View/Tranformation

RCA	Rich Client Application
RCP	Rich Client Platform
SA	Structured Analysis
SC	Synchronous C
SD	Structured Design
SysML	Systems Modeling Language
TCS	Textual Custom Syntax
TEF	Textual Editing Framework
TGE	Textual Generic Editor
ThinKCharts	Thin KIELER SyncCharts Editor
UML	Unified Modeling Language
URI	Uniform Resource Identifier
UTF	Unicode Transformation Format
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

Chapter 1

Introduction

Building models, or just *modeling*, is common practice in the natural and engineering sciences as well as in industrial practice. It is an invaluable means for abstraction in order to cope with the complexity of real world objects or processes. Modeling has existed at least since the antiquity. Whenever people try to investigate things they cannot conceive exhaustively, they construct models in their mind. These models reflect currently approved and/or assumed properties of the of real world correspondents. Famous models are the Ptolemaic system, Newton's law of motion, the Mendelian inheritance, and Einstein's mass-energy equivalence ($E = mc^2$).

One of today's most important application fields of (mathematical) models is weather forecasting. In our civilized and networked world undiscovered extreme weather may lead to extensive damage and even deaths. Further disciplines leveraging modeling are the machinery and constructional engineering as well as economics. Developers of motorized vehicle engines employ models while optimizing the combustion. Constructional engineers perform stress analyses on models of buildings to be erected. Economists rely on statistical methodologies in order to describe consumer behavior. Considering the manifold possibilities for the employment of modeling the software engineers' utilization of this paradigm is self-evident.

1.1 Towards Model-Driven Software Development

Formalizing requirements and design specifications of software systems has been a problem for decades. Lots of techniques have been proposed and some of them have become accepted in practice.

Important approaches are the Structured Analysis (SA) [8, 56] and Structured Design (SD) [57, 38]. Whereas the purpose of SA is to improve the requirements specifications and the processes of creating them, the aim of SD is to bridge the gap between the requirements and the implementation. Although these methods have been developed independently, they usually occur in alliance. They had their period of prosperity in the late 1970s and 1980s and have been superseded by the Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD) in the 1990s.

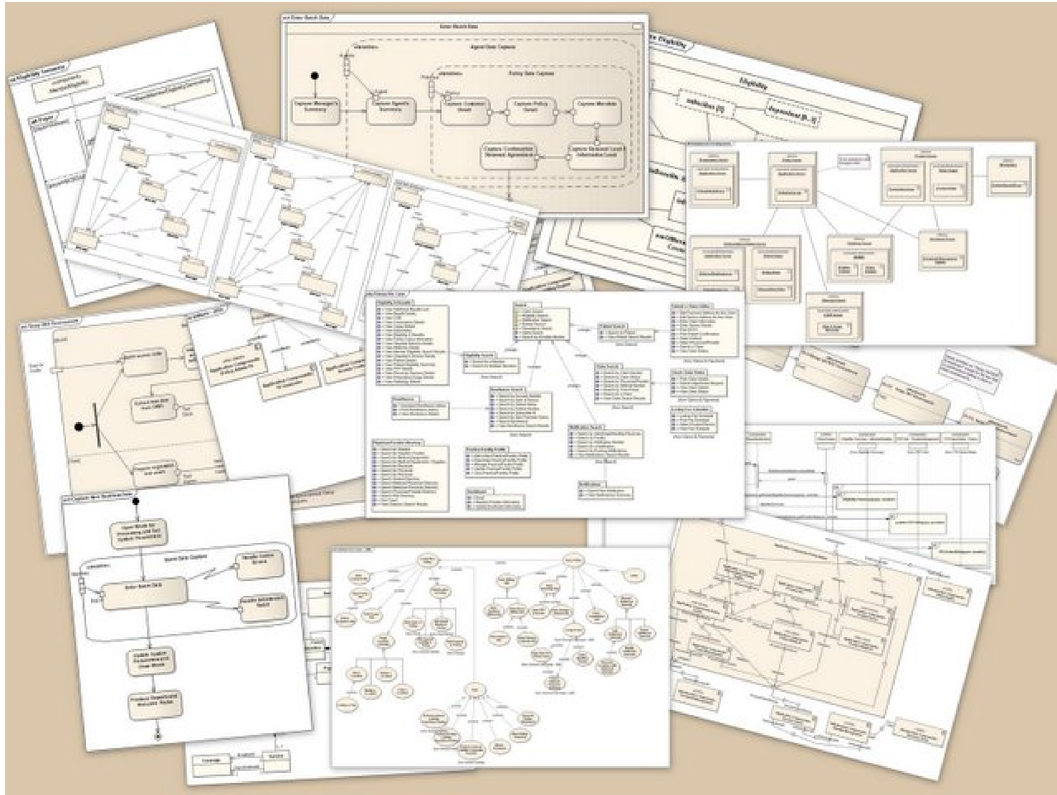


Figure 1.1: Various UML Diagrams.[1]

The Unified Modeling Language

Various object-oriented methods have been proposed by Rumbaugh [41], Jacobson [23] and Booch [6]. These approaches culminated in the Unified Modeling Language (UML) that has been officially accepted by the Object Modeling Group (OMG) in November 1997 and gets enhanced continuously. The UML is a graphical language providing a bunch of *diagrams* that are basically sub languages. Each of them is designed to allow the formulation of certain aspects of the business domain or software system designs. Examples are the *Use Case Diagram*, the *Class Diagram*, the *UML State Machine*, the *Sequence Diagram*, the *Communication Diagram*, and the *Component Diagram*. A selection of available diagrams is shown in Figure 1.1. For detailed information on them refer to the official specification [37] or Booch et al. [7].

[1] http://en.wikipedia.org/wiki/File:UML_Diagrams.jpg

Domain-Specific Languages

A source of criticism of the UML is its *one-size-fits-all* approach as pointed out by Stahl et al. [48]. In order to tailor the design processes and employed modeling languages such that they meet the demands of the particular business domains, different approaches have been pursued:

1. Since release 1.3 the UML comes with a specializing mechanism called *UML Profiles*. This concept provides the ability of specializing the UML itself, resulting in more fine-grained semi-custom modeling languages providing, e. g., several types of classes. An important example of such a language is the Systems Modeling Language (SysML) [36].

2. Contrary to the *generic* language UML the idea of DSLs was carried into the software systems modeling field. Full-fledged DSLs are characterized by simplicity, abstraction, conciseness, freedom of redundancy, and the focus on a certain niche. These properties imply an increased usability, maintainability, and portability as specifications may address facts, problems, and solutions in a high level fashion. Details on how to implement solutions are to be omitted.

According to Fowler [10] a DSL is determined by the following parts: The *abstract syntax* characterizes the data that shall be expressed in the DSL. An *editor* must be provided, allowing to manipulate the content, which implies the definition of a *concrete syntax*. This concrete syntax can be of graphical or textual nature, or a mixture of both, i. e., composed of a graphical stand supplemented by textual refinements. Finally, a *generator* has to be given describing the translation of the content into executable data, i. e., defining the semantics of the content.

Internal and External DSLs

There are two variants of DSLs. The first one, internal DSLs, are built upon existing languages. This concept has been very common amongst LISP programmers since the early days of this programming language. The Synchronous C (SC) language (formerly named SyncCharts in C) proposed by von Hanxleden [54] can be considered as an internal DSL constructed upon the hosting language C. Even the formerly mentioned UML Profiles are proper DSLs, provided the semantics of their refinements are given according to the above criterion.

Advantages of internal DSLs are the availability of editor tooling, provided such editors exist for the host language. Moreover, a valuable concept of today's developing environments is the so-called *symbolic integration* (Fowler [10]). It enables features like *content assist* and immediate checks, e. g., *use-def-analyses*. Depending on the type of DSL and the semantics of its constructs, these techniques will apply to the new language more or less well. On the other hand the restriction to syntactic constraints of the host language may be disadvantageous in some situations.

In contrast, external DSLs are built up from scratch. They can be constructed such that content can be formulated, edited and read as easily and comfortably as

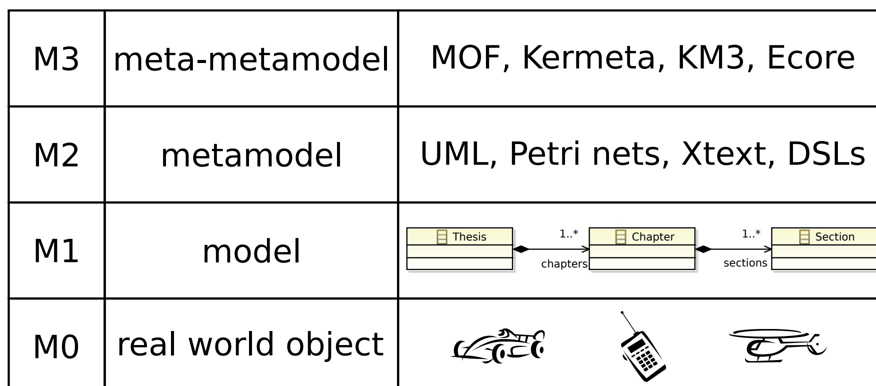


Figure 1.2: The Meta-Object Facility (MOF) architecture.

possible without any restriction. However, this freedom must be bought dearly by the prize of the development of tailored tool kits.

External DSLs are nothing new at all, examples exist since the early days in shape of the Unix minilanguages [39]. Further typical external DSLs are *Graphviz*' DOT language[2] and the input format of the configuration management tool *make*.[3]
In the following the term DSL is used synonymously with external DSL.

Metamodeling

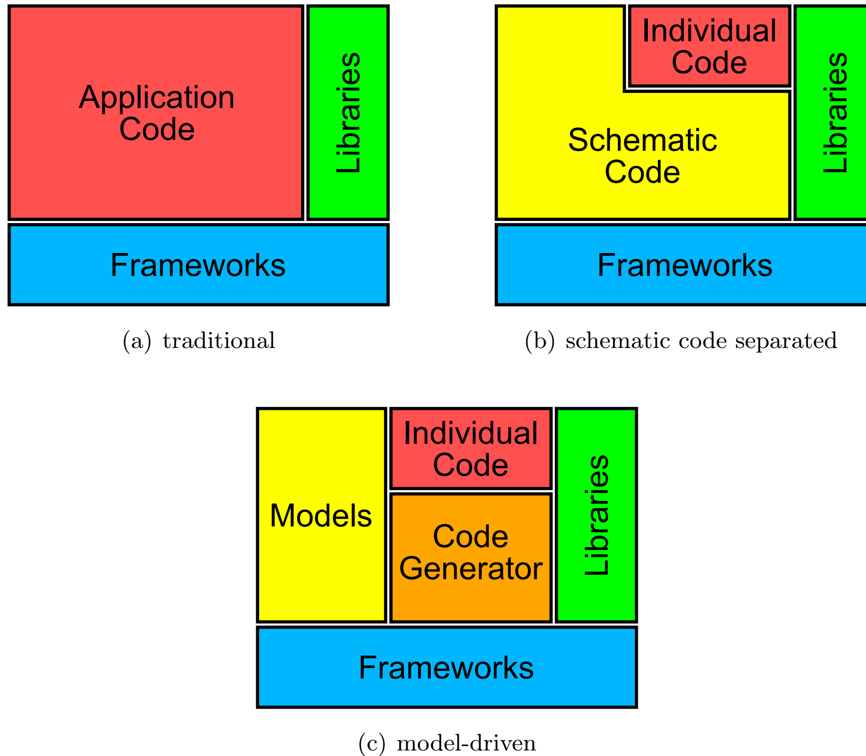
In oder to unify the definitions of modeling languages, the OMG approved the Meta-Object Facility (MOF) standard. This standard defines an architecture consisting of four layers, depicted in Figure 1.2.

As shown real objects to be described are placed at level 0 of this classification and are called M0 artifacts. Documents or specifications that describe those objects are called models and are situated at level 1. Typically, models describe only chosen aspects of real world objects while abstracting the remaining ones to overcome the objects' complexity.

Obviously, in order to describe anything, a language or formalism providing the necessary constructs must exist. Such formalisms are called metaobjects of level 2 (M2) or just *metamodels*. The set of the M2 languages comprises generic formalisms such as the UML or Petri nets, as well as the DSLs introduced above.

The MOF architecture is completed by the set of the M3 languages, alternatively called *meta-metamodeling languages*. M3 languages are intended to be employed while designing metamodels, e. g., DSLs. In order to cut the chain of metaobjects, valid M3 languages have to conform themselves, thus are described in itself. The first M3 language, the MOF language proposed by the OMG, originates in the UML. Due

[2] http://www.graphviz.org/doc/info/lang.html
[3] http://www.opengroup.org/onlinepubs/009695399/utilities/make.html



(a) traditional

(b) schematic code separated

(c) model-driven

Figure 1.3: Reorganization of software systems.

to its complexity a subset called Essential MOF (EMOF) has been proposed. Further examples of such languages are listed in Figure 1.2.

Model-Based and Model-Driven Software Development

Besides the change from modeling with generic modeling languages towards DSLs, the prominence of models in the software development processes has been altered eminently.

Traditionally, models are created to document facts and decisions, be printed out, and be delivered to the programmers who implement the software completely manually. In addition, those models serve for determining test cases and other purposes. The resulting software systems are structured as illustrated in Figure 1.3(a). Stahl et al. [48] refer to this approach as *model-based* software development.

The Model-Driven Software Development (MDSD) is characterized by the separation of the code into schematic parts and individual parts (Figure 1.3(b)). Whereas the individual portion has to be created manually anyway, the schematic parts are *derived* automatically by means of the models and dedicated code generators. Hence, in contrast to the former approach, models are an integral component of the MDSD process. The resulting software structure is depicted in Figure 1.3(c).

In the literature the terms Model-Driven Development (MDD), Model-Driven Engineering (MDE), and Model-Driven Software Engineering (MDSE) are used interchangeably in order to refer to the above explained approach. The term Model-Driven Architecture (MDA) was coined by the OMG and denotes its standardization initiative on MDSD. However, those concepts do not appear to be popular in the industry [48] and are very controversially discussed in the community [10].

1.2 Problem statement and approach to a solution

This diploma thesis addresses the problem of the tight linkage of models and their representations, which restricts the usability of models. In particular, textually formulated models are persisted in shape of their concrete representations; graphically specified models and representations are persisted separately. The manipulation of one part of these pairs, however, often leads to the loss of both. Hence, representations, which are also called views, are virtually part of valid models. In practice, however, tasks such as tracing mistakes in modeled specifications require other views than, e. g., exploring the entire system.

The novel approach presented here is henceforth termed *transient view approach*. Such transient views are considered to be a beneficial toolkit while working with models. Like different problems or tasks in the real world require different tools to resolve or perform them, different tasks within the MDSD may require different views on models. Each of those views is dedicated to particular tasks or purposes, e. g., the formulation of a specification or the verification of certain requirements of the design. Consequently, views shall not be maintained by modelers. Instead they are to be provided by the modeling environment such that they satisfy the needs of the modelers, and assist them as well as possible while performing certain tasks. Such model representations may differ in their degree of abstraction, their amount of detail, and their concrete syntax. They can be employed in order to represent information for various purposes and different interest groups (stakeholders). In addition, different representations typically exhibit different aptitudes with respect to facilitating modifications of model contents and displaying them onscreen or in printouts.

The transient view approach paves the way for advanced navigation and editing mechanisms in an industrial-oriented modeling environment that is being developed by MENGES[4], a joint research project of industrial and academic partners. The German-language project title *Modellgetriebene Entwurfsmethoden für eine neue Generation elektronischer Stellwerke* (MENGES) denotes the overall target of the project: the utilization of model driven design methods for a new generation of electronic interlocking blocks. A reference implementation of the transient view approach has been contributed to the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER), which is an academic, experimental modeling environment, in

[4] http://menges.informatik.uni-kiel.de

order to prove the feasibility of this concept.

Transient model views are supposed to be created on modeler's demand, allowing him or her to perform his or her task, and be dismissed if they are not needed anymore. They may display arbitrary content with a chosen amount of detail, which can be extracted from arbitrary sources. Note that *arbitrary* is restricted to a certain scope predetermined by concrete implementations.

The key enabler of this idea is the availability of reliable conditioning mechanisms. This denotes algorithms for arranging graphical representations, as well as techniques that are able to format textual views in a reasonable way. Specifically, graphical diagrams usually require dedicated layout methods that are adequate with respect to the kind of information to be represented. Besides computing initial layouts, such algorithms may either rearrange the whole diagram if a change of the underlying model occurs, or even preserve its overall structure and weave added elements smoothly into. On the textual side, serializers are required that are able to augment textual representations with line breaks and indentions according to syntax specific rules. Furthermore, textual representations of model parts that are untouched by refreshments shall retain their shape rather than get reformatted each time a refreshment takes place.

Those techniques are needed in order to bring model representations, either graphical or textual, into a usable and appealing form. This is necessary, since the transient view approach, as presented here, performs on the abstract syntax of models. Updating the concrete syntax representation is delegated to the particular rendering tools. Thus, the impact of the ideas developed within this thesis is directly dependent on the capabilities of such preparing techniques.

The state of the practice

The landscape of today's modeling tools is widely spread: There are integrated modeling environments that are used in enterprise application engineering such as MID's Innovator[5] and IBM's Rational Software Architect.[6] They provide rich editing facilities and various kinds of model representations. In the field of safety critical systems tools like MathWorks' Simulink,[7] Esterel Technologies' SCADE Suite,[8] or HIMA's ELOP II Factory[9] are established. They are focused on the generation of code that satisfies certain criteria with respect to soundness and reliability. Their modeling facilities are mostly characterized by a strict coupling of model content and representation. Customizing representations of models is usually not supported by those tools.

The spectrum is completed by slim editors that are built in order to obtain tools

[5] http://www.mid.de/en/products/innovator-modeling-platform.html
[6] http://www-01.ibm.com/software/rational/products/swarchitect/
[7] http://www.mathworks.com/products/simulink/
[8] http://www.esterel-technologies.com/products/scade-suite/
[9] http://www.hima.com/products/elop_ii_factory_default.php

supporting the use of tailored Domain-Specific Languages (DSLs). Such DSLs are mostly employed for the purpose of improving the software development process by means of the generation of schematic code based on abstract descriptions. Since these modeling languages are often used for the duration of certain software development projects, their life cycles and those of related tools are much shorter compared to the former mentioned ones.

Consequently, domain specific tools often lack advanced means providing efficient navigating and editing techniques. Modeling with those tools is rather archaic and cumbersome as stated by Fuhrmann and von Hanxleden [14]. The impact of these shortcomings grows with the complexity of the models. Therefore, much work has been done by researchers in order to improve several activities involved in the modeling process: In the graphical modeling field, an alternative method for creating and connecting elements has been proposed by Matzen [31]. Techniques for the configuration of model elements, i. e., the setting of properties such as name, type, attached elements, etc., have been developed by Scheidgen [43]. The problem of positioning graphical elements in graph-based diagrams has been investigated in depth. A collection of layout algorithms is proposed by Di Battista et al. [9]. Implementations have been contributed, e. g., by Spönemann [47], commercial tools are provided by IBM ILOG[10] or yWorks.[11] Related work in the textual world has been done as well, as discussed in chapter 2. These activities, however, cover only single representations of models, whereas the handling of multiple, use case-specific representations is not addressed yet.

1.3 Outline of this work

This thesis is structured as follows: After the introduction of modeling in the software development field and the placement of this work, related work on graphical and textual modeling frameworks is discussed. The chapter starts with a review of foundations on the creation of DSL tooling. It proceeds with a presentation of frameworks enabling the graphical formulation of models, and a comparison of various frameworks for creating text-based DSL tools. Afterwards, previous approaches of combining graphical and textual model representations are reviewed. In the subsequent chapter the technologies, which are used in the practical part of this work, are presented in some detail, since they form diverse restrictions and constraints on the solutions that are developed in this work.

chapter 4 starts with the investigation of previous approaches on the topic of this thesis. This includes a discussion of their advantages, their applicability in certain environments, and their disadvantages. Finally, the transient model representation approach is elaborated. Subsequent to investigating of the concepts, the reference implementation mentioned above is presented in chapter 5. That chapter starts with

[10] http://www.ibm.com/software/integration/visualization/jviews/graph-layout-eclipse/

[11] http://www.yworks.com/en/index.html

the brief introduction of the KIELER tool and the graphical modeling language that is supplemented by an appropriate textual representation. After presenting that textual syntax, the design of its integration with the existing tool is outlined. The chapter concludes with an explanation of the implementation and a discussion of open issues.

The application of the transient view approach in the context of the MENGES project is studied in chapter 6. Therefore, initial drafts of graphical representations of domain specific models, which adhere to textually formulated models, are proposed, as well as a lightweight method for obtaining them. Subsequently, use cases that are enabled by application of the transient view approach are discussed by means of the previously introduced views.

chapter 7 concludes this thesis and lists some topics for future work.

Chapter 2

Related work

The utilization of customized domain-specific languages has gained a lot of popularity in the recent decade and is common practice in these days. Hence, a lot of work has been done on investigating their characteristics and methodologies allowing the efficient employment of them in the software development processes.

As mentioned in chapter 1, the use of (external) DSLs is conjoined with a large amount of effort to be spent on the development of appropriate tooling tapping their full potential. Kleppe [26] identified a list of tasks that have to be performed in order to get a tool dedicated to the chosen DSL. Creating tooling for graphical DSLs involves:

1. Definition of the abstract syntax of the language.
2. Definition of the graphical concrete syntax, e. g., boxes, circles, polygons, and simple lines, colors, shapes, etc.
3. Specification and implementation of a transformation translating content given in the concrete syntax into the abstract syntax representation.
4. Creation of a graphical editor dedicated to the concrete syntax.

Adapting these steps to the process of creating tools for textual DSLs results in:

1. Definition of the abstract syntax of the language.
2. Definition of the structure of the textual concrete syntax forming data structures similar to a parse tree definition.
3. Derivation of a grammar from the concrete syntax structure supplemented with keywords, etc. that serves as input of a parser generator.
4. Specification and implementation of a transformation translating the concrete syntax into the abstract syntax including static analyses, cross reference binding, etc.
5. Creation of a textual editor dedicated to the concrete syntax that may be supplemented with syntax highlighting, content assist, etc.

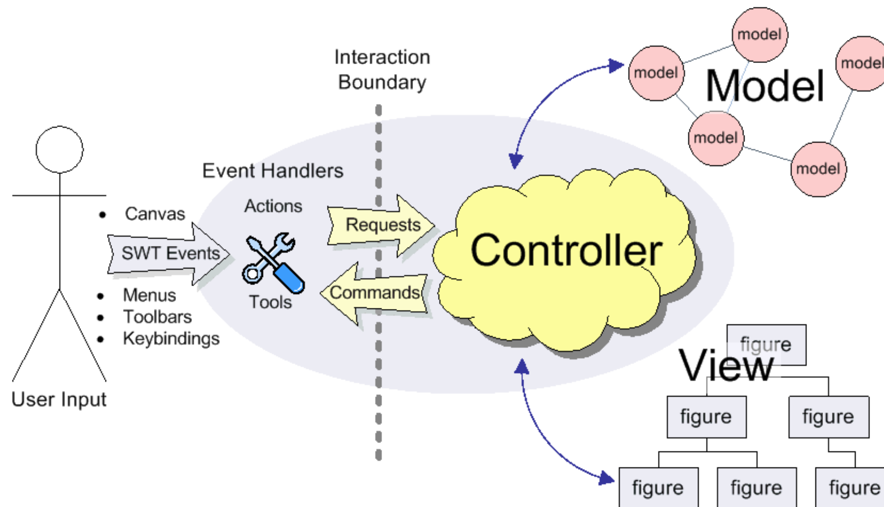


Figure 2.1: Model-View-Controller (MVC)-based implementation of GEF.[3]

The following presents some tools and frameworks for obtaining DSL-specific tools. Each of them is discussed briefly and examined according to the extent of assistance on performing the fundamental tasks. Afterwards, some attempts of combining graphical and textual modeling are reviewed.

2.1 Graphical Modeling

A framework that is established for a couple of years is the Graphical Editing Framework (GEF).[1] Its aim is to enable graphical editing in Eclipse.[2] This includes the supply of necessary mechanisms, e. g., drag and drop, and interfaces allowing the creation of tailored graphical editors with reasonable effort.

GEF-based editors implement the Model-View-Controller (MVC) pattern in a very strict way. The controller is realized by GEF's *Edit Domain*. This entity is in charge of updating the view(s) after the execution of *commands* on the model. Any kind of manipulation of the model is represented by such a command. This principle is outlined in Figure 2.1. Besides this kind of coordination of model modifications GEF does not impose any restrictions or requirements on the model, so this part can be implemented by any kind of data. Consequently, GEF does not capture any persistence issues at all.

The views are realized in form of graphical diagrams that consist of elementary or compound figures. To build up a graphical editor using GEF, these figures, i. e., the concrete syntax of the language, must be implemented manually. A collection of

[1] http://www.eclipse.org/gef/
[2] http://www.eclipse.org
[3] http://help.eclipse.org/helios/topic/org.eclipse.gef.doc.isv/guide/images/gefmvc.gif

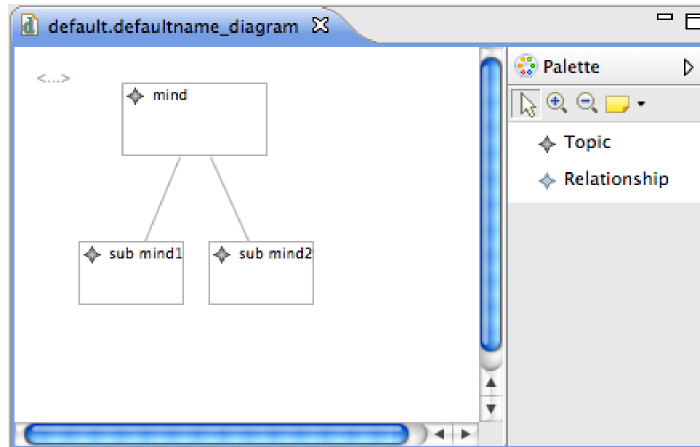


Figure 2.2: A simple GMF-based mindmap editor.

standard shapes is provided by a library called *Draw2D*.

According to the full custom coupling of the graphical editor and the related model, GEF does not impose any notion of abstract syntax. This implies that only item 4 of Kleppe's list of tasks is assisted by GEF.

The Graphical Modeling Framework (GMF) is a framework that is built upon GEF. However, in contrast to GEF, which provides the basics of graphical editing, GMF targets the big picture and paves the way for the model-driven development of graphical editors. Hence, GMF fills GEF's gaps concerning the supply of implementations of the concrete syntax as well as the whole persistence layer. A key property of GMF-based editors is the strict separation of semantic information according to the abstract syntax and notational information such as positions, sizes, and distances that are part of concrete syntax.

By means of GMF it is possible to create a rather simple graphical editor based on a given abstract syntax without writing a line of code. Instead, all of the crucial aspects are specified in an abstract, model based way. An example of such an editor is depicted Figure 2.2, the related source specification is shown in Figure 2.3.

GMF's workflow provides assistance on Kleppe's tasks almost completely. For defining the abstract syntax GMF leverages the meta modeling facility of EMF, which is presented in the next chapter. The concrete syntax is specified in a so-called *graph model*; the correlation between concrete and abstract syntax is defined in the *mapping model*. Exemplary instances of such models are shown in the upper half of Figure 2.3. The remaining tasks are covered by the runtime library and code generation facility.

Considering GMF's capabilities on integrating graphical and textual modeling one has to notice that this aspect has been neglected by the developers of GMF almost completely. Solely a rudimentary tokenizing mechanism for splitting labels of connections is available.

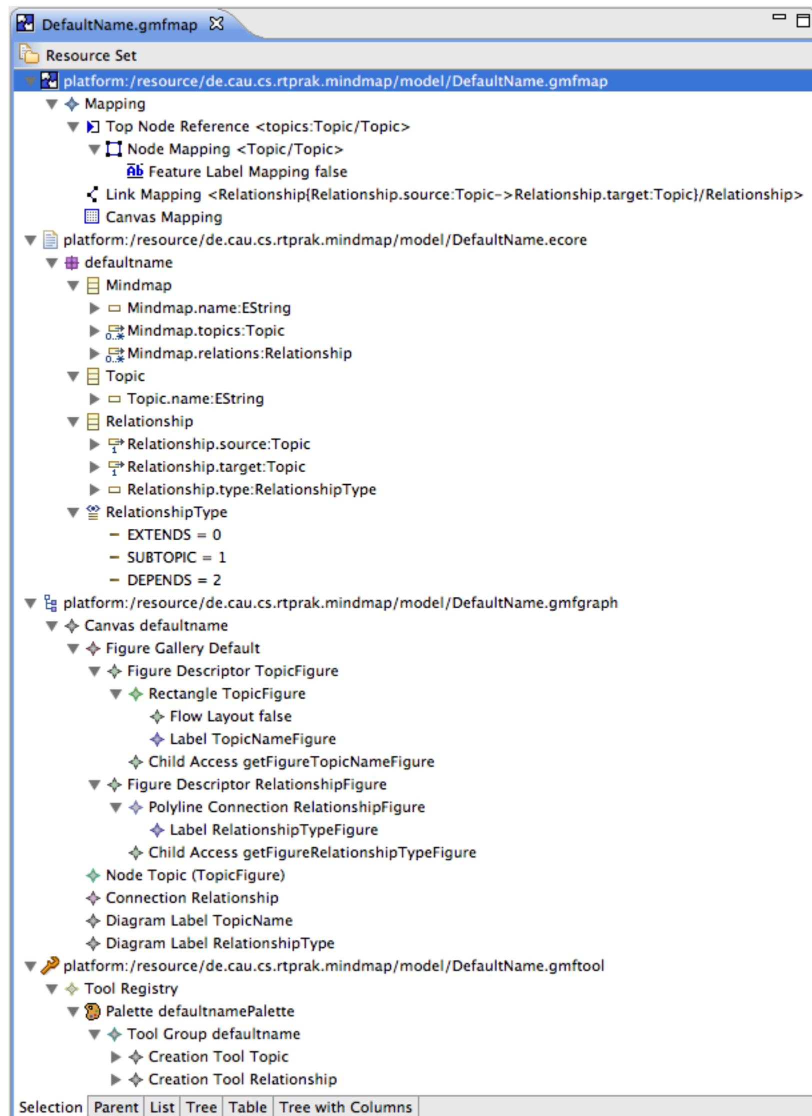


Figure 2.3: Specification of abstract and concrete syntax of a GMF-based editor.

2.2 Textual Modeling

Graphical representations of models are mostly convenient for documenting specifications and communicating solutions to various interest groups. However, editing graphical representations can be cumbersome as pointed out by Fuhrmann and von Hanxleden [14]. Furthermore, in certain situations, e. g., the generation of schematic code, the deployment of a graphical editor for describing the variable parts is just like cracking a nut with a sledgehammer. Further reasons for preferring textual languages to graphical ones, such as tool independence, speed of creation, and the suitability

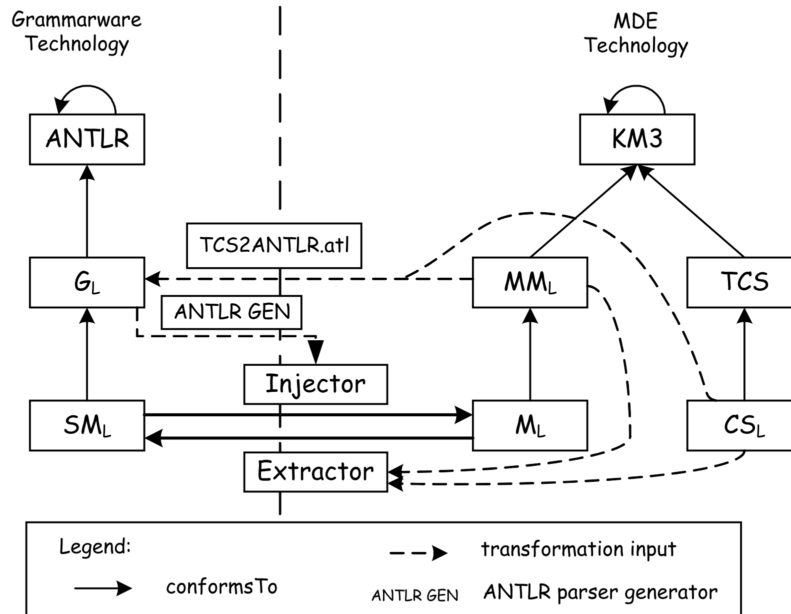


Figure 2.4: TCS and its context (Jouault et al. [24]).

for versioning, are stressed by Grönniger et al. [18]. Therefore, various approaches have been pursued to bridge the gap between textual editing and modeling.

The first one to be mentioned is the Textual Custom Syntax (TCS). This language addresses the relation of a concrete syntax to a given abstract syntax. TCS, developed by the ATLAS team [24], is embedded in the MDE framework ATLAS Model Management Architecture (AMMA).[4] It integrates with AMMA's meta-metamodeling language Kernel MetaMetaModel (KM3) and the ATLAS Transformation Language (ATL).

In order to develop a DSL with TCS the abstract syntax, also referred to as the metamodel of the language (MM_L), has to be provided in KM3. Subsequently, the concrete syntax of each element of the abstract syntax has to be formulated in TCS, resulting in a very tight coupling between the abstract and concrete syntax. In practice this usually means that compromises must be accepted on the concrete syntax side, or the abstract syntax must be adapted, respectively. Note that the abstact syntax of TCS is formulated in KM3, too.

The tools around TCS provide automatic generation of a so-called *injector* that is in charge of parsing the textual inputs and constructing objects according to MM_L. Serialization is also covered by the TCS tooling. In contrast to the full-custom injectors, a generic *extractor* is deployed in order to pretty print models. In addition, parameterizations for the Textual Generic Editor (TGE) are generated providing assistance while browsing the models (outline, hyperlinks, hovers). An

[4] http://wiki.eclipse.org/AMMA

overview on the components involved while working with TCS is given in Figure 2.4, which is taken from Jouault et al. [24].

The assessment of TCS reveals that each one of Kleppe's tasks on the way to a tailored editor for a textual DSL is covered by this language and its related tools. The definition of task 1 and 2 are formulated in KM3 and TCS. Task 3, 4, and 5 are performed by TCS's generator tool chain.

The topic of automating the generation of domain specific tooling has been investigated by Krahn et al. [28], too. They propose a framework called MontiCore that is basically intended to serve the same purpose. However, compared to TCS, assumptions and focus differ significantly from the former approach. Krahn et al. state that reasons for separating abstract and concrete syntax play a minor role in practice. Hence, the MontiCore grammar specification language is designed such that the abstract syntax can be derived directly from the custom syntax. Consequently, only one syntax definition must be created.

Besides, MontiCore aims at improving the language design with respect to compositionality. This means languages may be extended (*Language extension*) or restricted (*Language specialization*) [29]. In order to satisfy these use cases MontiCore has been equipped with a multiple-inheritance mechanism as well as a concept of *abstract language definitions*. Abstract languages may contain gaps that are supposed to be closed by concrete derivatives of such languages.

Generating a full-custom editor equipped with syntax highlighting, validation, content assist and further features is also a major contribution of MontiCore [27]. However, the lack of a serialization facility is a disadvantageous characteristic of this framework. With respect to Kleppe's list of tasks only task 2 has to be done manually, while all the other artifacts are generated by the tool chain.

While the definition of a concrete syntax starting with blank a document is the central part of work in the former approaches, the procedure of obtaining an editor for a textual DSL has been turned over by Heidenreich et al. [21]. They state that refining an existing syntax is much easier then creating one from scratch. This idea has been realized in EMFText.

Designing a textual DSL with EMFText requires the specification of the abstract syntax formulated in EMF's metamodeling language called *Ecore*. Based on this an initial concrete syntax definition that conforms to the Human-Usable Textual Notation (HUTN) standard [40, 35] is derived.

Beyond the initial grammar proposal, EMFText provides automated generation of the related parser, printer, and a tailored Eclipse-based editor. The process of creating the infrastructure for a DSL by means of EMFText and its related artifacts are shown in Figure 2.5, taken from Heidenreich et al. [21].

Similar to TCS task 1 and 2 of Kleppe's list have to be done manually while working with EMFText. The remaining ones are performed by the framework.

A notable case study on the power of EMFText is presented in Heidenreich et al. [22], called Java Model Parser and Printer (JaMoPP). In this study the authors created a Java 5 compliant toolchain enabling to treat Java source code as models.

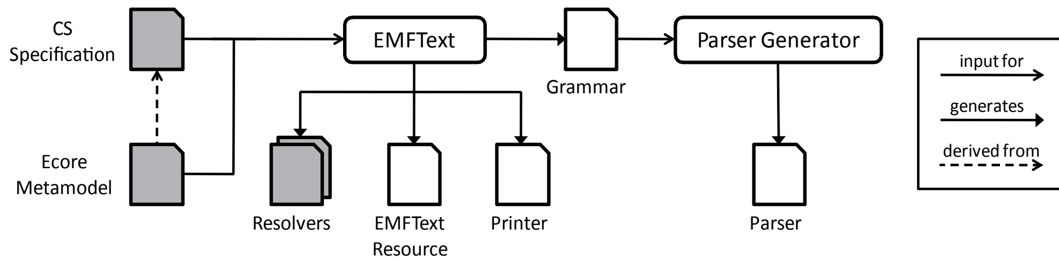


Figure 2.5: Overview on the artifacts of EMFText (Heidenreich et al. [21]).

The last framework to be mentioned in this row is Xtext,[5] which is superseding the former open Architecture Ware (oAW) Xtext. It has been developed since 2008 and unifies most of the key benefits of the former approaches. For instance, the syntax definition of a DSL might be distributed over several grammar specifications implying an improved reuse of such language fragments. Another example is the abstract syntax definition. If the DSL shall be (partially) built upon a given abstract syntax, Xtext allows to import this definition, otherwise an abstract syntax definition is synthesized. Hence, depending on the given situation, either task 2 or both task 1 and 2 have to be performed by the developer, while the tasks 3, 4, and 5 are accomplished by the tool chain. Since Xtext is explained in more detail in chapter 3, further characteristics are omitted here.

There are several further so-called Language Development Kits (LDKs) that have been compared by Goldschmidt et al. [17]. As most of them appear to be obsolete by now, the discussion of those frameworks is omitted.

A framework that redefines the notion of text documents is the Meta Programming System (MPS)[6] introduced by JetBrains. MPS is a language development toolkit that covers the whole process of working with DSLs starting with the definition of the abstract and concrete syntax, creation of code generator templates, and also the creation and maintenance of specifications in the DSL.

In contrast to the former approaches pursuing a parsing and printing strategy, MPS persists the abstract syntax of the document content. When such a document is loaded, the content is displayed in a so-called *projectional editor*. The concept of the projectional editor is adopted from the graphical editors that usually project content formulated in an abstract syntax into graphical representations such as boxes, ellipses, and arrows, as pointed out by Völter [53].

The projectional editor realized in MPS is basically a tree editor (such as depicted in Figure 2.3) that is rendered in the shape of a text editor. Consequently, editing a document in MPS is like editing a tree-shaped data structure rather than writing down a text document. This is outlined in Figure 2.6. The opened document

[5] http://www.eclipse.org/Xtext/
[6] http://www.jetbrains.com/mps/

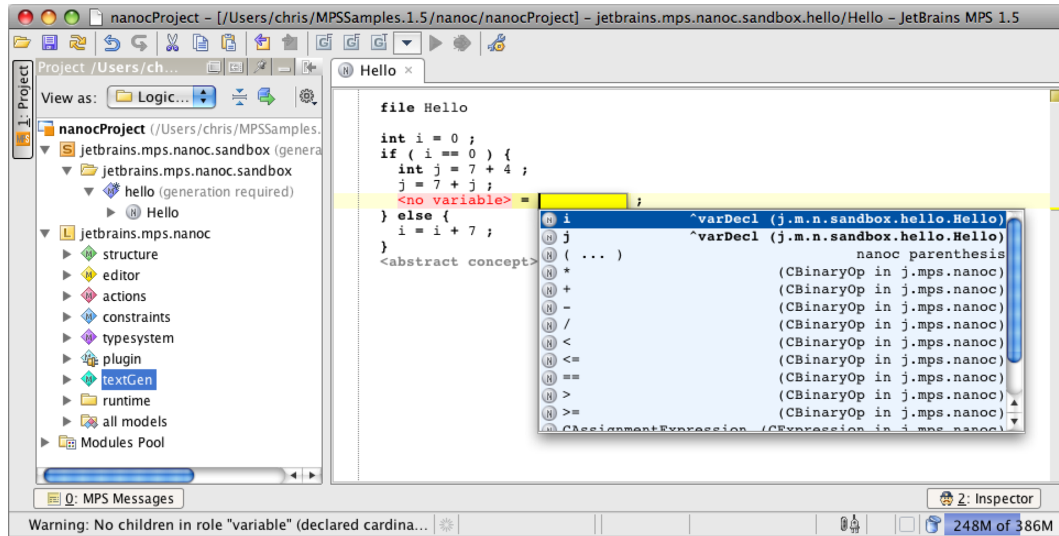


Figure 2.6: Projectional editor of the Meta Programming System (MPS).

contains a code fragment specified in a language called nanoC. This DSL contains variable declarations, variable assignments, compound expressions, and conditional statements.

In order to insert a statement, e. g., a variable assignment such as shown in Figure 2.6, a structure-based operation inserting the related abstract syntax element has to be performed, here the insertion of the assignment statement. Subsequently, the left hand side as well as the right hand side of this statement have to be filled.

In conclusion, the MPS framework assists the language developer while describing the abstract and concrete syntax of a textual DSL, i. e., performing tasks 1 and 2 of Kleppe's list. The projection approach results in the absence of any parser and a strict one-to-one mapping of the abstract and concrete syntax making tasks 3 and 4 superfluous. Task 5, i. e., the creation of the editor, is performed by the MPS tool.

2.3 Combination of Graphical and Textual Modeling

It is widely accepted that graphical and textual modeling are not alternative but rather complements (Heidenreich et al. [21]). However, only few researchers investigated this field as yet.

Wischer [55] describes various ways editing models. He extended the Kiel Integrated Environment for Layout (KIEL) StateCharts browser with facilities enabling the direct editing of the abstract syntax of the model (*Structure-based editing*) as well as modifying the model via changing a textual representation. He equipped KIEL with an editor exhibiting a textual representation of the whole model. Figure 2.7, taken from Wischer [55], shows the KIEL tool providing different views on an exemplary model.

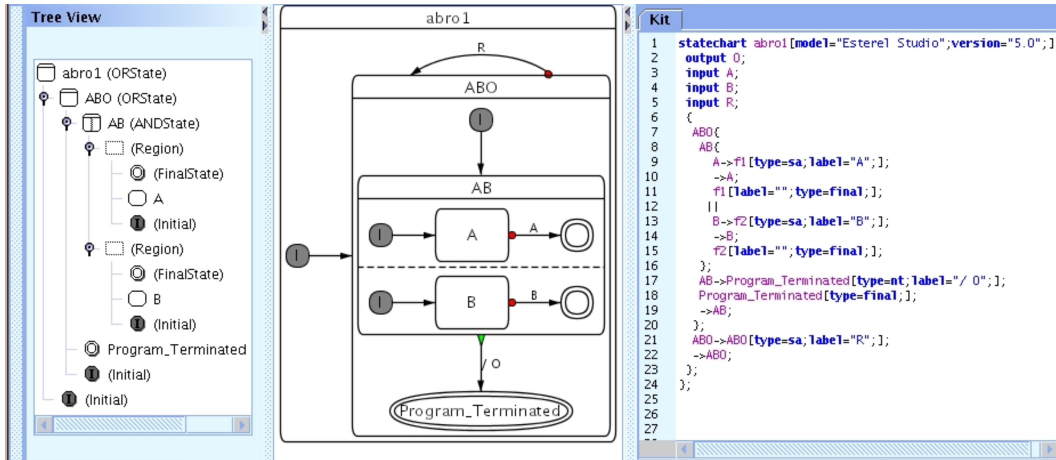


Figure 2.7: Model representations in the KIEL tool (Wischer [55]).

In order to establish the integration of those views, KIEL's model managing component provides an interface allowing the viewers, e. g., the textual editor, to read and display the model. If the textual view is changed by the modeler the editor component creates a new model, i. e., a new Abstract Syntax Tree (AST). This new model is handed over to the model managing component subsequently. Lastly, the viewer components are informed about the replacement of the model in order to update their views according to the MVC paradigm.

Bayramoğlu [4] carried this idea into the worlds of GMF and Xtext. She installed a graphical and a textual editor, which are are working on the same document, side by side. They are synchronized while invoking the save action of the editor the modification has taken place in. However, this approach suffers from certain issues that are investigated in chapter 4.

The principle of extending graphical editors, e. g., based on GMF, such that they support the editing of textual portions of models as comfortable as common textual editors do is described by Scheidgen [42, 43]. He proposes the Textual Editing Framework (TEF). Besides enabling the textual editing of models as described in the previous section, TEF provides hovering textual editors that can be attached to basically any kind of editor working on EMF models. By means of this method modeling parts that are tiresome to perform in the graphical world, e. g., setting of parameters of entities, can be done very elegantly in textual form. Figure 2.8 outlines this technique.

The last work to be mentioned here has been contributed by Andrés et al. [3]. They stress that the increasing complexity of systems to be modeled necessitates the splitting of their specifications into smaller parts leading to increased comprehensibility and maintainability. Instead of partitioning DSLs into DSL families with both distinct abstract syntaxes as well as distinct concrete syntaxes, Andrés et al. propose

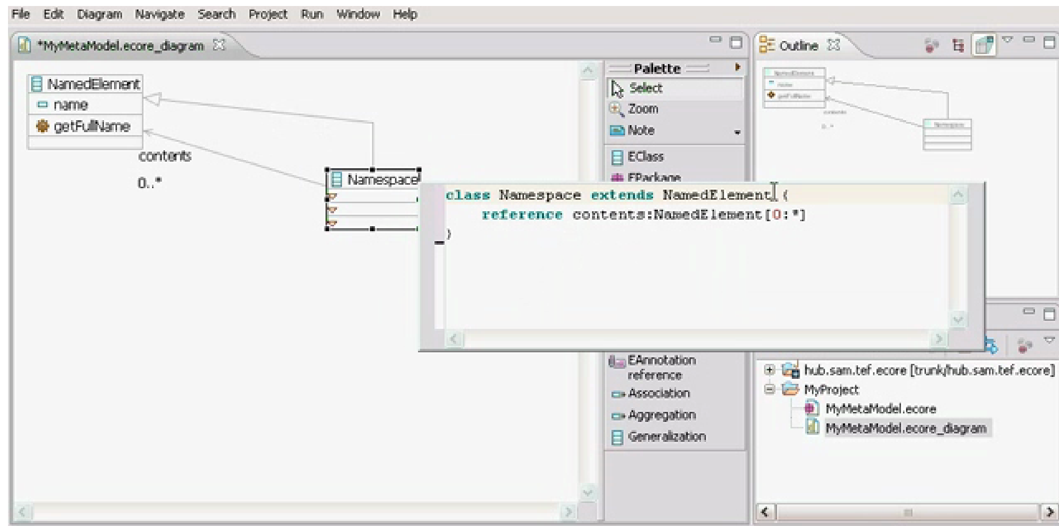


Figure 2.8: Hovering textual editor provided by TEF.[7]

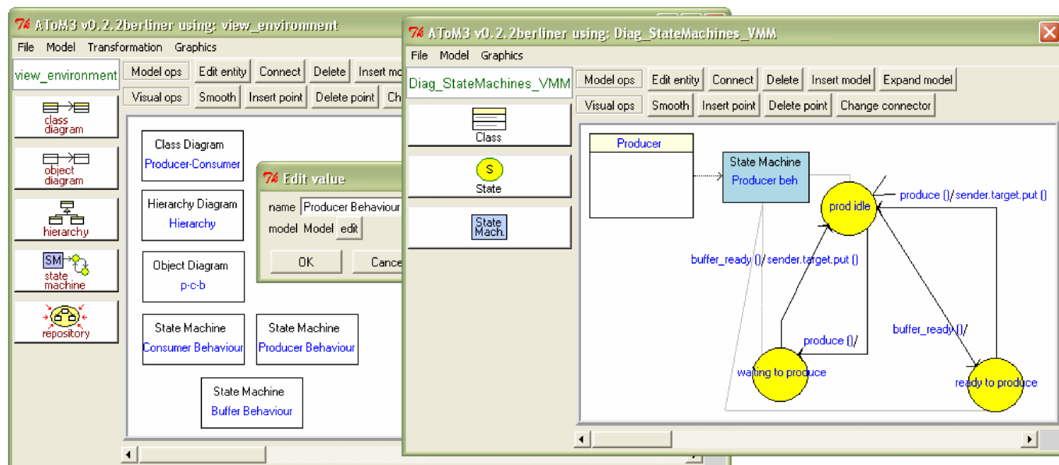


Figure 2.9: Multi-View-DSLs in AToM3 (Guerra & de Lara [19]).

the employment of *Multi-View-DSLs*.

Such DSLs are combined of a family of various graphical and/or textual concrete syntaxes in order to describe each aspect of a system in the most convenient and effective notation. A well-known example of Multi-View-DSLs is the UML that encompasses several diagram types, as introduced in chapter 1. Andrés et al. call these diagrams *viewpoints* and define them to be projections of the overall abstract syntax.

The Multi-View-DSL approach, introduced by Guerra and de Lara [19], has been

[7] http://www.informatik.hu-berlin.de/~scheidge/tef/tefgraph.divx

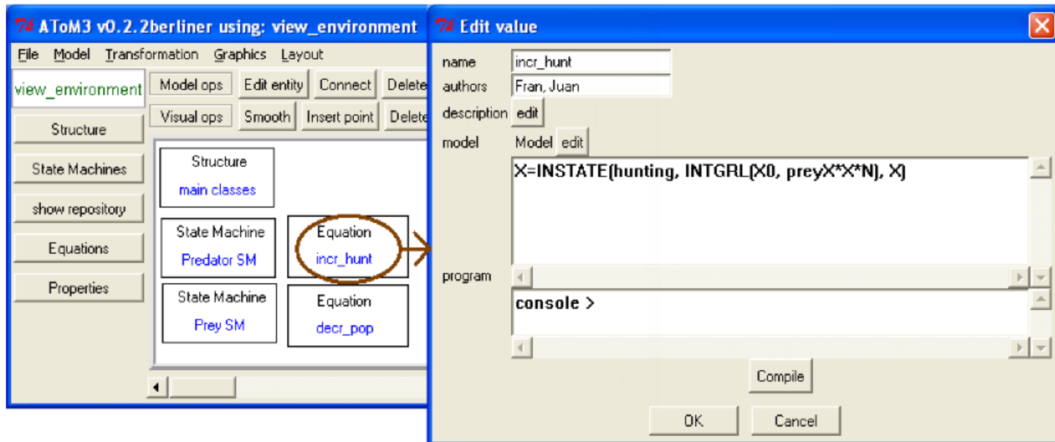


Figure 2.10: AToM3 equipped with textual views (Andrés et al. [3]).

implemented in a test bed called A Tool for Multi-formalism and Meta-modelling (AToM3), see Figure 2.9. The window on the left is the main window and forms the top level view on the system. On its left it provides certain viewpoints, i. e., diagram types, for describing a system in the given Multi-View-DSL (here a subset of the UML). The canvas of the main window shows various specifications formulated in some of the available diagrams. The window on the right of Figure 2.9 depicts the specification of an aspect of the system formulated in an UML State Machine-like formalism. A Multi-View-DSL leveraging textual viewpoints is shown in Figure 2.10, taken from Andrés et al. [3].

Chapter 3

Used Technologies

This chapter presents the tools and frameworks used to implement the solutions proposed later on in this work. Though some of them have already been mentioned in chapter 2, the aim of this chapter is to introduce them in some (technical) detail and to elaborate their crucial characteristics as far as they are relevant for this work.

3.1 The Eclipse Project

The Eclipse project has been introduced by IBM™. It is an open source software application running on the Java Virtual Machine (JVM). In its early days it has been conceptually designed to be an advanced and extensible Integrated Development Environment (IDE) for the Java programming language.

With the version 3.0 Eclipse has been restructured into an open component-based software system. The resulting Eclipse Rich Client Platform (RCP) solely consists of a couple of basic platform components such as the workbench core, the help center, the update mechanism, bindings to the file and window system, and a minimal runtime kernel.

The component model realized in Eclipse is the OSGi Service Platform that has been developed by the OSGi Alliance. The Eclipse implementation of this specification is called Equinox.

The basic element of the OSGi Service Platform is the *bundle*. Bundles may

- export runnable code,
- import code of other bundles, and
- define dependencies to other bundles.

Eclipse bundles are called *plugins*. In addition to the OSGi standard, plugins may

- define *extension points*, and
- contribute *extensions* according to available extension points.

Figure 3.1, taken from Fuhrmann [11], visualizes these relations between Eclipse plugins.

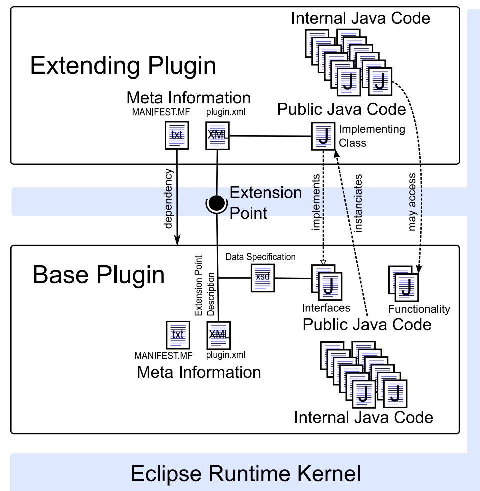


Figure 3.1: Relations of Eclipse plugins (Fuhrmann [11]).

Based on these foundations a diverse landscape of development toolkits for miscellaneous programming languages as well as enterprise applications has arisen.[1] There is also the possibility of creating so-called Rich Client Applications (RCAs). Such applications solely consist of the runtime kernel and those plugins implementing the desired features.

While working with Eclipse one may observe certain principles concerning the ergonomics. One that is crucial for this work is the distinction of Eclipse's user interface elements. Besides the typical menu, tool, and status bars Eclipse provides *editor parts* and *view parts*.

Editor parts, usually referred to as *editors*, represent content of files that *are* persisted anywhere on the hard disk. This is opposed to usual application software like word processors or drawing tools. Those tools use to provide empty documents that may be edited arbitrarily and rejected without leaving any traces on the disk. Eclipse editors cannot be opened without any given input.

View parts, in short just *views*, may display any data in any shape. They are not linked to files as strictly as editors. Views are rather used to show additional information concerning the document of the currently active editor or provide overview representations. Furthermore, views are employed to serve as file browsers or exhibit further information. Figure 3.2 provides an overview of the Eclipse workbench and denotes editor and view parts.

Considering the popularity of Eclipse and its spreading in the software developer sphere it is not surprising that the modeling ideas have been carried into it. Around the Eclipse Modeling Framework (EMF), which is the central modeling facility in Eclipse, a bunch of frameworks came up as shown in Figure 3.3 (Fuhrmann [11]).

[1] http://www.eclipse.org/community/example_rcp_applications_v2.pdf

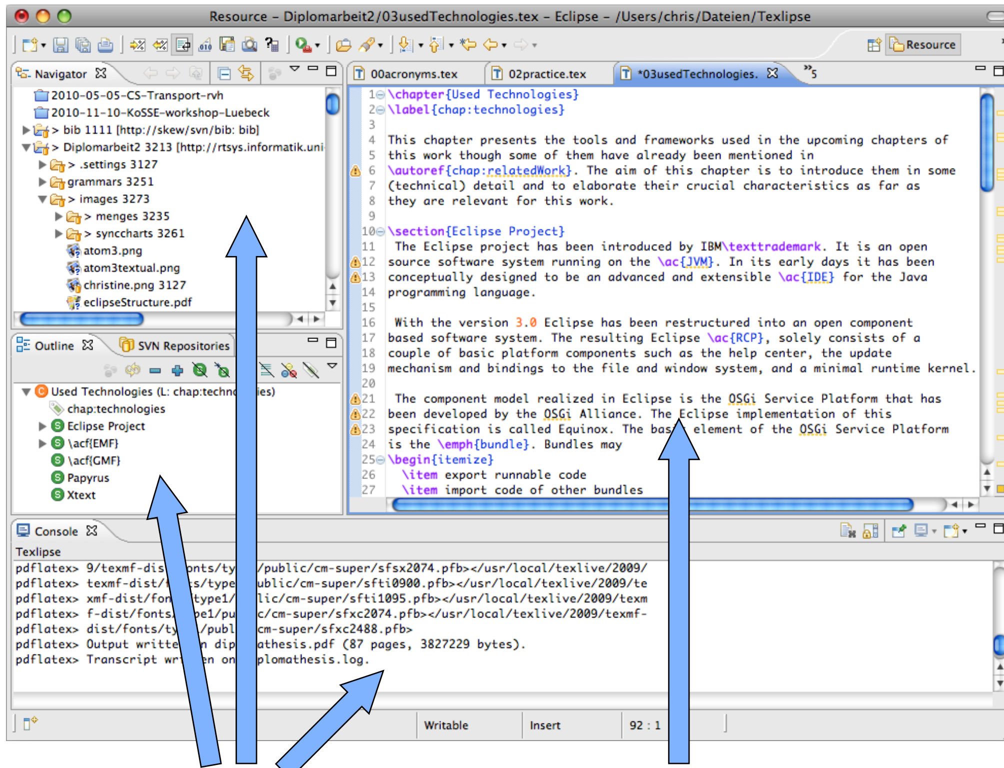


Figure 3.2: The Eclipse Workbench.

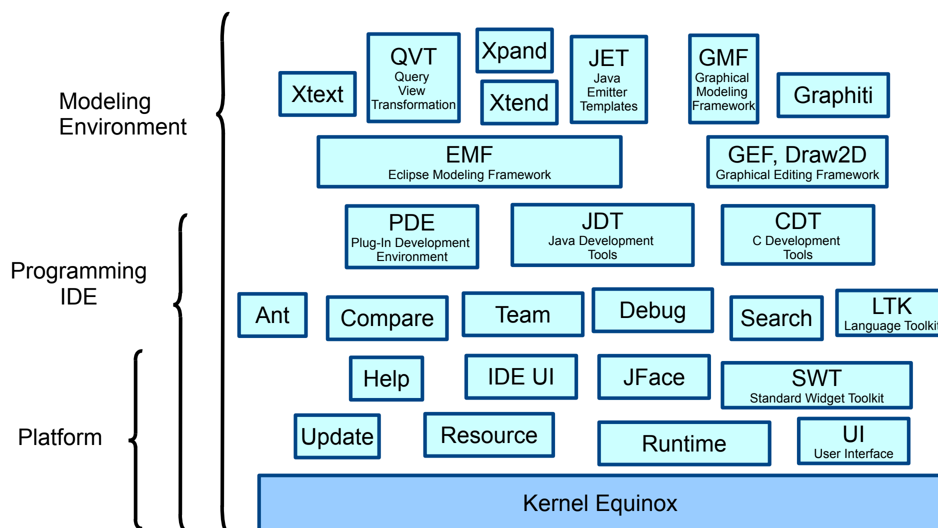


Figure 3.3: Components of Eclipse (Fuhrmann [11]).

3.2 Eclipse Modeling Framework (EMF)

The primary aim of EMF is to enable the specification of abstract syntaxes and handling of instances of them. According to OMG's MOF architecture (Figure 1.2) abstract syntaxes are referred to as *metamodels*, and instances of them are called *models*. Models may be composed programmatically or created by means of a tree-based editor, one of the artifacts provided by the framework. EMF enables further basic operations such as persisting and loading, examining and manipulating models. It can be considered as a bridge between modeling, i. e., describing things, and programming.

Application fields of EMF-based models are the abstract specification of entities or processes, e. g., in order to generate code, the transport of information, the persistence of information, or the configuration of software components. A prominent example of the last item is the redesign of the Eclipse workbench in Eclipse 4 (e4). In order to facilitate contributions to the workbench and increase the application designers' degree of freedom the whole user interface has been specified by means of EMF.[2]

EMF consists of 3 major parts: (1) the *metamodeling facility*, (2) the *code generator*, and (3) a *runtime framework*. The metamodeling facility provides editors for creating metamodels. They are formulated in a language called *Ecore* that is similar to OMG's EMOF language mentioned in chapter 1.

Ecore (meta)models usually consist of *classes* that are in relationship with each other. Similarly to UML classes, Ecore classes are named entities that may have attributes and operations. Possible relations between classes are the generalization, the aggregation, and the association. In addition to classes, Ecore supports interfaces and custom data types that are aliasing standard types such as `int` or refer to custom enumerations, which can be defined in Ecore models, too. For the purpose of structuring larger Ecore models classes, interfaces, data types, and enumerations are assigned to packages. Figure 3.4 shows a simple metamodel expressed in Ecore in EMF's tree-based and graphical editors.

Reusability is also addressed by Ecore. Metamodels formulated in this language may refer other metamodels in order to leverage their elements. Thus, languages defined by their Ecore-based abstract syntax formulation may be composed of other languages or language fragments.

In addition to describing a language by creating a metamodel manually, EMF's metamodeling facility allows the import of definitions given in various formats such as XML Schema,[3] Rose,[4] or plain Java classes.

It remains to note that Ecore has been bootstrapped, i. e., it is described in itself. Hence, it is a valid M3 language according to the MOF architecture.

The task of EMF's code generator is to provide Java implementations of the data

[2] http://www.slideshare.net/LarsVogel/eclipse-e4-tutorial-eclipsecon-2010-3587897

[3] http://www.w3.org/XML/Schema

[4] http://www.ibm.com/software/awdtools/developer/datamodeler/

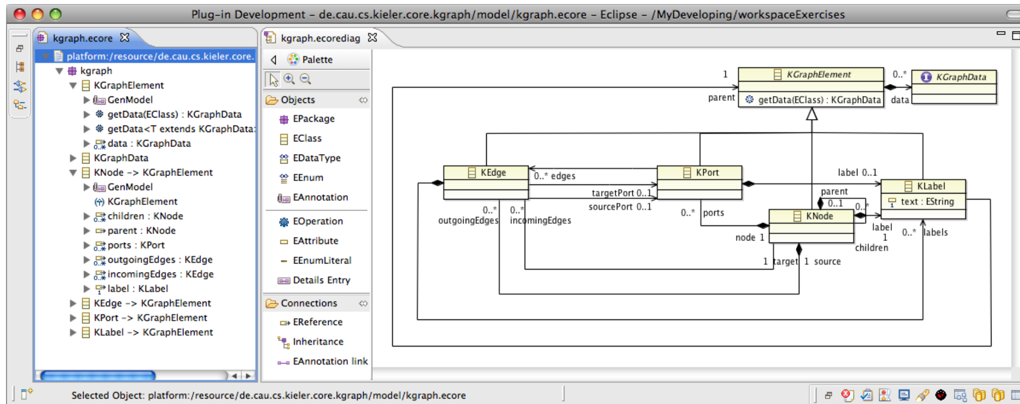


Figure 3.4: EMF's tree-based and graphical Ecore editors.

structures specified in the metamodel. This comprises code implementing the data structures themselves, code for a customized tree-based editor, and various helper classes. A characteristic worth to note is EMF's strict realization of the *factory design pattern*. Consequently, factory classes related to the Ecore package are generated. Further details on EMF's code generation can be found in Steinberg et al. [49].

The runtime framework contributes a lot of functionality to the Java implementations provided by the code generator. This includes

- helper methods providing, e. g., efficient navigation on models,
- a notification mechanism,
- a reflective Application Programming Interface (API), and
- the implementation of a resource concept allowing EMF-based models to be addressed, loaded, and persisted.

The EMF core is supplemented by a bunch of additions, for example

- the EMF Validation framework,
- the EMF Transaction framework,
- the EMF Query framework, and
- the EMF Compare framework.

This outline of EMF concludes with some notes concerning the resource concept. EMF resources are similar to file descriptors. They are aware of a location a model is to load from or to save to. In addition, resources encapsulate the format, e. g., XML Metadata Interchange (XMI), and the encoding of the persisted models like ASCII or UTF-8.

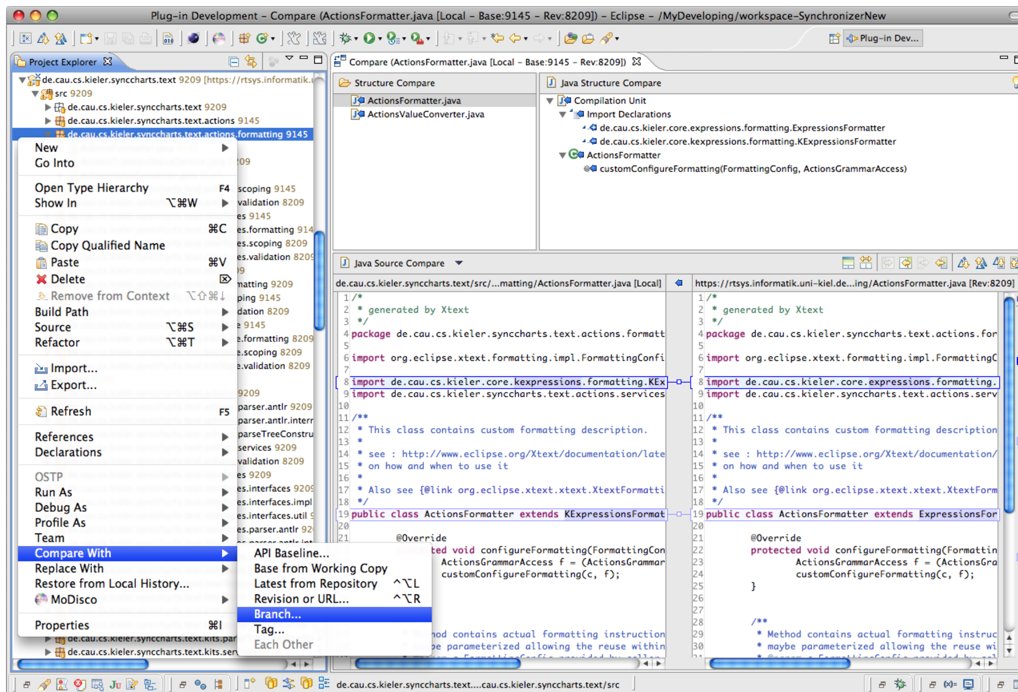


Figure 3.5: Eclipse's Compare tool.

In order to save a model to a certain location, a corresponding resource has to be requested from the framework. If this request succeeds and the resource is provided, the model can be attached. Note that a model can be attached to at most *one* resource. This is because the relation between resources and models is a containment association similar to the aggregation between classes. Hence, attaching the model of resource A to resource B will remove it from A and vice versa. The same holds for attaching a model of a resource to another model or put child elements of a model into a separate resource. This fact affects the solutions presented in the following chapters of this work significantly.

3.3 EMF Compare

A beneficial feature that Eclipse contributes to Integrated Development Environments (IDEs) is the Compare tool. It supports the comparison of almost any kind of artifacts such as text files, source code, images, jar files, etc. Therefore the Compare editor, shown in Figure 3.5, and necessary infrastructure are supplied by this tool. In line with most of the Eclipse components it provides a simple interface allowing to contribute specialized compare engines and visualizers of differences.

The EMF Compare project proposed by Toulmé [52] paves the way for employing the Eclipse Compare tool in the modeling environment. It enables the comparison of

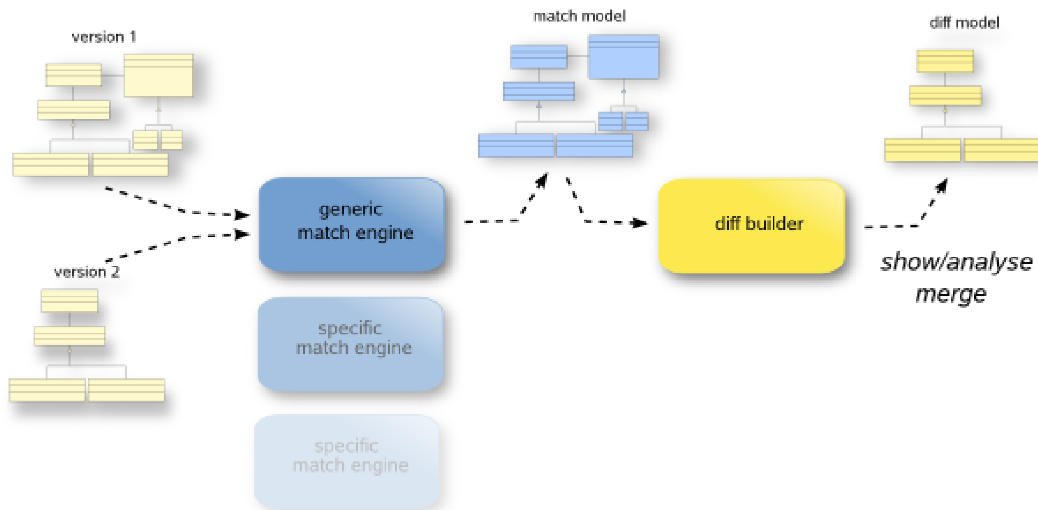


Figure 3.6: The process of comparing models by means of EMF Compare.[5]

different models as well as different versions of the same model. Schipper et al. [44] leveraged this framework for the development of a difference method of graphical representations of models.

Besides comparing *pairs* of models or pairs of versions of models, EMF Compare facilitates the *three-way-comparison* as pointed out by Mülder et al. [33]. The ability of comparing three models is the key enabler for using a compare engine in version control systems. The typical scenario looks as follows: (1) a local working copy of a model checked out of a repository and (potentially) modified, (2) the revision in the repository that has been checked out once, and (3) the head version of the model in the repository must be compared in order to preclude the emergence of inconsistencies while committing or updating modifications.

Comparing models by means of EMF Compare involves two major steps: (1) find pairs/triples of matching model elements and (2) determine differences by means of the unmatched model elements. The first one is performed by a match engine producing a match model. The second one is performed by a diff engine, respectively. Based on the match model it computes a dedicated difference model. This process is depicted in Figure 3.6.

Although EMF Compare is shipped with generic match and diff engine implementations it is designed to be highly configurable allowing metamodel designers to register tailored match and diff engines. In order to coordinate and encapsulate the selection of the right engines utility classes named `MatchService` and `DiffService` are provided by the framework.

[5] http://wiki.eclipse.org/images/7/7b/CompareGlobal.png

```
// Matching model elements
MatchModel match = MatchService.doMatch(left, right, Collections.emptyMap());

// Computing differences
DiffModel diff = DiffService.doDiff(match, false);

// Merges all differences from left to right
MergeService.merge(diff.getOwnedElements(), true);
```

Listing 3.1: Comparing and merging models programmatically.

EMF Compare's range of functionality is completed by its ability to merge models. Merging models denotes the process of unifying them element per element in a given direction. As merging may only take place between two models, EMF Compare calls these models *left* and *right*. Hence, merging from left to right denotes creating, removing, and modifying elements of the right model such that it conforms to the left one when the process terminates. Elements of the right model that have a counterpart in the left one are preserved.

Merging of models is handled by the third utility class called `MergeService`. Besides selecting the best fitting merger with respect to the metamodels of the left and right model, this service is in charge of selecting the correct one according to various types of differences that may occur. With respect to a pair of left and right model these are

- the absence vs. presence of a model element,
- a differing order of children elements with respect to a containment association in the metamodel,
- the absence vs. presences or mismatch of a model element's attribute value,
- the absence vs. presence or differing target of a reference to another model element (non-containment association), and
- a differing order of references to elements with respect to a non-containment association in the metamodel.

Listing 3.1 shows the programmatical utilization of EMF Compare in order to align the right with the left (indicated by the Boolean parameter of the method `merge`).

3.4 Xtend

Xtend is a functionally oriented model to model transformation language that integrates with EMF. It originates in a part of the former code generation framework open Architecture Ware (oAW). The aim of the predecessor of today's Xtend was to enable the formulation of advanced queries on model elements while generating code, thus, to *extend* the set of operations provided by the model elements' classes.

Today it is situated in the Xpand code generation framework of the Eclipse Model To Text (M2T)[6] project and can be used separately.

The Xtend language provides the typical arithmetic and Boolean operations programmers are used to as well as built-in operations for standard types such as `toUpperCase()` for instances of the class `String`. In addition it provides a bunch of collection-specific operations such as `select`, `reject`, `forAll`, and `exists`.[7]

Besides support for gathering and preparing information out of models, Xtend has been equipped with the following features enabling actual model translations and transformations:

- the `new` operator allowing to create new model objects,
- the `let` construct in order to preserve objects temporarily,
- the ability to cause side effects via `set` or `add` methods,
- the chain expression `->` enabling the concatenation of statements, and
- an escape mechanism delegating to Java methods.

According to the functional concept of Xtend a model transformation specification is separated into functions that are called *extensions*. Extensions usually take typed arguments and provide a return value or object. The return type can be omitted in most cases as it will be inferred at runtime.

For convenience and due to the lack of look-up mechanisms such as hash tables, variants of extensions have been introduced: *cached* extensions and *create* extensions.

In contrast to standard extensions, which are strictly executed each time they are called, cached and create extensions are executed at most once for a given list of parameters. The result returned to the caller extension is cached. Further calls of the extension with the same list of parameters reveal the cached result rather than recomputing it. The subtle differences between cached and create extensions are explained in the official reference[8] and therefore omitted here.

Xtend comes with a dedicated editor supporting error handling and content assist. Xtend specifications are interpreted by a runtime engine called XtendFacade that is initialized with the location of the chosen extension file, the name of the root extension, and the runtime model to perform on. Persisted models may be transformed by means of the Modeling Workflow Engine (MWE) delegating to the XtendFacade.

Compared to other model to model transformation frameworks such as the ATLAS Transformation Language (ATL) or Query/View/Tranformation (QVT), Xtend is very flexible as pointed out by Matzen [31]. Besides in-place transformations, i. e., performing side effects on a given model, Xtend allows cross metamodel transformations. This is explained in detail by Matzen as well.

[6] http://www.eclipse.org/modeling/m2t/

[7] http://help.eclipse.org/helios/topic/org.eclipse.xpand.doc/help/r10_expressions_language.html

[8] http://help.eclipse.org/helios/topic/org.eclipse.xpand.doc/help/Xtend_language.html

3.5 Graphical Modeling Framework (GMF)

As mentioned in chapter 2 GMF is a framework for constructing graphical editors of DSLs. It leverages EMF, which contributes the abstract syntax of the DSL as well as the persistence issues. The drawing of graphical representations of models is delegated to the Graphical Editing Framework (GEF), which is introduced in chapter 2, too.

At this point technical details of GMF that impose constraints on the solutions developed within this work are examined. This basically concerns GMF's nature of separating abstract and concrete syntax and the way GMF-based editors establish this principle.

Graphical representations of elements provided by GMF-based editors are composed of default shapes such as rectangles, ellipses, and connection arrows. These *figures* may be decorated with text fields and further shapes in order to reflect the complete amount of information given in the related element of the *domain model*, which is an instance of the abstract syntax. For the purpose of coordinating the life-cycle of the figures an abstraction layer consisting of *edit parts* has been installed. Thus, each element in a *diagram*, the graphical representation of a model, is represented by a related edit part.

Besides the content of the domain model, a diagram contains more information. The most important part of this *secondary* information is the layout of the elements in the diagram. In addition, diagrams may show either a restricted set of domain model elements or even elements of related domain models. This information needs to be persisted as well and is captured by a further EMF-based model called *notation model*. While the appearance of the concrete syntax elements of a GMF-based diagram is mostly hardcoded in the editor, the notation model is used to preserve information such as

- which elements are shown in the diagram,
- which children do they have,
- what are the parameter values of the related figures (position, size, bend points), and
- what are the related elements in the domain model.

Figure 3.7 outlines the layers of objects maintained by GMF-based editors.

In order to coordinate the access on both the domain model and the notation model, GMF-based editors exploit the EMF Transaction framework. Accessing these models for the purposes of modifying them is subjected to a so-called *transactional editing domain*. Each modification must be encapsulated in a *command* and subsequently executed by a *command stack*, which is provided by the editing domain. There are many predefined commands performing elementary operations; complex modifications can be "composed" by means of a compound command. Benefits of this approach are the avoidance of obtaining inconsistencies between the domain and notation model and, even more importantly, the contribution of proper *undo/redo* handling.

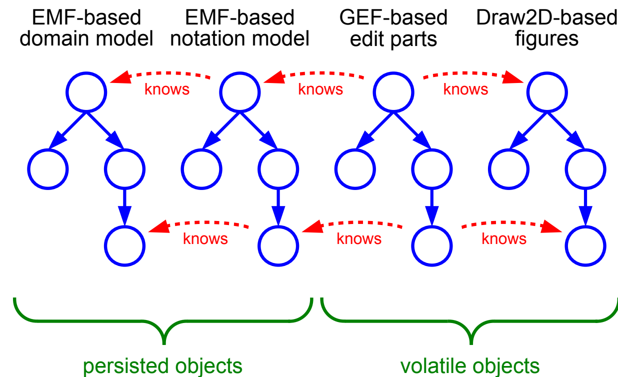


Figure 3.7: Data structures employed by GMF-based editors.

A further principle of GMF-based editors is called *canonical editing*. It denotes the strict adherence of a *bijective element-representative-relation*, meaning that each element added to the domain model will be represented in the diagram, and vice versa. The same holds for the deletion of elements. This strategy is implemented by a *canonical edit policy*.

The concept of edit policies goes back to GEF, too. It allows to determine the features of edit parts on a very abstract level. The edit policy concept is tightly coupled with the *request for command* concept and GEF's command mechanism (not to confuse with EMF transaction commands). Further edit policies are the *resizable edit policy*, the *XY layout policy*, and the *drag and drop edit policy*.

The canonical edit policy is both a blessing and a curse. On the one hand it paves the way for structure-based editing as worked out by Matzen [31]. On the other hand it prevents the creation of tailored diagrams restricted to parts of the entire model. Therefore, popular GMF-based editors come without canonical edit policies. Examples are the graphical Ecore editor (Figure 3.4) and the UML editors of the Papyrus tool chain.

3.6 Papyrus

The mission of the Papyrus[9] project is the development of advanced open source UML tools. According to Gérard et al. [16] these tools are positioned to be an alternative to commercial design environments for businesses who are not able to afford high quality tools with long-term support. The lack of flexibility with respect to extensibility of proprietary tools is another issue to be addressed by the Papyrus designers. This way a platform allowing to investigate new methods and technologies shall be provided to the community and academia.

[9] http://www.eclipse.org/modeling/mdt/papyrus/

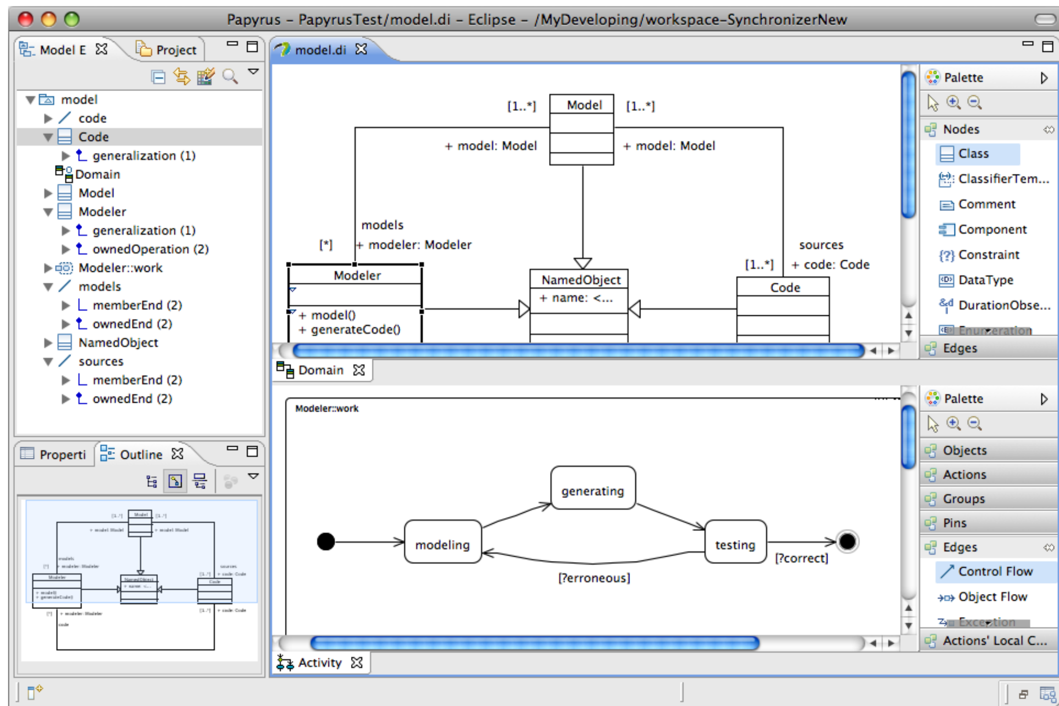


Figure 3.8: The Papyrus modeling environment.

The Papyrus UML editors are based on GMF and the Eclipse UML infrastructure provided by another section of the Eclipse Model Development Tools (MDT) project.[10] The first contribution is a dedicated *model browser* that is similar to those available in typical UML tools such as ArgoUML,[11] Together,[12] or MagicDraw.[13] It is supposed to abstract the file-based organization of data in Eclipse and move the focus to the model.

The second, most crucial one is the *sash window system*. In addition to ordinary Eclipse editors it provides a *multi-part editor* that is able to display multiple diagrams, each of them representing certain aspects of the model under development. This way Papyrus addresses multi-viewing of models. Both the model browser as well as the multi-part editor are shown in Figure 3.8. The latter one is currently selected, as denoted by its blue outline.

In addition to the diagrams of the UML2 standard, Papyrus supports the development of internal DSLs according to the UML profiles standard. Therefore, an Object Constraint Language (OCL) facility has been integrated. An exemplary extension

[10] http://www.eclipse.org/modeling/mdt/

[11] http://argouml.tigris.org/

[12] http://www.borland.com/us/products/together/

[13] http://www.magicdraw.com/

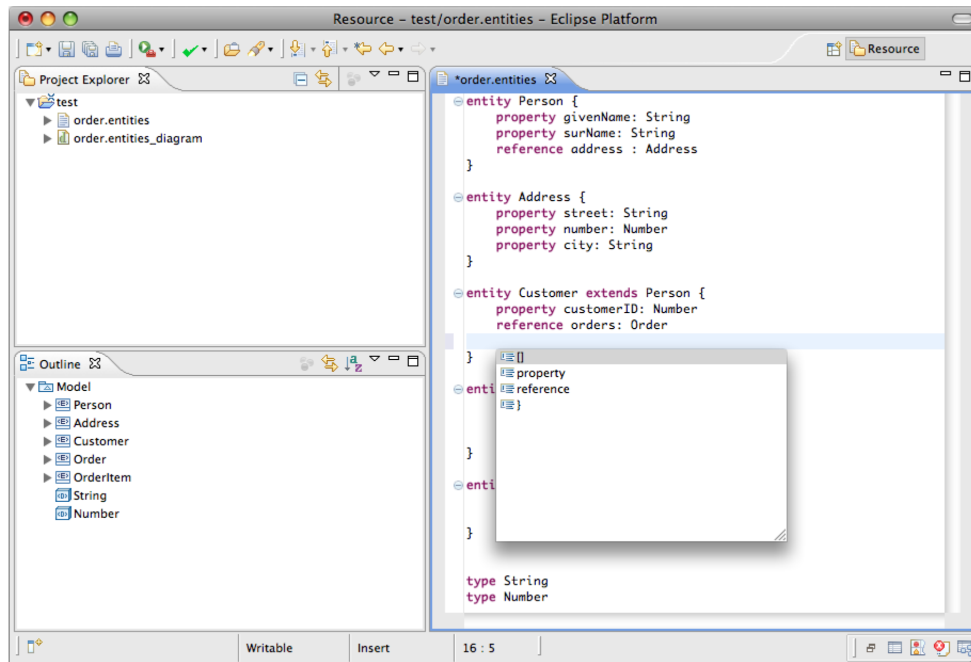


Figure 3.9: A Xtext-based editor tailored to a DSL.

contributed by the Papyrus developers is the supply of SysML.

Besides creating and maintaining models Papyrus supports code generation. Furthermore, it provides extension hooks allowing developers to engage custom tools that are working on the model for arbitrary purposes.

3.7 Xtext

Xtext is a modern Language Development Kit (LDK) for textual DSLs providing the creation of parsers, printers and formatters. The framework is built upon EMF as well and therefore it is compatible to the rich collection of EMF-based tools. The parser generation is delegated to proven frameworks like ANTLR. In addition tailored editors equipped with editing assistance features such as semantic folding, content assist, and validation are provided by Xtext. An exemplary editor is shown in Figure 3.9.

The Xtext framework comes with an own grammar language used to define the concrete syntax of the DSL. The abstract syntax may be derived from this grammar definition as pointed out in chapter 2. The same holds for the other way round: Xtext has been equipped with an import mechanism proposing an initial concrete syntax based on the Human-Usable Textual Notation (HUTN) for a given abstract syntax definition. This grammar proposal may be tailored towards the desired concrete syntax. Moreover, Xtext-based languages may be composed of several languages. A grammar may import various abstract syntaxes in form of metamodels. In addition, a

grammar may be based on another grammar. This allows the outsourcing of common parts of languages such as Boolean and arithmetic expressions.

The syntax definition works as with a conventional parser generator like ANTLR, SableCC[14], and Yacc[15]. Terminal symbols are described by means of regular expressions, the actual syntax is formulated in *productions*. The first production of a Xtext grammar denotes the start rule. While composing productions in an Extended Backus-Naur Form (EBNF)-like fashion the occurrence of terminals or references to other productions have to be named. These names refer to the attribute and association names in related metamodel.

Furthermore, cross references, i. e., non-containment associations, are supported by the Xtext grammar language. By their means the framework is able to handle DAG-shaped models rather than tree-shaped ones. Such cross references are treated in a so-called linking step that follows on the parsing step. The linking involves a class called `ScopeProvider` that must be implemented manually. It is responsible for revealing the model element that is referenced by means of an arbitrary terminal symbol, which is determined in the grammar.

In contrast to ANTLR and Yacc, semantic actions are not supported by Xtext. Further details on the grammar specification language can be found in the Xtext User Guide.[16]

The set of artifacts produced by Xtext's code generator involves a grammar compatible to the employed parser generation tool. By means of this grammar the language-specific parser classes are generated. More artifacts are a further parser dedicated to the syntax highlighting, a look-up class used by the serializer, and various stubs to be used for customizing purposes.

Xtext-based models are persisted by saving their concrete syntax representation to disk. This way secondary information such as line breaks, indentions and comments are preserved. The abstract syntax representation of such a model is created while opening it in the related editor. Consequently, it has to be updated each time a modification of the editor content takes place in order to avoid inconsistencies between the abstract and concrete representation. Concretely, each time the modeler adds, removes, or changes at least a character the related element of the domain model is discarded and recreated. This strategy is called *background parsing* (Scheidgen [43]).

The integration with EMF is achieved by means of a dedicated resource implementation. This paves the way for dealing with Xtext-based models programmatically or employing them, e. g., in a GMF-based editor. The printing and saving of such models is performed by a serialization facility. Custom formatting rules can be supplied, instructing the serializer to format the synthesized output as desired.

In order to preserve the secondary information of models that undergo only little modification, lots of effort has been spent by the Xtext developers. A *parse tree*

[14] http://sablecc.org/

[15] http://dinosaur.compilertools.net/yacc/index.html

[16] http://www.eclipse.org/Xtext/documentation/latest/xtext.html

model is built up by means of semantic actions added to the synthesized grammar. This parse tree, which is also an EMF-based model, preserves secondary information such as line breaks, etc. Subsequently, the Abstract Syntax Tree (AST) with respect to the related metamodel is derived.

Printing a model involves two steps: (1) constructing/updating the parse tree and (2) serializing the tokens according to the parse tree. The parts of the (domain) model that are unchanged since the model has been loaded are printed out by skipping the first step and just serializing the related parse tree elements.

Major requirements of the work presented here are the ability to (1) access the domain model maintained by the DSL-specific editor and to (2) update the content of the editor with respect to changes of the domain model. Accessing the model is basically as simple as with other editors, i. e., achieved by requesting the current active editor and its document. However, these accesses as well as accesses of the background parser must be coordinated in order to provide a consistent model. This is ensured by means of a proprietary lightweight transaction mechanism.

Operations examining the model must be encapsulated in an object satisfying the interface `IUnitOfWork`. Within such units of work the access to the resource is granted. They can be executed by the methods `readOnly()` and `modify()` provided by the `XtextDocument` class. Note that the latter method actually modifies the domain model, although the editor content is untouched.

This task is addressed by classes implementing the `IDocumentEditor` interface. They are able to process units of work and refresh the related editor accordingly. Note that the term “editor” is overload here: instances of `IDocumentEditor` have nothing to do with Eclipse editor parts. Here an “editor” rather denotes worker objects that can be instantiated and deployed in an arbitrary number in order to *edit* the model without user interference.

Chapter 4

Towards Multiform Model Editing

After the brief introduction of related work contributed by researchers and the community and the outline of constraints imposed by frameworks used within this work, approaches on maintaining and representing models shall be examined. This chapter starts with the investigation and assessment of techniques that are implemented in common tools, and proceeds with attempts of carrying these techniques into the world of Eclipse. Since these attempts exhibit certain shortcomings, the way of employing model editors has been fundamentally revised. Based on this revision a new approach of putting models into editors is proposed, which forms the major contribution of this work.

4.1 Repository-Based Model Management

A widely spread method of organizing the model management within modeling tools is characterized by the employment of a (local) model *repository*. In the current context the term repository denotes a local container of model elements that have to conform to a certain metamodel. The principle is studied using the example of the AToM3 tool, which has been presented at the end of chapter 2.

The way the model is maintained in AToM3 is basically an exemplary implementation of the Model-View-Controller (MVC) pattern. The model exists as a single source in the repository. The views, which are called viewports here, examine the model in order to display representations of the model, mostly chosen parts or aspects. They usually allow the execution of dedicated operations, which are also part of the model according to the strict definition of MVC [15]. Finally, a controller is in charge of coordinating the accesses to the model and notifying the viewports after modifications are performed on the model. The structure of AToM3's implementation of this methodology is depicted in Figure 4.1, which is taken from Guerra and de Lara [19].

A similar approach has been pursued by the developers of the Kiel Integrated Environment for Layout (KIEL) tool and can be found in lots of modeling tools such as the classical Computer-Aided Software Engineering (CASE) environments

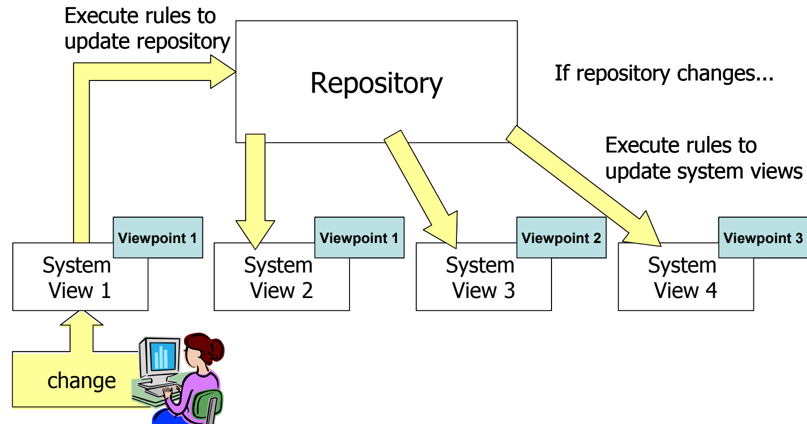


Figure 4.1: Repository-based model management in AToM3 (Guerra & de Lara [19]).

and their successors like Borland's Together[1] and MID's Innovator.[2] Advantages of this approach are

- the strict realization of the *single source* principle,
- the avoidance of inconsistencies between several views,
- clear interfaces between repository and views, and
- the extensibility of the tool with respect to further views.

A major drawback is the need for a dedicated model repository. In open platforms like Eclipse such a repository is not available. Furthermore, graphical and textual editors provided by frameworks such as GMF and Xtext work on plain documents. Therefore, it is most likely that a modeler is not working on just one but on multiple documents concurrently. Thus, a lot of effort ought to be spent in re-engineering these editors, in order to integrate them with such a repository.

4.2 Lightweight View Construction Methods

In platforms like Eclipse almost any kind of data is stored in the form of files. As this holds for model data, too, these are persisted in shape of their XMI or plain text representation. Defining DSLs and creating dedicated editors usually implies the introduction of new file types denoted by their file extension.

It is further common that domain-specific models are created and edited within the same editor. Apart from tree-based outlines, alternative representations are rarely available. In order to come to alternative views of models various ways are possible.

[1] http://www.borland.com/de/products/together/

[2] http://www.mid.de/en/products/innovator-modeling-platform.html

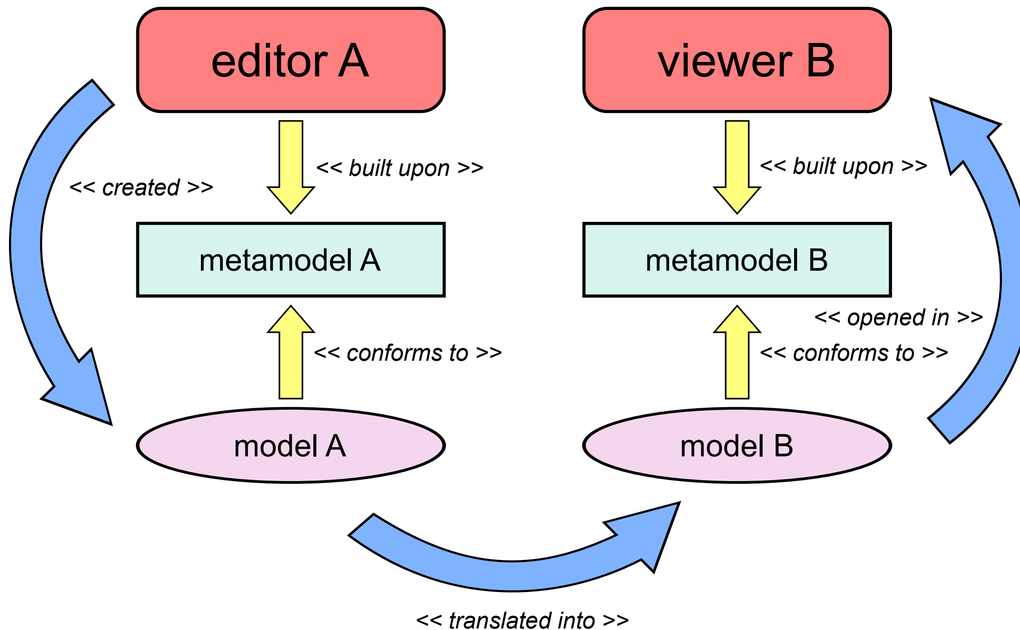


Figure 4.2: Obtaining model representations by means of model transformations.

The first one to be outlined leverages existing model viewers such as graphical, structural, or textual editors. Those tools are typically based on a metamodel that differs significantly from the metamodel of the models to be visualized in a certain way. Therefore, the original models must be translated into the syntax of the viewer tool. The resulting models can be opened in the tool, afterwards. A scenario involving this approach is sketched in Figure 4.2. Note that viewer B may be an editor as well.

This approach has certain advantages:

- avoidance of creating further custom tools,
- applicability in combination with closed source tools,
- very little “time to result”, and
- obtainment of suggestions/inspirations for a potential custom syntax.

Limitations/drawbacks of this approach are:

- The look of concrete representatives (figures) of model elements, e. g., stroke width or font alignment, is hardly influenceable or even not at all.
- Preserving the correlation of model elements and representatives is generally impossible, as the viewer’s metamodel is not able to keep references to the original elements. This implies

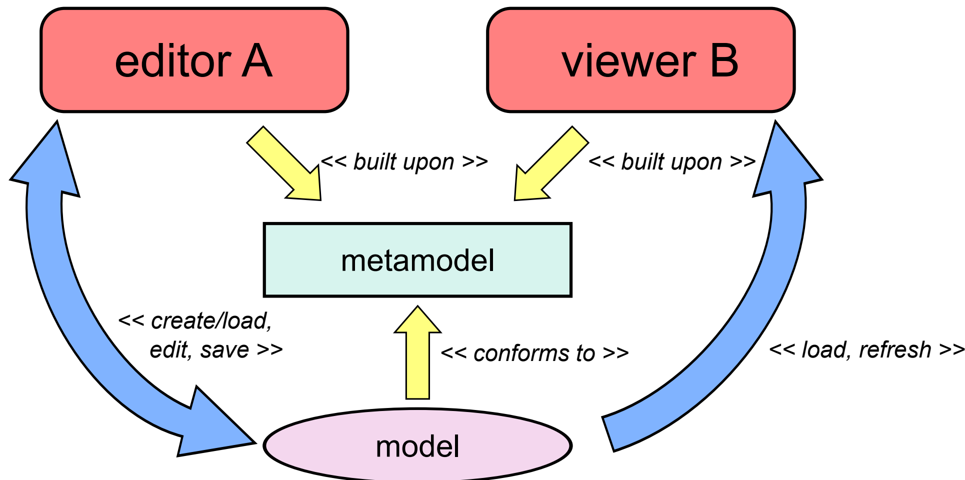


Figure 4.3: Obtaining model representations by means of dedicated model viewers.

- the difficulty of providing incremental updates of the views and
- the impossibility of feeding modifications made in views back to the original model.

Finally, lots of (temporary) data are generated on the disc.

The second method to be explained is applicable if views of existing tools do not satisfy the requirements on proper representations. In such a situation it is mostly unavoidable to construct a further viewing or editing tool in order to obtain dedicated model representations. As such tools used to be built upon a certain metamodel, it is natural to choose the metamodel the given models conform to. The resulting situation looks as follows: two or more tools for editing/displaying models exist as depicted in Figure 4.3. Each of them provides a dedicated concrete syntax for visualizing models or chosen parts of them.

Besides the tailored representations this setting benefits from the fact that no more transformation step is needed. Instead, the model can be opened by multiple editors/viewers, which may be refreshed each time the model has been saved. Furthermore, such tools might be installed side by side in order to provide developers different views on the models they are working on. This can help them to validate their modifications resulting in less mistakes.

The next step of improvement is inspired by the fact that different parts and aspects of the model are expressed by means of different representations. A typical example is the characterization of classes in the UML. Static information such as name, attributes, methods, and relations to other classes are expressed in Class Diagrams, whereas dynamic properties like interaction with other classes or internal sequential behavior may be specified by means of Communication Diagrams or UML State Machines.

The most straightforward way to realize the characterization of model elements

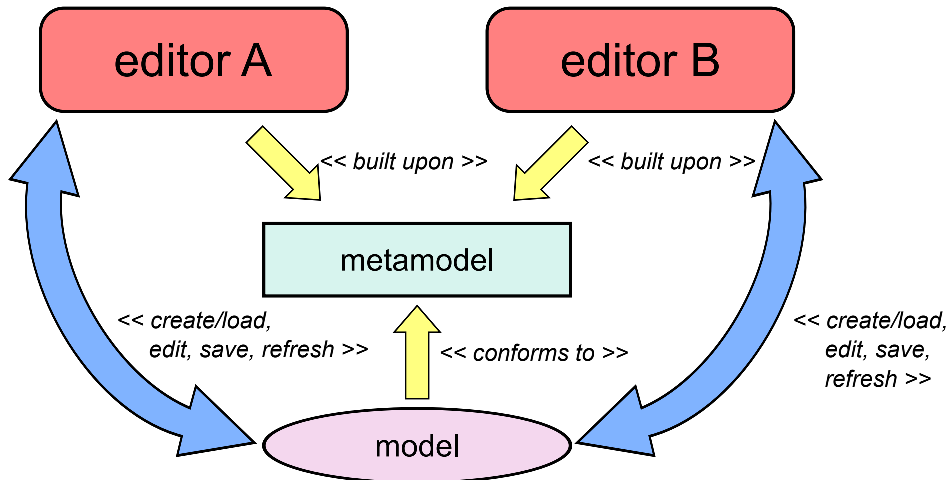


Figure 4.4: File-based synchronization of multiple model editors.

by means of different syntaxes is the distribution of the particular definitions over several files. A consequence of this solution is the emergence of a potentially large number of files that have to be maintained and kept consistent. Alternatively, a single file can be used in order to preserve the whole information of the model. This results in a setting with multiple editors that are working on the same file, whereby each one facilitates the specification of chosen aspects of the model. Such a scenario is given in Figure 4.4. Note that the "scopes" of the editors need not be disjoint, meaning that convenient functionalities such as renaming a class in a Communication Diagram might indeed be possible.

This approach has been chosen by Bayramoğlu [4] for the purpose of enabling the browsing and editing of graphical and textual representations of the same model concurrently in the context of Eclipse. A further, basically similar, attempt has been made by the developers of Xtext themselves. Both experiments suffer from issues that occur while modifying a file with multiple tools simultaneously.

In order to explain them in some detail consider the following situation: a model is created in shape of a textual representation. Afterwards, a graphical (GMF-based) representation is derived and opened in the related editor, too. Each time the model is modified and saved in one of the editors, say editor A, the other one, B, is caused to reload the model. This, however, works only if the content of B is not changed with respect to the content of the model file on the disc. If B is *dirty*, i. e., modifications are made without saving the model to the disc, conflicts may come up and the refreshing process fails.

Further problems are due to GMF's handling of domain and secondary information. Recall that GMF-based editors rely on *notation models* in order to persist graphical representations. This involves the set of elements to be displayed, sizes, positions and references to the elements in the *domain model* to be represented. The notation model is the primary source of information of such an editor.

Adding further elements to the domain model according to the scenario above will not necessarily cause the graphical editor to represent them in the diagram, although it is refreshed. Removing elements from the domain model will be even worse. As the notation model still contains representatives and broken references, deformed and incomplete figures remain after the refresh of the editor. Though this can be alleviated by modifying the editor such that it refreshes the notation model by means of the canonical edit policies, further issues remain. These are due to EMF's addressing mechanisms that are used in order to denote the domain elements.

By default elements of models are addressed by means of so-called *fragment URIs*. Fragment URIs contain a relative path to the resource (the file) the addressed element is contained in. This path is followed by an absolute path to the addressed element along the containment references of the related metamodel. The exemplary fragment URI *model.kixs#//@states.*1*/@signals.*2 denotes the third element of the list “signals” contained by the second element of the list “states” of the root element contained in the resource “model.kixs” in the current folder.

While addressing model elements this way is robust with respect to changes of names, identifier, etc. confusion will occur if the order of elements in a list is changed. This is generally not acceptable as, e. g., the order of variable declarations in a program or class definitions in a class diagram does not play any role.

Alternatively, *key attributes* may be determined in order to address model elements. Such key attributes must satisfy the *id* property, which means that valid attribute values must be conform to a certain regular expression. Furthermore, any value may occur at most once in a model. In contrast to the former address method, this one is robust with respect to movements of elements. However, renaming them is discouraged as that leads to model inconsistencies and confusion as well.

In addition to those issues on the graphical side there are further problems with textual representations. A consequence of the change from sources in form of plain code to sources in form of models is the need for documentation within models. A rather simple method of treating such information is integrating it into the abstract syntax, i. e., considering it while designing the metamodel. If this is not possible, comment lines, which are well-known from common programming languages, can be added to textually formulated and persisted models. Such information, however, is not available in other representations of the model. Even worse, a proper maintenance of such comments is basically impossible to achieve if the related model elements are moved or removed.

A further, conceptional drawback of the *on-save-synchronization* is the fact that the document has to be persisted in order to achieve a refresh of the alternative views. This generally contradicts the desires of developers, who are used to the freedom of making changes and rejecting them if necessary. These considerations lead to the conclusion that this way of editing models is not a worthwhile option.

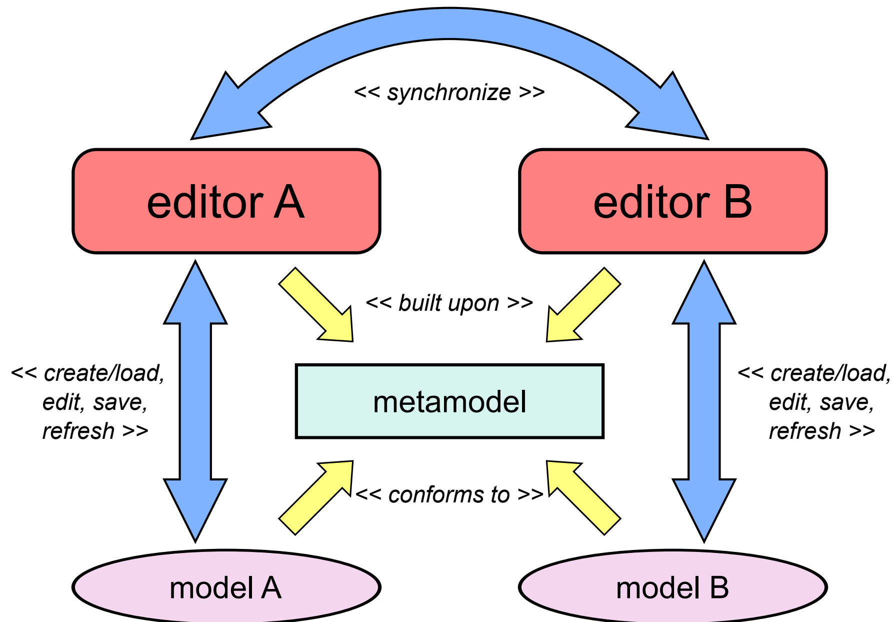


Figure 4.5: *Difference-and-merge*-based synchronization of multiple model editors.

4.3 Model Representations based on Transient Views

Considering the various difficulties and limitations of the techniques introduced in section 4.2, the *synchronize-on-save* approach has been rejected. The key idea that is pursued in this work can be put in a nutshell by means of the following statement:

> *Let the editors maintain their runtime model and do not influence the loading and saving processes. Instead employ an appropriate difference-and-merge mechanism in order to update complement views on the model.*

The first setting that follows this principle looks as follows: various editors and viewers may be opened for displaying models. Although these tools may share the same model file, it is suggested to let them work on different files in order to avoid the formerly outlined problems. Model modifications are broadcast from the active editor to the remaining ones on demand, or may be even triggered by the modification itself. This way synchronization and refreshment of the model representations is completely decoupled from the persistence issue. Furthermore, model accesses caused by broadcasts of changes are coordinated by the tools' dedicated transaction mechanisms. They ensure model consistency and provide undo/redo support. The structure of this design is outlined in Figure 4.5.

A disadvantage of this setting in the context of Eclipse is the need for dedicated files used by the editors to work on. Recall from the previous chapter that Eclipse editors cannot be opened without a valid input file. Therefore, an appropriate refinement of this setting has been developed.

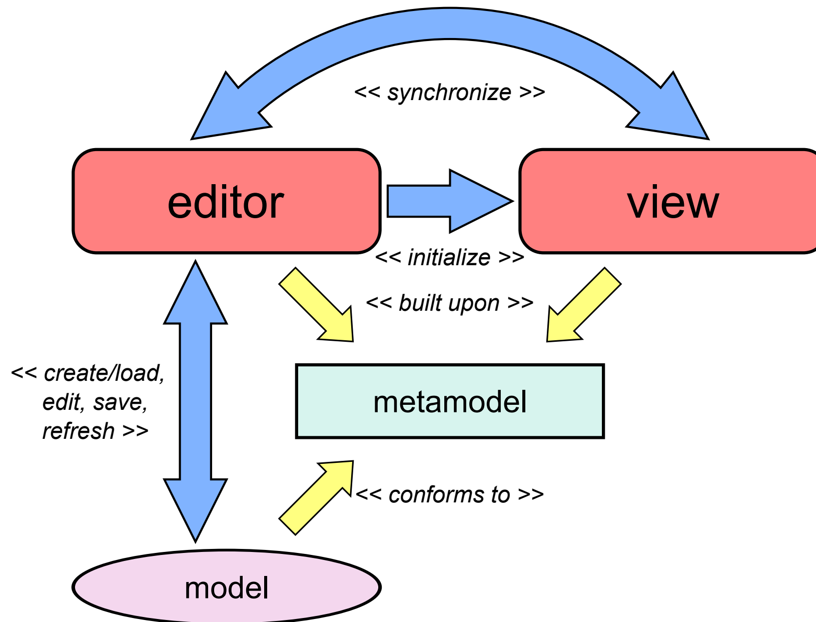


Figure 4.6: *Single source/multiple view*-based model representations.

The improvement is characterized by the restriction to exactly one fully equipped editor that maintains the model to be treated. Further representations of the model are desired to be provided by Eclipse views. These views are initialized with the content of the currently active editor and are refreshed each time a model is opened, the modeler switches to another (opened) one, or model modifications are to be broadcast. In contrast to displaying the entire model content (with respect to the model parts the desired representation is dedicated to), the (re-)initialization of the view may be selective. This means that the modeler is able to determine segments of the model to be displayed. Such an interactive view construction would allow the creation of customized class diagrams, for example, without dragging and dropping the chosen classes into a dedicated diagram.

This approach may even be extended to a powerful *semantic zooming and panning* mechanism. It can be realized by operations that involve adjacent model elements on demand or exclude particular ones from the representation resulting in an experience that is similar to browsing a digital map. This use case and further ones have already been identified by Fuhrmann and von Hanxleden [13]. In order to coordinate and decouple the concepts from concrete editor or viewer implementations, they propose the establishment of a dedicated *View Management*.

Besides exhibiting information, view parts may be employed for the purpose of providing a user interface that allows the modification of arbitrary content. This can be exploited in order to enable model editing beyond the capabilities of the model editor. This scenario has been realized in KIELER. The structure of the solution is depicted in Figure 4.6, its context and implementation are discussed in chapter 5.

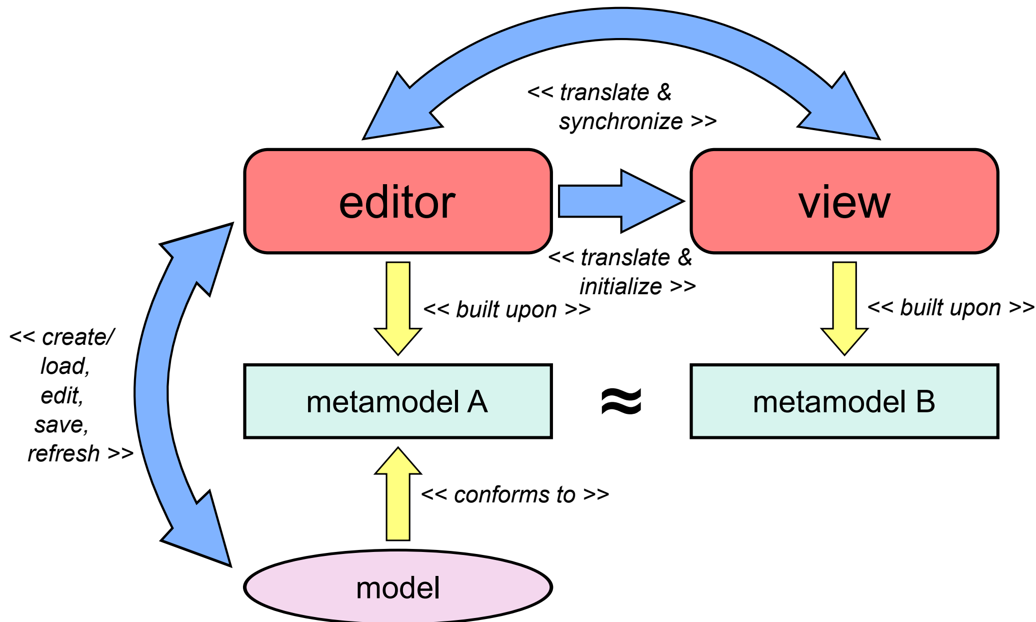


Figure 4.7: Cross-metamodel-based supply of model representations.

An interesting property of this approach is the convergence to the repository-based model management. On a closer look one observes that the infrastructure, which enables the initialization and synchronization mechanisms, emulates a model repository by means of the model editor.

The last technique for obtaining model representations based on transient views to be discussed here abolishes the restriction to a unique metamodel. This is motivated by the fact that various frameworks for generating model editors, e. g., GMF and Xtext, set certain conditions on the metamodel. In order to achieve a certain representation of model content, it is often unavoidable to adapt the meta model accordingly. In such cases it might be helpful to build tailored editors/viewers upon a (slightly) modified metamodel. This advantage admittedly requires with further effort.

The initialization and synchronization of these views involve a model translation step that induces some of the problems mentioned at the beginning of section 4.2. These issues, however, do not restrain the applicability as hard as in the first proposal, since the degree of abstraction and the amount of information will not change significantly while the model is translated. This assumption is based on the prerequisite that the metamodels differ just partially. The accommodation of auxiliary information is much easier, too, as the metamodels are not given by any tool and can be adapted accordingly. The resulting structure of tools and their interaction is shown in Figure 4.7. Since an implementation of the latter method is not in the scope of this work, experimental results and experiences are not available yet. However, it seems worth investigating this scenario in the future.

Chapter 5

Multiform Modeling in Kieler

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) targets the development of modeling techniques beyond the common *drag and drop* paradigm. In particular, it aims at improving the *pragmatics* of the modeling languages. This encompasses the applicability of a modeling language as well as related modeling techniques, as stated by Fuhrmann and von Hanxleden [13, 14]. Furthermore, KIELER is designed to be a testbed for investigating the execution semantics and runtime behavior of modeling languages. It provides the following core concepts:

Automatic Layout: The KIELER Infrastructure for Meta Layout (KIML) contributes methods that enable automatic layout of graphical model representations. In addition to implementations of diverse layout algorithms, further mechanisms are contained. These mechanisms allow the pre-definition of preferred layout settings for particular diagram elements as well as a user interface for exploring the optimal layout manually.

Structure-Based Editing: The KIELER Structure-Based Editing Framework (KSBasE) framework provides a valuable alternative to the common drag & drop based modeling techniques. It enables the *structure/syntax directed editing* of arbitrary models. This method is similar to the editing paradigm pursued in MPS (see chapter 2). It has been carried into the graphical modeling world of Eclipse, although its applicability may easily be extended to textual modeling languages.

View Management: In order to integrate the formerly mentioned techniques and further ones, the *View Management* concept proposed by Fuhrmann and von Hanxleden [13] has been implemented in the KIELER View Management (KiVi) project. Its aim is to provide simple means for modifying representations of models on demand. Such modifications are called *effects* and cause model representations to change in a certain way. The execution of such effects is initiated by events called *triggers*. Triggers and effects are connected by *combinations*, which are decoupled from concrete editors and designed to be implementable as comfortable as possible.

Simulation, Code Generation, Data Visualization: In addition to creation and representation of models, this section of KIELER covers their execution. The

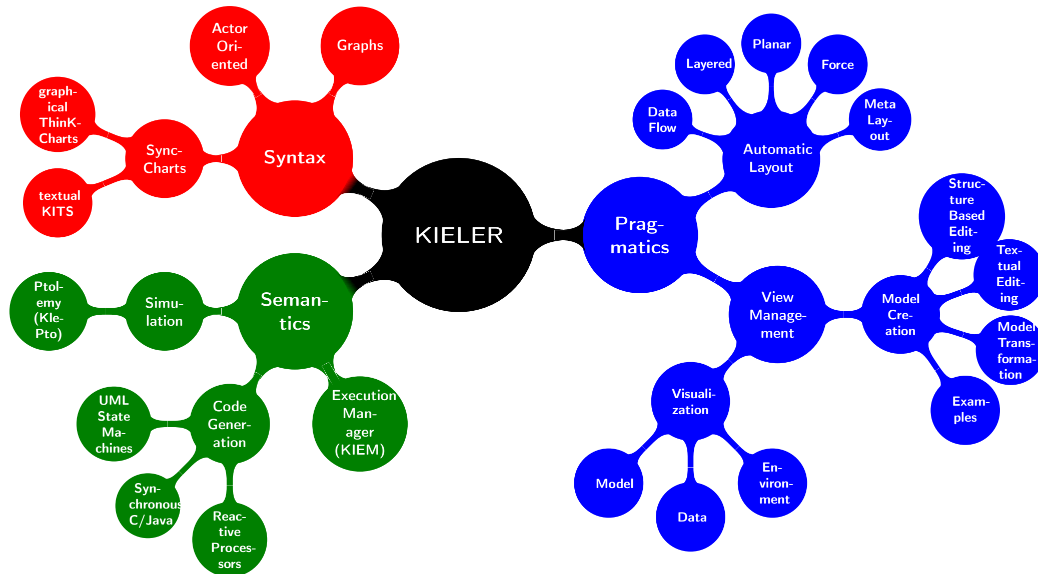


Figure 5.1: Components of KIELER.

> KIELER Execution Infrastructure (KEI) enables the simulation of models, their execution on soft processors, and the generation of implementation code within Eclipse. Besides integrating this collection of execution platforms, it supports feedback of data and their visualization in the model and/or dedicated schematic environment representations.

An overview of selected components of KIELER is given in Figure 5.1.

Within this environment a model editor has been equipped with an additional view according the method that is contributed by this work and explained at the beginning of section 4.3. This chapter proceeds with the presentation of the graphical editor and lists particular shortcomings that motivate the need for another model representation. In the following, an alternative textual syntax is presented. Afterwards, details on the integration of the textual SyncCharts representations are discussed. The chapter concludes with a description of the implementation and some limitations.

5.1 The Thin KIELER SyncCharts Editor (ThinKCharts)

The Thin KIELER SyncCharts Editor (ThinKCharts) is a GMF-based editor, which has been created by Schmeling [45]. It supports the formulation of behavior specifications in the *SyncCharts* language. This language is a dialect of *StateCharts*, which have been proposed by Harel [20] in 1987. StateCharts in turn is an advancement of Mealy Machines [32], supplemented by hierarchy, concurrency, event broadcast and data. SyncCharts was introduced by Charles André [1] in 1995. Compared to StateCharts it is restricted and extended in certain points. The most important characteristic is

its subjection to the *Hypothesis of Synchrony* [5] implying a discrete time behavior and strict determinism.

Elements of SyncCharts

In the following the elements of the SyncCharts language are presented in some detail. Since this work is about modeling techniques here, the focus is on syntactic aspects, whereas semantic issues are mostly omitted. Therefore, the constructs are explained by means of their implementation in ThinKCharts. For details on their semantics refer to André [1, 2].

Consider the example in Figure 5.2, called ABRO. It is the most common example for discussing synchronous languages since it illustrates their essential concepts.

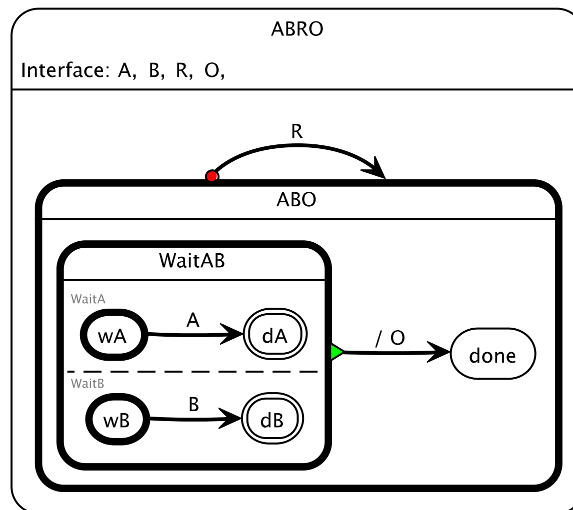
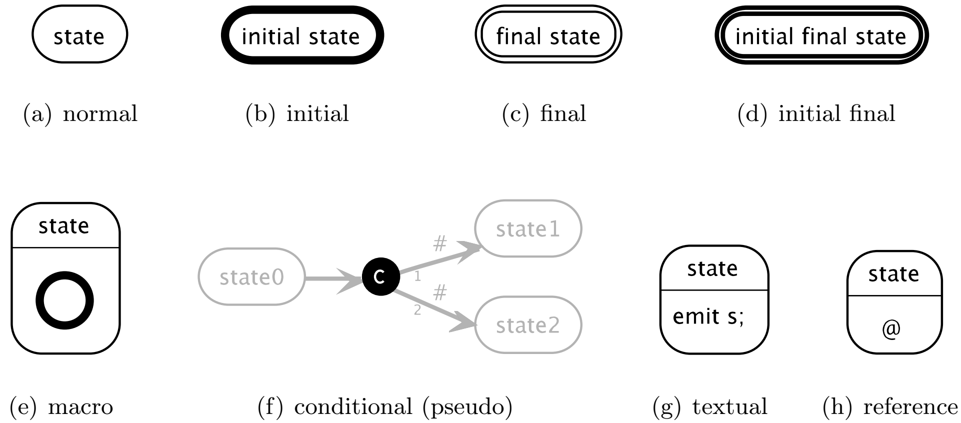


Figure 5.2: The "hello world!" example of *SyncCharts*, called ABRO.

States such as ABO, dB, and done are represented by circles or rounded rectangles. They are usually labeled and can be marked to be initial, final, or initial final states. In addition, they may accommodate *regions*. Regions in turn are supposed to contain states, at least an initial state. This way hierarchy is introduced in SyncCharts. Regions may be labeled, too, but do not have fixed borders. Adjacent regions are separated by dashed lines, as observable between the regions WaitA and WaitB. This way, i. e., by attaching multiple regions to a state, concurrency is expressed in SyncCharts.

Transitions are depicted by arrows leading from a source state to a target state. They can be labeled with guard and effect expressions and a priority setting. Furthermore, transitions may be decorated with red circles or green triangles denoting different kinds of preemption of the source state, if such refinement content exists. An overview on the possible occurrences and configurations of states, regions, and transitions is given in the following.

- *States* are typically circles or rounded rectangles of solid lines, which may be labeled, see Figures 5.3(a) - (d). In addition, states may occur in other settings, as shown in Figures 5.3(e) - (h).

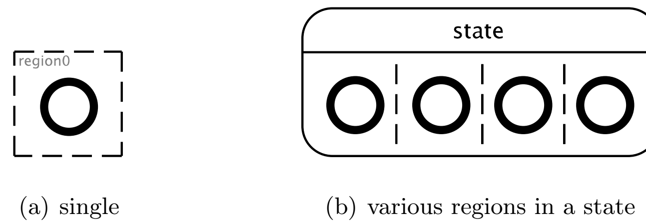


(a) normal (b) initial (c) final (d) initial final

(e) macro (f) conditional (pseudo) (g) textual (h) reference

Figure 5.3: Types of states in SyncCharts.

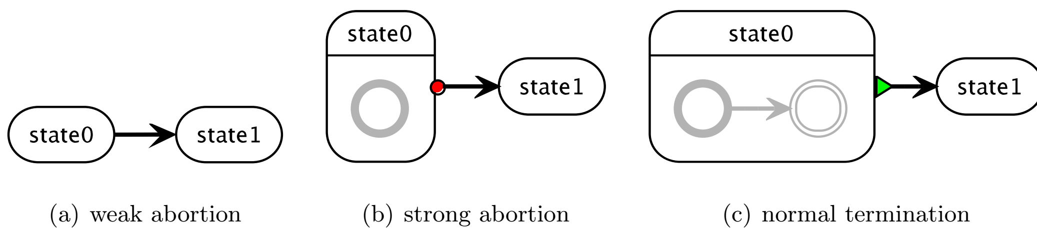
- *Regions* accommodate states, and are separated by dashed lines that drop near surrounding state lines, see Figure 5.4.



(a) single (b) various regions in a state

Figure 5.4: Regions with initial states.

- *Transitions* are arrows connecting states, and may be labeled, see Figure 5.5.



(a) weak abortion (b) strong abortion (c) normal termination

Figure 5.5: Types of transitions.

States and regions may be equipped with *signals* and *variables*. Both have a name, a type, and an initial value. Whereas variables may be defined to be constant, signals may be local ones, inputs, outputs, or both inputs and outputs. In ThinKCharts, however, only the name of signals and variables is visible in the diagram. Figure 5.6 shows signals and variables declared in a state.

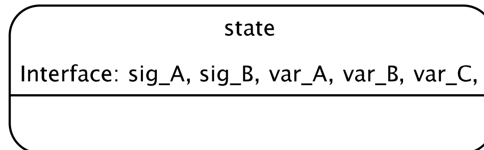


Figure 5.6: Signals and variables attached to a state.

Signals and variables can be accessed in actions related to states and transitions. Signals can be emitted and tested. Emitted signals are called *present* signals, the remaining ones are called *absent* signals. In addition, signals may be declared to be *valued* in order to record values. Variables may only record values. The Boolean value of the present state of signals as well as the value of valued signals and variables can be composed to complex Boolean or valued expressions, as known from common imperative programming languages. Such expressions might serve as guards of actions or right hand side values of assignments or value emissions. Figure 5.7 shows an example SyncChart model exploiting signals and variables. In the example in Figure 5.7 `BUF` is a valued signal. The remaining identifiers denote variables.

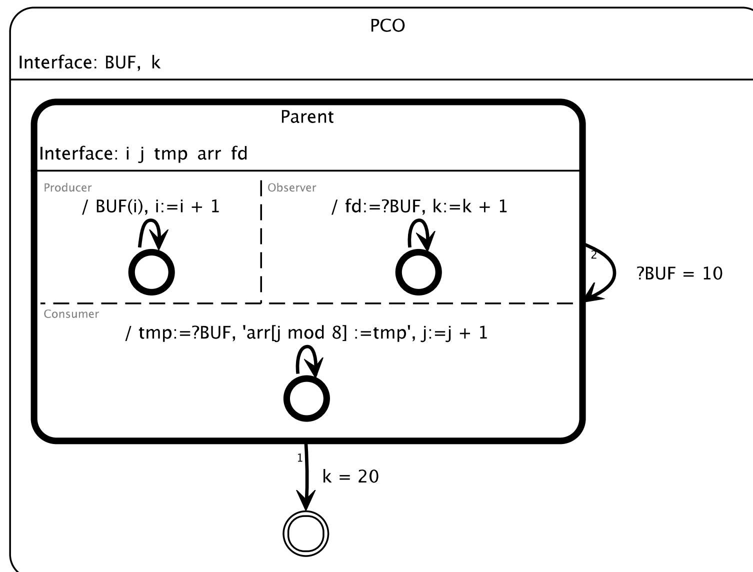


Figure 5.7: Usage of signals and variables in ThinKCharts.

The remaining part of this section covers the actions that may be attached to states. Consider Figure 5.8.

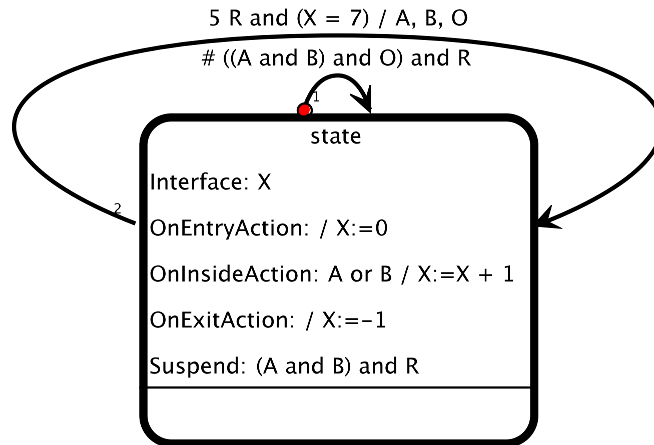


Figure 5.8: Usage of state actions in ThinKCharts.

Besides transitions SyncCharts allows states to be equipped with actions dedicated to different situations: actions to be executed during the entry into a state, while pausing in a state and on leaving a state. At a closer look one will observe that all of them might be formulated by means of transition actions. Hence, they may be noted as a form of syntactic sugar. An exception is the suspension trigger, a further concept SyncCharts adopted from the synchronous world. If the flow of control pauses in the related state and the Boolean guard evaluates to true the execution of the state's content is frozen and does not affect the system in any way. In contrast to state actions, the suspension may not have any side effects beyond the freezing.

Consequently, all of them, including transition actions, are specified in the same manner. They are formulated textually and have to conform to the same grammar. The start rule of this grammar is shown is Listing 5.1.

```
1  Action:
2    '#'?  INT?  BooleanExpression?  ( '/'  Effect  ( ','  Effect )* )?
```

Listing 5.1: Grammar excerpt of action declarations in ThinKCharts.

The optional leading hash mark determines the *immediate* property of a transition. It may be followed by a delay count. Afterwards, a Boolean expression serving as the guard of an action or suspension trigger may be determined. Finally, effects of the action under consideration can be added after a slash. Multiple effects are comma-separated.

Issues of ThinKCharts

After this inventory of elements of SyncCharts and their representation in ThinKCharts, some shortcomings that confirm the need of another representation are discussed.

As already mentioned, configurations of declared signals are not visible in the graphical representation provided by ThinKCharts. Besides states, regions may be equipped with signals and variables as well. The editor, however, is not able to deal with this yet. Furthermore, states and regions, which are often referred to as *scopes*, may reference other arbitrary models or contain arbitrary content that need not conform to SyncCharts.

Last but not least, it is not possible to annotate representations of elements of a model such that the annotation is part of the abstract syntax, although the metamodel permits the incorporation of such information. Currently, ThinKCharts does not support access to these annotation means. By default, the only way of attaching such kind of secondary information is the use of graphical notes that are available in any GMF-based editor. This is illustrated in Figure 5.9. Such notes, however, are just part of the diagram.

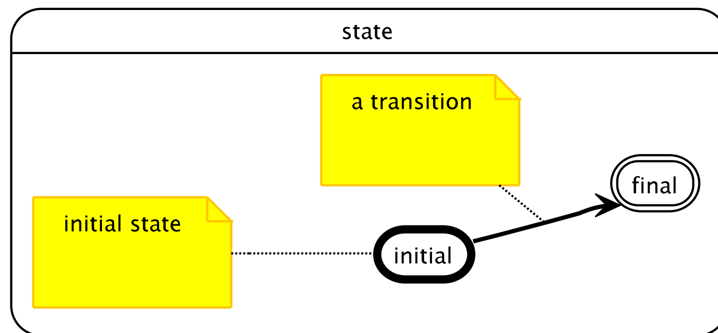


Figure 5.9: Graphical notes provided by GMF.

The metamodel

Considering the subsequent proposal of a textual representation of SyncCharts models that is based on the same abstract syntax, this introduction of ThinKCharts concludes with a short outline of the meta model the editor is built upon. Figure 5.10 shows the Ecore model reflecting the elements and relations of the SyncCharts formalism and some additions.

For reasons of reusability, parts of the SyncCharts meta model have been moved to dedicated meta models. This applies to classes representing atomic as well as compound Boolean and valued expressions. The resulting expressions meta model is depicted in Figure 5.11. Other candidates of outsourcing are classes providing the annotatability of the kernel elements of SyncCharts models. They have been moved to a dedicated annotations meta model, whose elements are intended to be imported and reused by further concrete application specific meta models. To improve the

lucidity of the relations between these meta models, some of the referenced classes are depicted in the referencing meta models, indicated by the small boxed arrows in the upper right corner.

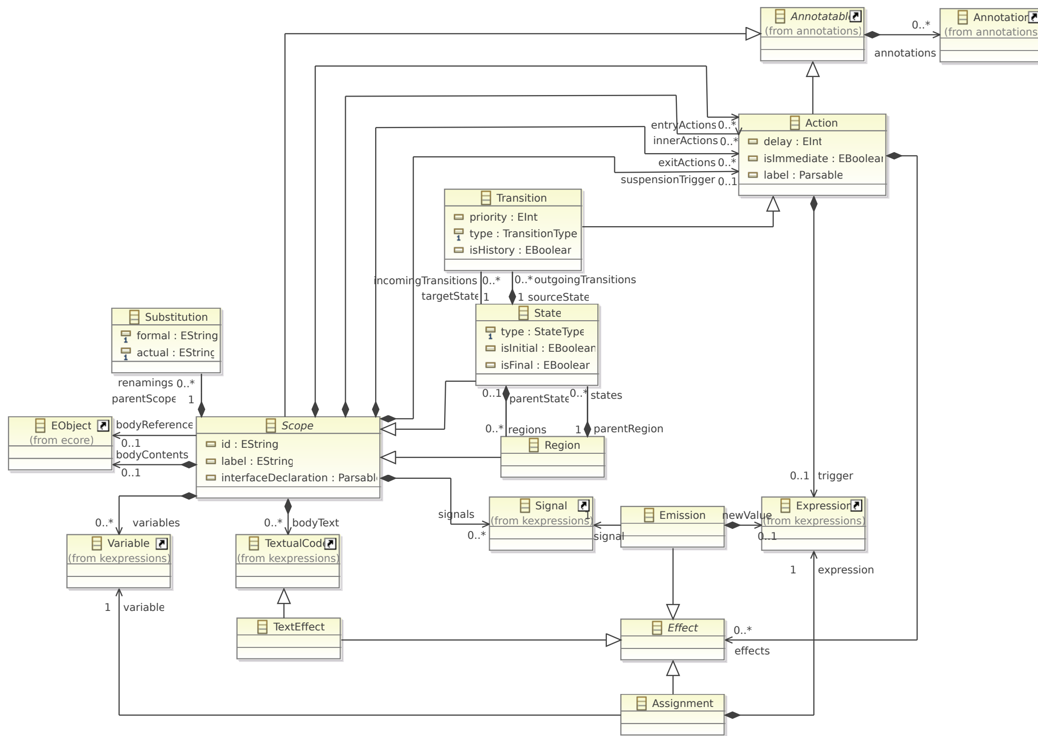


Figure 5.10: The SyncCharts meta model.

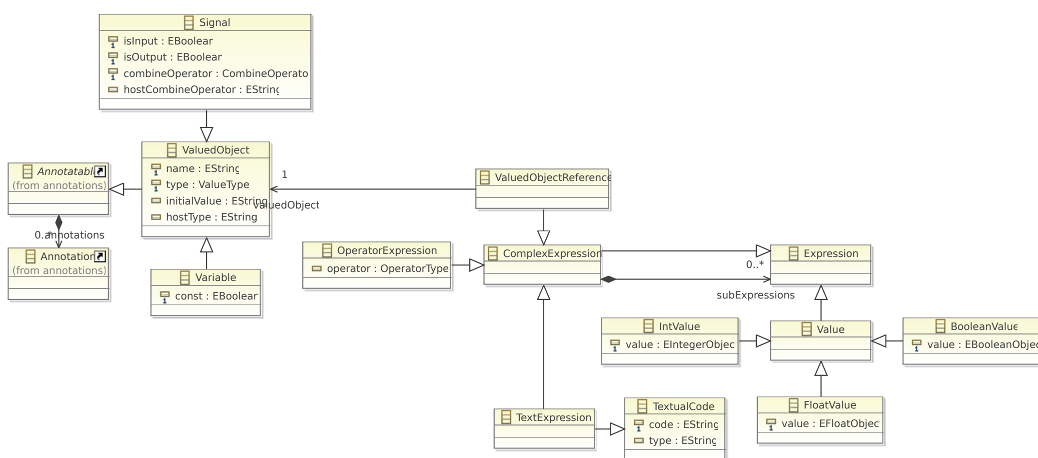


Figure 5.11: The expressions meta model.

5.2 A textual SyncCharts language

Besides the issues of ThinKCharts revealed in the previous section, the graphical editing of models may be rather tedious, as motivated earlier in this work. To overcome these drawbacks, a textual syntax has been designed such that every construct defined in SyncCharts can be formulated in this language. It has been named KIELER Textual SyncCharts (KITS) and is built upon the SyncCharts metamodel. KITS has been initially developed by Bayramoğlu [4] and is further elaborated within this work.

Before going into the details of KITS's definition, the SyncCharts model we met already in section 5.1 (page 51, Figure 5.2) is presented in shape of its KITS representation in Listing 5.2:

```
1  state ABRO {
2    input signal A
3    input signal B
4    input signal R
5    output signal O
6    init state ABO {
7      init state WaitAB {
8        region WaitA:
9        init state wA
10       --> dA with A
11       final state dA
12
13       region WaitB:
14       init state wB
15       --> dB with B
16       final state dB
17     }
18     >-> done with / O
19     state done
20   }
21   o-> ABO with R
22 }
```

Listing 5.2: ABRO in the textual KITS language.

At first glance one will notice the keywords `state`, `signal`, `region`, etc. They are in charge of making the modelers able to associate the related graphical elements immediately in their mind. The leading keywords of transition definitions, `-->`, `o->` and `>->`, are chosen to serve the same purpose. Syntactical separators such as parentheses are reduced to a minimum. Solely the refinement contents of states have to be enclosed in curly braces.

The `state` statement

After the short outline of the overall design the definitions of the particular statements of KITS are discussed. The first one is the `state` construct since it is usually the root element in KITS representations. Its grammar is depicted in Listing 5.3.

```
State:
    (StringAnnotation)*

    ( ('init' 'final'?) | ('final' 'init'?) )? StateType? 'state' ID STRING?

    ( ('@' [State|ID] ('[' Substitution (',' Substitution)* ']')? )
     |
      ('{'
          (
            ( Signal              |
              Variable            |
              'onentry'    Action |
              'oninner'    Action |
              'onexit'     Action |
              'suspension' Action
             )*
            ( TextualCode )*
            ( SingleRegion (Region)* )?
           )
          '}')
     )?
    (Transition)*;

Substitution:
    ID '/' ID;

enum StateType:
    'normal' | 'conditional' | 'reference' | 'textual';
```

Listing 5.3: Grammar of KITS's `state` statement.

First, states can be annotated with string annotations. Annotations are discussed later on, so no attention will be paid on them here. The actual state declaration starts with optional `init` and `final` modifiers that can occur solely, or in combination. States configured with these modifiers are depicted in Figure 5.3(a) - (d), page 52. Subsequently, another modifier may occur configuring the type of state in terms of being a normal or conditional pseudo state, or the kind of specification of its content. Possible values of `StateType` are `normal`, `conditional`, `textual` and `reference`. The graphical representations are depicted in Figure 5.3(e) - (h). If this type is not mentioned the state is assumed to be a normal state. Depending on whether it contains regions, the state is either drawn as a simple state or a macro state, as shown in the Figures 5.3(a) and 5.3(e), respectively.

After those modifiers the mandatory keyword `state` must occur. It has to be followed by an identifier that is mandatory, too, and is used to address the state in transitions. As the name 'identifier' already suggests, this attribute must be unique amongst all states contained in the same region. This is sufficient since SyncCharts does not allow inter-level transitions and transitions from one region into a sibling region do not make sense with respect to the execution semantics. The identifier must satisfy the terminal definition rule `ID`, implying that it may be composed of letters and digits starting with a letter. Besides, states may be labeled with an arbitrary string enclosed in double quotes (`"..."`) according to the `STRING` terminal

```
1  Region:
2      (StringAnnotation)*
3      'region' ID? STRING? ':'
4      (Signal | Variable)*
5      (TextualCode)*
6      (State)*;
7
8  SingleRegion:
9      (
10       (StringAnnotation)*
11       'region' ID? STRING? ':'
12       (Signal | Variable)*
13       (TextualCode)*
14      )?
15     (State)*;
```

Listing 5.4: Grammar of KITS's `region` statement.

rule.

Next, a referenced SyncCharts model proceeded by a leading `@` and signal substitutions may be addressed if the current state is set up to be a reference state. Alternatively, nested elements can be added to the state enclosed in curly braces. Possible attachments are interface elements such as signals, variables, actions and suspension triggers as well as textual code blocks and nested regions. The `TextualCode` fragment is a means for integrating other languages in KITS.

The interface parts can be defined without any ordering. In contrast, textual code and regions must not be mixed up with such elements, but follow them strictly.

Finally, the definition of state transitions may be attached to the state. They are to be understood as transitions that leave the state they are attached to and lead to another (target) state. Note that this allows self-transitions, which leave and re-enter the same state.

The `region` statement

The second statement to be reviewed is the `region` statement. As seen in the grammar fragment of the `state` statement, regions may be contained by states. On the other hand regions usually contain further states. The grammar fragment of defining the region definition is presented in Listing 5.4, lines 1 - 6.

Similar to states, region definitions start with optional annotations. They are followed by the mandatory `region` keyword, optional identifier and label attributes, and a mandatory colon. As mentioned in the preceding section, signals and variables may be declared in the scope of a region. This takes place after the region's naming. Furthermore, regions may contain textual code, too. Finally, the states located in the region are defined.

Note that only regions may contain states, not states themselves. However, recall ABRO in Listing 5.2. Observe that WaitAB seems to be directly contained in ABO that in turn is directly contained in ABRO. To allow such a compactification, which is

```
1  Transition:
2      (StringAnnotation)*
3      TransitionType (INT)? [State|ID]
4      ( 'with'
5       (
6         ( '#'? INT? BooleanExpression? ('/' Effect (',' Effect)*)? )
7        |
8         ( STRING )
9        )
10      )?
11     'history'?;
12
13 enum TransitionType:
14     '-->' | 'o->' | '>->';
```

Listing 5.5: Grammar of the transition definition in KITS.

reasonable of course, a further, specialized grammar rule has been introduced, shown in Listing 5.4, lines 8 - 15. According to this rule, a region need not necessarily be named if it shall not be annotated and does not contain any declarations of signals, variables, or textual code. Observe in Listing 5.3, line 18 that the first region contained in a state must satisfy this `SingleRegion` rule whereas the former `Region` rule has to hold for further regions. This implies that line 8 in Listing 5.2 may be omitted.

Transitions

Next, the concrete syntax of transition definitions will be discussed by means of Listing 5.5. As well as scopes, i. e., states and regions, transitions may be annotated with string annotations.

A transition specification starts with one of the keywords `->`, `o->`, or `>->` determining the transition's semantics in terms of its kind of preemption according to the variants introduced in Figure 5.5, page 52. The leading transition type keyword is followed by an optional priority setting and the mandatory target state identifier . As emphasized while explaining the `state` statement, the identifier is an obligatory attribute of states that must be unique in the context of the enclosing region.

The target state is followed by optional settings concerning action characteristics of transitions. They are introduced by the keyword `with` and formulated in the same way such as in ThinKCharts, given in Listing 5.1. Besides the ordinary definitions of guard and effects, a transition label satisfying the `STRING` rule may be defined, alternatively. This way it is possible to specify transition labels that do not conform to the actions' syntax. This permits to use KITS not only for describing valid SyncCharts models, but also "pseudo code". Finally, the optional `history` flag may be set by appending the corresponding keyword.

```
Signal:
    (StringAnnotation)*
    'input'? 'output'? 'signal' ID (':=' AnyValue)?
    (
      (':' (ValueType | STRING))
     |
      (':' 'combine' (ValueType | STRING) 'with' (CombineOperator | STRING))
     )?;

AnyValue:
    Boolean | INT | Float | ID | STRING;

enum ValueType:
    'pure' | 'boolean' | 'unsigned' | 'integer' | 'float' | 'host';

enum CombineOperator:
    'none' | '+' | '*' | 'max' | 'min' | 'or' | 'and' | 'host';
```

Listing 5.6: Grammar of the `signal` declaration in KITS.

Signal and variable declarations

In order to complete the introduction of the statements used in the ABRO example, the `signal` statement still needs to be explained. In the SyncCharts language, signals are understood as a form of events, and they may also serve as interfaces and memories of data. The signal declaration grammar rule is given in Listing 5.6.

After the usual opportunity of attaching annotations, signals may be declared to be an input, an output, to serve as input and output or none of them, i. e., local signals. A signal must have an identifier that has to be unique within its parent scope, i. e., the state or region it is attached to. Nested children of a signal's parent scope may contain further signals with the same identifier. Thus, a redefinition of signals is possible (as allowed for variables in some programming languages, e. g., C).

Afterwards, an initial value and a formal type may be determined if the signal is valued. Otherwise the type is set to `pure` by default.

While executing specifications containing valued signals, semantic conflicts might occur if such a signal is emitted concurrently in different regions with different values. Such conflicts can be resolved by attaching a *combine function* according the syntax in Listing 5.6, line 7. Valid combine functions must be associative and commutative. Predefined ones are listed in Listing 5.6, line 14. Alternatively, custom combine functions denoted denoted by their plain name (string) may be used.

Besides the signal concept, SyncCharts is equipped with variables as mentioned in the previous section during the presentation of ThinKCharts. However, they are not subject to the synchrony hypothesis as strictly as signals. Whilst signals are often used to exchange information between regions or to trigger preemptions, variables are mostly in charge of preserving values that are visible within a certain region. The grammar rule of variable definitions is given in Listing 5.7.

```
Variable:
    (StringAnnotation)*
    'var' ID (':=' AnyValue)? ':' (ValueType | STRING);
```

Listing 5.7: Grammar of the variable declaration in KITS.

Further elements of KITS

Now some of the auxiliary language elements of KITS, which are part of the kernel statements, are outlined. The definition of the `Action` declaration, which is referenced by the grammar rule of the `state` statement, is already given in Listing 5.1 on page 54. Actions in turn consist of Boolean expressions and effects. Due to the complexity of the expressions grammar, refer to Appendix A for the definition of (`Boolean`)`Expression`. Effects may occur in different forms as illustrated by the related grammar rule given in Listing 5.8.

```
Effect:
    Emission | Assignment | TextEffect;

Emission:
    [Signal] ( "(" Expression ")" )?;

Assignment:
    [Variable] ":=" Expression;

TextEffect:
    HOSTCODE ("(" ID ")")?;
```

Listing 5.8: Grammar definition of effects in KITS.

The emission of a signal is expressed by just mentioning its name. If the signal is valued and the value is to be updated, the expression describing the new value enclosed in parentheses is appended. Assignments of variables are similar to assignments in common imperative programming languages. The `TextEffect` allows to provide arbitrary code that does not need to conform to KITS. Such code must be enclosed by single quotes according to the `HOSTCODE` terminal rule ('...'), and may be followed by the parenthesized name of the language it belongs to.

The language fragment `TextualCode` mentioned in the definitions of the statements `state` and `region` basically serves the same purpose. It allows to attach arbitrary code to states and regions. Its grammar rule is given in Listing 5.9.

```
TextualCode:
    'textual' 'code' ( '(' ID ')' )? ':' HOSTCODE;
```

Listing 5.9: Grammar definition of `TextualCode` in KITS.

Annotating Models

The last part of this section addresses the annotation of models and model elements. Annotating and commenting contents, specifications, or instructions is common practice in almost all established textual programming or description languages. Accordingly, such comment features are provided by modern LDKs and can be added to domain specific textual syntaxes very easily. As seen in the previous section attaching secondary information to graphical representations of models is possible as well. These techniques, however, turned out to be insufficient while dealing with dynamically created representations as they work on views of models, not the models themselves. Precisely, an annotation or comment attached to a certain model element in a certain view will get lost if the view is dismissed.

In order to overcome this shortcoming, annotations must be added to the semantic elements. Unfortunately, domain specific data structures created by means of the common meta modeling framework EMF do not provide generic annotation hooks. Consequently, custom hooks must be added to the domain specific meta model allowing to attach different forms of annotations to actual model elements. In order to separate the annotation concerns from domain specific entity classes and their relations, a dedicated meta model, the annotations meta model, has been created. It is depicted in Figure 5.12.

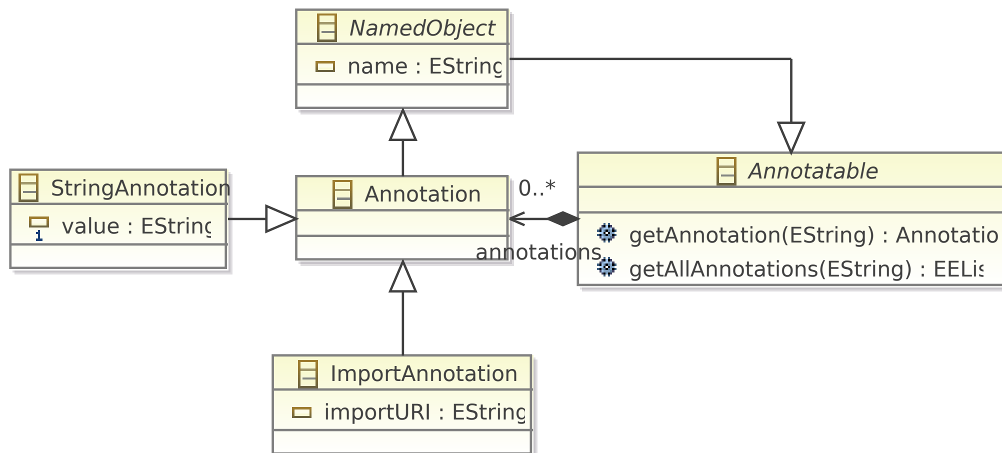


Figure 5.12: The annotations meta model.

This generic meta model is supposed to be employed by domain specific meta models. In order to equip domain specific elements with annotatability, designated classes have to specialize the abstract class Annotatable. Custom annotation types can be introduced by means of further classes specializing Annotation.

This way annotatability has been introduced into the SyncCharts meta model, which was presented in the previous section on page 56. As observable in the Figures 5.10 and 5.11, the classes Scope, Action, and ValuedObject specialize Annotatable.

Besides the annotations meta model, which forms the abstract syntax definition, related concrete syntax fragments have been designed allowing to represent various kinds of annotations in KITS specifications. The first one presented here is the *comment annotation*. It defines how *semantic comments* that shall be attached to elements of domain specific models have to look like. In contrast to usual program comments, which are just syntax and will be ignored by every compiler, interpreter, or other processing tools, these semantic comments are part of the specification. Consequently, semantic comments are not bound to certain model representations, e. g., a textual one, but rather are part of the model itself, and can be represented in multiple ways.

The concrete syntax of semantic comments is adopted from the syntax of Javadoc [51] annotations. A semantic comment can be any arbitrary string enclosed in the keywords `/**` and `*/`. This is implemented by the dedicated `COMMENT_ANNOTATION` terminal rule. Listing 5.10 shows the related grammar definitions.

```
1  CommentAnnotation:
2      COMMENT_ANNOTATION;
3
4  // custom terminal rule introducing semantic comments
5  terminal COMMENT_ANNOTATION:
6      '/**' -> '*/';
```

Listing 5.10: Grammar definition of comment annotations.

Besides annotating model elements with secondary information in plain text there are more use cases of annotations. Imagine that a model might be annotated with information guiding a compiler or interpreter, e. g., layout hints. Other information might setup the layout engine if graphical representations of models shall have a certain shape. These informations, which are also non-functional, can be expressed by means of *key-value-annotations* very comfortably. The syntax of this kind of annotation is inspired by the Javadoc tags [51]. Thus, a key identifier satisfying the `ID` terminal rule (see page 58) with a preceding `@` is followed by a value. This value must satisfy `ID` as well, or be any arbitrary string enclosed in double quotes as given by the `STRING` rule. The grammar rule is given in Listing 5.11.

```
1  KeyValueAnnotation:
2      '@' ID (EString)?;
3
4  EString:
5      ID | STRING;
```

Listing 5.11: Grammar definition of key-value-annotations.

In order to provide a generic "interface rule" to the annotation syntax definitions, the `StringAnnotation` grammar rule, given in Listing 5.12, has been introduced. It delegates to the alternatives `CommentAnnotation` and `KeyValueAnnotation`, as

yet, and is open to further extensions.

```
StringAnnotation:
    CommentAnnotation
    | KeyValueAnnotation;
```

Listing 5.12: Grammar definition of `StringAnnotation`.

The use of both kinds of annotation shall be demonstrated by means of the former `ABRO` example. Recall that all of the essential entities of a SyncCharts model, i. e., states, regions, signal and variable declarations, and transitions can be augmented by the above defined `StringAnnotations`. Consider Listing 5.13.

```
/** ABRO is the 'hello world'
    example of synchronous
    programming languages */
@author "chsch"
state ABRO {
  input signal A
  input signal B

/* C is not needed right now */
//  input signal C

  input signal R
  output signal O

  @layoutDir "down"
  init state ABO {

    /** macro state WaitAB with
        parallel regions */
    init state WaitAB {

      @layoutDir "right"
      region WaitA:
      init state wA
      --> dA with A
      final state dA

      @layoutDir "left"
      region WaitB:
      init state wB
      --> dB with B
      final state dB
    }
    >-> done with / O
    state done
  }
  /** resetting ABRO */
  o-> ABO with R
}
```

Listing 5.13: `ABRO` with annotations.

The model is augmented with a comment providing a short explanation of the given content and a tag indicating its author. This information will be preserved while persisting the whole content, as it is encapsulated in semantic annotations. In contrast, syntactic comments like in lines 9 and 10 will not be added to the model while loading and get lost if the model is not persisted in shape of the text but, e. g., in an object-oriented data base[1]. Besides further semantic comments, key-value-tags have been added to regions forming hints to layout engines (lines 15, 22 and 28).

In order to bridge the gap to the graphical world, strategies on how to represent such annotations in graphical representations must be found. A proposal on integrating the annotations introduced in Listing 5.13 into the diagram of ABRO (Figure 5.2) is given in Figure 5.13.

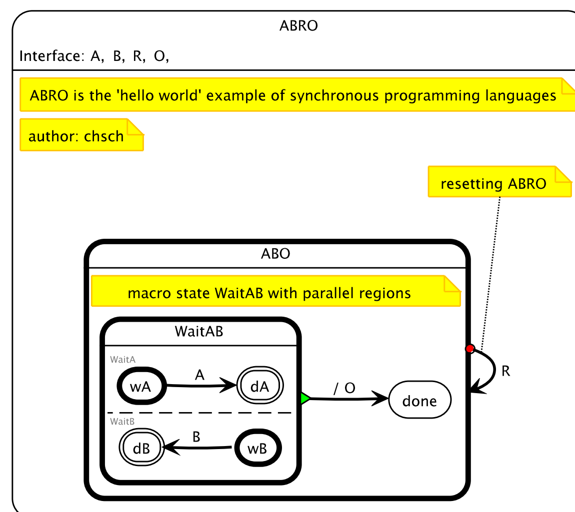


Figure 5.13: ABRO with annotations.

In this proposal all semantic comments are depicted by means of GMF's graphical notes. The same holds for the 'author' tag. The layout hints, however, are not put into such a note but applied while arranging the elements of the diagram.

An open question is the concrete placement of the notes. Since this question is beyond the scope of this work, solely some related issues are mentioned shortly.

In the above diagram annotations of macro states, e. g., "macro state WaitAB..." are placed into the figure of the macro state. Alternatively, they might be placed outside nearby the state figure, and optionally connected by a dotted line. The question of where to place graphical annotations seems even harder to be answered for simple states. In other words: does it make sense to convert the representation of a simple state into a macro state figure enabling the accommodation of notes inside? Last but not least, arranging annotated diagrams adequately is not captured by any

[1]For information on object-oriented data bases visit, e. g., http://www.odbms.org

of the available layout methods explicitly.

Summary

The KITS language is a textual representation of SyncCharts models. It has been designed to enable the synthesis of alternative views on such models. The implementation has been realized by means of the Xtext LDK, which provides parsing, printing, and user interface tooling.

In contrast to the common workflow of defining a syntax, obtaining a derived meta model and specifying model to model transformations in order to get instances of a desired meta model, the KITS tooling has been directly built upon the SyncCharts meta model. Due to the close integration with established Eclipse modeling components, KITS can be used to persist SyncCharts models. In addition, an infrastructure has been developed allowing models to be decorated with secondary information in form of various kinds of annotations.

In order to support the reusability of the annotation and expression language, parts of their definitions have been moved to dedicated grammars. They are given in Appendix A. Finally, customizations like syntax highlighting, formatting rules, and sophisticated serializer modifications have been added.

5.3 Integration of Transient Textual Views

Based on ThinKCharts and KITS an implementation of the concept of *dynamic transient* model representations has been developed. As emphasized in Section 4.3, this strategy involves an editor, which is in charge of the persisting issues, and views[2] that are initialized with model contents.

For this purpose, a dedicated Eclipse view has been installed. It provides a textual editing field that is used to display the KITS representation of models and enables typical operations such as *copy and paste*. In accordance with the conceptual design, the view maintains its own runtime instances of SyncCharts models. In the following this view is referred to as the 'KITS view', in order to prevent confusions with views on models. It is shown on the right of Figure 5.14, delineated by a blue rectangle.

While modeling with ThinKCharts, the KITS view is involved in the following way: if a SyncCharts diagram is loaded, the KITS view is initialized with the textual representation of the diagram's model. If a further model is opened, the previous content is dismissed and replaced by the latter model. The same holds, if the focus changes from any editor to a ThinKCharts editor, e. g., while switching between SyncCharts models and/or other documents.

In addition to this sensitivity on alternating the active editor, the content of the KITS view may be limited to a certain part of the chosen SyncCharts model. The

[2]Recall the ambiguity of the term view that either denotes a representation of a model according to the MVC, or, here, an Eclipse view part, i. e., a user interface component.

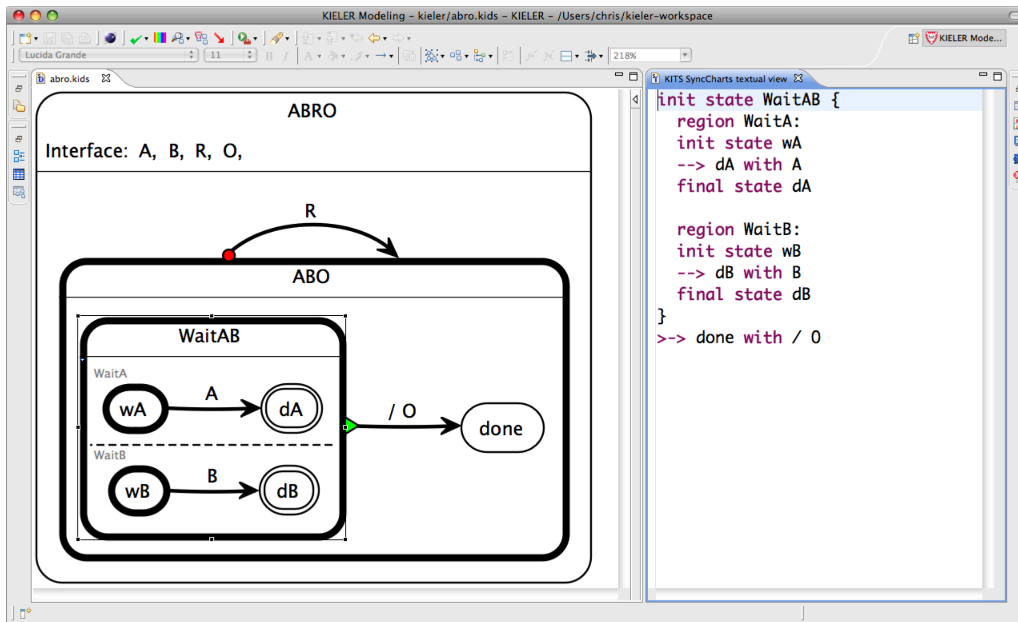


Figure 5.14: Integrated graphical and textual views in ThinKCharts.

modeler can determine such a model part like a state, a region, or a transition by just selecting the desired element in the graphical representation. This is demonstrated in Figure 5.14. Instead of displaying the whole model, the content of the textual view is restricted to the selected state WaitAB and its children. The definition of ABRO and its related signals A, B, R, and O is hidden.

Changes in the textual view are transmitted to the editor immediately after they have been performed, provided the textual representation conforms to KITS' grammar and does not contain syntactical mistakes. This way, the divergence of the graphical and textual representation is limited to changes modelers are able to type without pausing less than a configurable delay, which currently has a default set to 1 second.

The propagation and incorporation of model modifications is delegated to the merging facility of EMF Compare. This follows from the requirement of preserving as much elements of the graphical editor's model as possible. Recall from Section 3.5 that GMF-based editors pursue canonical edit policies. Thus, such editors update their representations whenever model elements are touched. Hence, adding or removing such elements is uncritical, since related concrete syntax elements must be added or removed, anyway. However, replacing a model element, or even a compound sub model, will result in the removal and re-creation of related figures. This leads to the loss of the entire secondary information such as size and position in the diagram. In order to avoid this, the minimal invasive method of comparing the models and merging the differences, as described in Section 3.3, is applied.

While the current implementation uses this synchronization technique for transmitting changes from the textual to the graphical view, i. e., from right to left ac-

cording to Figure 4.5 on page 45 and Figure 5.14, the reverse way is not realized by means of this technique, but covered by initialisation of the KITS as described above.

There are various reasons for this strategy. The first one is GMF's policy for creating elements: if, e. g., a signal shall be added to the model, the signal object is created in a first step, whereas its name is set in a second independent transaction. The state of the model between these, however, is invalid with respect to the textual representation. Recall from the previous section that the name of signals is a mandatory property. Consequently, a generic *refresh-after-element-addition* policy will not work.

A further, even more important issue is of generic nature. Models treated by means of their graphical representation conform to their metamodel since this is ensured by the set of possible operations provided by related graphical editors. In other words, graphical editors do not allow the creation of invalid models with respect to their metamodels. This assumption, however, does not hold for (modifiable) textual representations and textual editors. In general, the mutation of a single character is sufficient to invalidate the whole model. This characteristic affects the dynamic-view-approach as shown by means of the following scenario:

Let the KITS view display a model representation that has been corrupted such that particular model parts cannot be parsed correctly. If the modeler performs various changes on the graphical view that would be propagated to the textual view, the content of the resulting updated textual representation is rather unpredictable, as the serializer preserves broken content that cannot be parsed properly. If the view tries to propagate its model to the graphical editor, afterwards, the chaos is almost perfect. Thus, in order to avoid such mishaps, the KITS view is re-initialized after almost any significant action that is performed on the graphical view, as most of them are accompanied with changing the selection.

5.4 Remarks on the Implementation

After explaining the integration of different views on SyncCharts models from a modelers point of view, details on the implementation as well as rationales for seemingly odd approaches of realizing the particular steps are discussed.

Whereas there was no need for touching ThinKCharts, the KITS grammar has been developed first. While elaborating the elements of the concrete syntax, various of additional implementations have been created. These Java classes tailor the scoping (cross-reference resolution/content assist), the linking process, the formatting, and constraints on the serialization of the Xtext-based KITS editor. These customizations are due to properties of the metamodel, e. g., the EOpposite relation of the (bidirectional) state target association of the transition class, and agreements concerning use of the languages such as the printout of the `region` keyword and the serialization of transitions' triggers and effects vs. the printout of the label string.

Afterwards, the tailored Xtext editor had to be ported to a view. This is generally accompanied with much effort, since concrete editors inherit a lot of functionality

from abstract editor implementations that is not available for views. Fortunately, an implementation that was formerly created by the Xtext developers and refined by the community satisfies the requirements and has been used.

Based on this preliminary work the initialization of the view has been realized. The particular steps that are involved in this process are outlined in Figure 5.15. First, a copy is of the entire model is created. This copy, however, is not needed until the synchronization part, as explained later on. Copying models involves two steps:

a) creation of new domain elements and establishing their parent-child-relationship according to the original model, and

b) setting up cross references such as the target states of transitions.

Whereas a) can be delegated to EMF's runtime library, predetermined cross-references are mostly wrong (e. g., point to the original model) or not set at all in case of bidirectional associations. Thus, step b) is achieved by an own implementation.

Determining the correct model element to be referenced can be achieved by inquiring the referenced object in the original model and obtaining the pendant by means of their fragment URIs. This method may fail in certain situations, though. Therefore, the EMF Compare's match mechanism has been leveraged in order to determine pairs of original and copy, and building up a local look-up-table. This way a safe identification of referenced, in short *linked*, model elements is ensured. As both methods require the model to be contained in a resource, it is added to a dummy resource beforehand, which will never be persisted to the disc.

Step 4 involves the identification of the model element in the copy with respect to the selected part in the diagram, and its handover to the KITS view. In order to transmit potential modifications back to the graphical view, some information must be kept previously:

- What is container element of the element to be moved?
- At which association is it attached to its container? For example states are attached to the association *states* of regions (see the metamodel in Figure 5.10 on page 56)
- If the container association is a list, at which index is the element located?
- Do other elements reference the current one, i. e., is it linked by other ones? For example states may be linked by transitions.

The final refreshing of the textual view is performed behind the scenes by an implementation of `IDocumentEditor`, as explained in Section 3.7.

Finally, it is worth noting that a unique so-called embedded Xtext editor, which is deployed in the KITS view, is not sufficient. This is because an embedded editor that it has to be initialized with a dedicated EMF resource, which in turn is linked to a particular parser. These parsers, however, are fixed with respect to their start

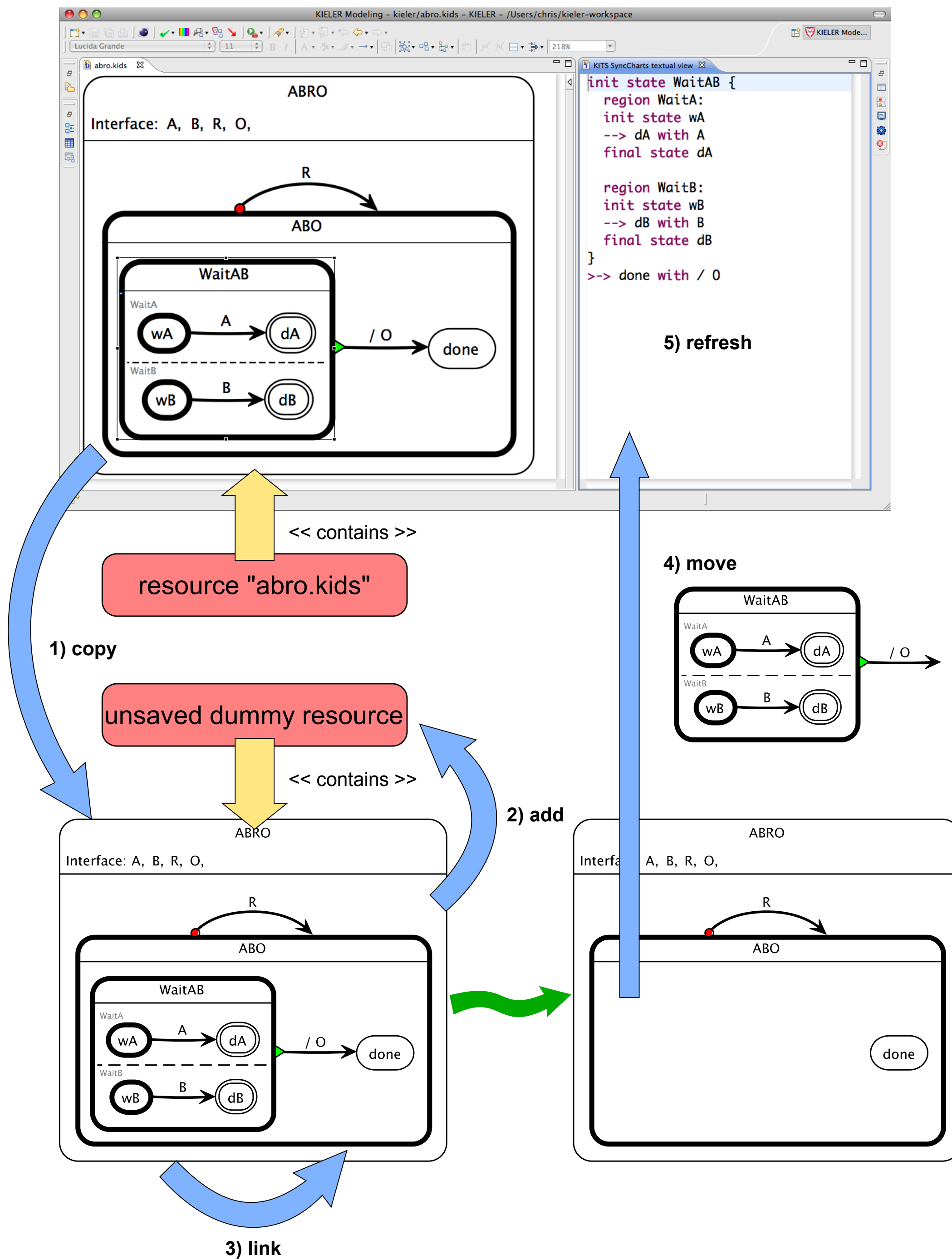


Figure 5.15: (Re-)Initialization of the KITS view: Triggered by a selection, here the state WaitAB, the process of initializing involves (1) copying the entire model, (2) attaching it to a dummy resource, (3) assure correct cross-references, and (4) moving the copied counterpart into the KITS view, which (5) refreshes itself afterwards.

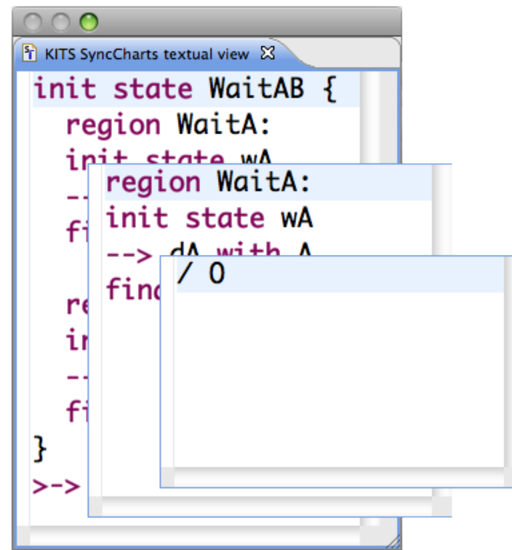


Figure 5.16: Multiple embedded text editors stacked on each other.

rule. Hence, three parsers have been generated, expecting the root elements *state*, *region*, and *action*, used by three dedicated embedded editors. These are attached to the KITS view altogether and arranged by a *stack layout policy*. Depending on the type of the selected element in the diagram, the related embedded editor is brought to top. This is depicted in Figure 5.16

The transmission of modifications performed in the KITS view works as shown in Figure 5.17. If the model derived by the parsed content of the KITS view changes, a synchronization action is scheduled. If further modification occur in a short amount of time, the synchronisation will be delayed accordingly. When synchronization starts, the model of the textual view is retrieved, and copied. Afterwards, this copy is inserted into the hole of the former copy of the entire model, which has been created while initializing the view, by means of the information explained above. This includes putting the sub model at the right position and redirect potential links, here transitions. Subsequently, the entire model is linked again, in order to make sure that all cross references are correct.

After preparing the "new" entire model, it is compared to the variant maintained by ThinKCharts. If this comparison reveals differences, a dedicated merge command, which is compatible to the transaction mechanism used by GMF, is scheduled. During its execution EMF Compare's `MergeService` is called in order to modify the model behind the graphical view accordingly. In addition, labels of transitions and state actions are re-created. Finally, ThinKCharts' canonical edit policy instances as well as KIML's layout service are called in order to refresh the diagram.

EMF Compare's default match and diff implementations rely on various metrics, each of them with a certain weight, in order to determine whether elements are similar or not. One of these metrics is the equality of their fragment URIs. This criterion

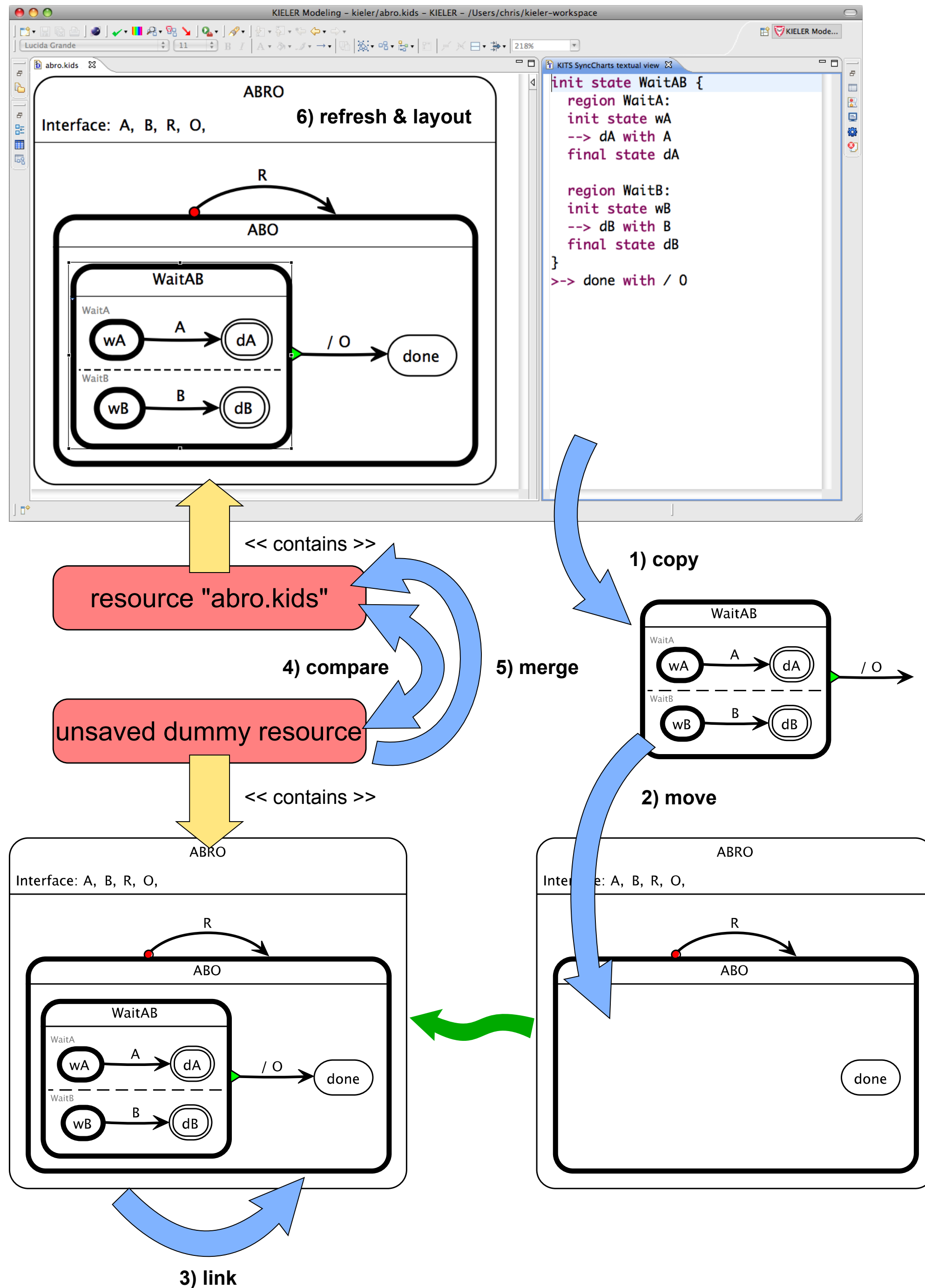


Figure 5.17: Transmitting modifications to ThinKCharts: Triggered by modifying the KITS view's content, the process involves (1) copying the model part maintained by the KITS view, (2) inserting it into the skeleton, (3) restoring the cross-references, (4) comparing the obtain model with the original one, and (5) merging potential differences into the model maintained by ThinKCharts, whose active diagram is (6) refreshed and rearranged afterwards.

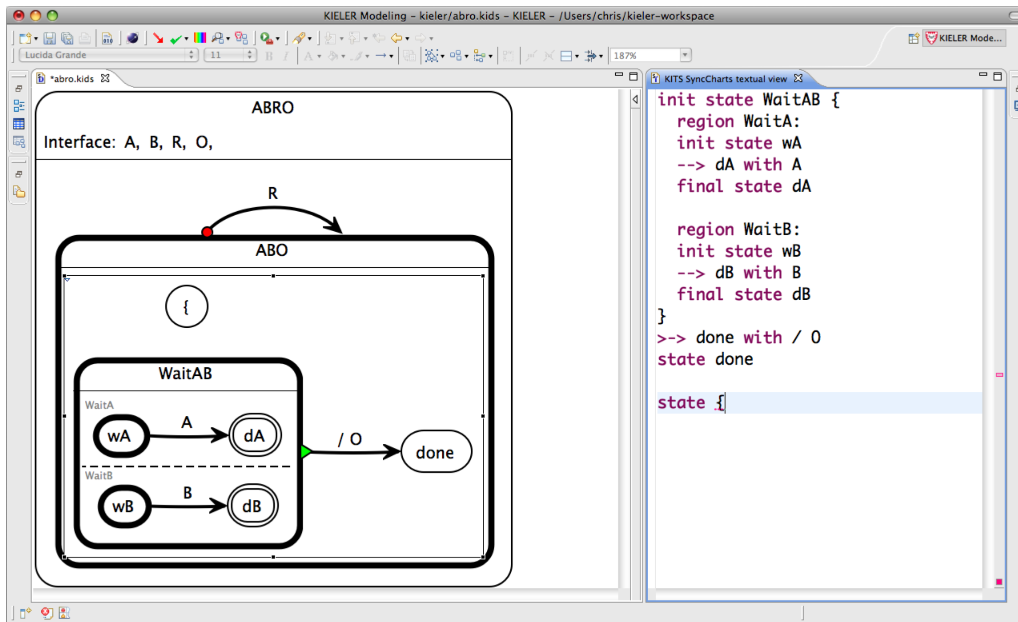


Figure 5.18: Exemplary glitch occuring while modeling with the KITS view.

is even associated with a high weight compared to other ones. Since the default engines are hardly influenceable (apart from copying their code into a custom implementation) this weighting cannot be changed. This precludes, e. g., the comparison of model sections while their locations in their containing models shall be ignored. For this reason the entire copy of the model maintained by ThinKCharts is needed. Similar issues come up with the default merging implementations. These problems can only be fixed by fully reimplementing the related merge method. Hence, although EMF Compare is intended to be widely customizable, doing so is rather challenging. In contrast, a rather simple tailoring is the instruction to ignore certain associations or attributes of model elements while comparing them. This has been done for one way of the bidirectional associations between transitions and states.

5.5 Open Issues

Although the overall result of this solution is very satisfying, occasionally some kind of "glitches" occur while working with the KITS view. They are due to Xtext's *partial background parsing strategy*, potential mistakes in the generated parser, and the two stage delay of processing modifications in the text. This delay consists of the delay between typing and parsing as well as the delay between parsing (and updating the model) and synchronizing. An example of such a glitch occurs while creating a state with refinement shown in Figure 5.18. In this situation – the modeler forgot the mandatory identifier – the opening brace is faultily considered to be the id. Although this should not happen due to the definition of the `ID` terminal rule, a related state

is preliminarily added to the model and merged into the graphical representation. Fortunately, this will be corrected with subsequent modifications of the KITS view's content by the modeler that transform the textual view into a valid one.

Lastly, an issue that is due to the internals of GMF is the notable freeze of the user interface during the merge process. Since the merging eventually involves changes in the user interface, the execution of the synchronization command must be performed by the UI thread of the Eclipse workbench. Whereas this is tolerable during slight changes, it might be confusing while writing down bigger model parts. This issue shall be investigated in the future, since it need not necessarily be due to the runtime frameworks, but some specifics added to ThinKCharts, e. g., the label handling. Another reason might by the missing integration with KiVi, which is also left to future work.

Chapter 6

On Integrating Modeling Language Families - The MENGES Example

Since the feasibility of the transient view approach is proven by the realization in KIELER, the concept is supposed to be leveraged in the domain-specific modeling environment that is being developed by the MENGES team.

The main objective of MENGES is the supply of a modeling environment that provides assistance throughout the compulsory development stages of safety-critical control systems for the rail-bounded transportation field. Such stages are

- the requirements engineering supplemented by an analysis of constraints due to the law,
- the design and implementation, and
- the verification of design vs. requirements as well as implementation vs. design.

The development of safety-critical railway signaling systems requires special Domain-Specific Languages (DSLs). Since such systems are composed of different parts, different languages are defined by the MENGES developers. Each of this languages is dedicated to certain system parts and aspects. An overview of those system parts and their relations is given in Figure 6.1. The shape of each language may be graphical, textual, or a mixture of both, i. e., a graphical language accommodating textual fragments. In certain cases it makes sense to have a representation of the content in both worlds, as we will see in Section 6.2. For convenience initial (textual) drafts of such DSLs have been developed by means of the Xtext framework.

In contrast to previous approaches, where systems were modeled in a rather informal way, the new languages are supposed to be complete in the sense that no further knowledge besides the model content is required in order to conceive the specified content or perform computations on it. However, only a limited amount of information targeting a certain aspect of the whole system shall be expressed in a certain language. Thus, the specification of the whole system results from the individual descriptions collectively.

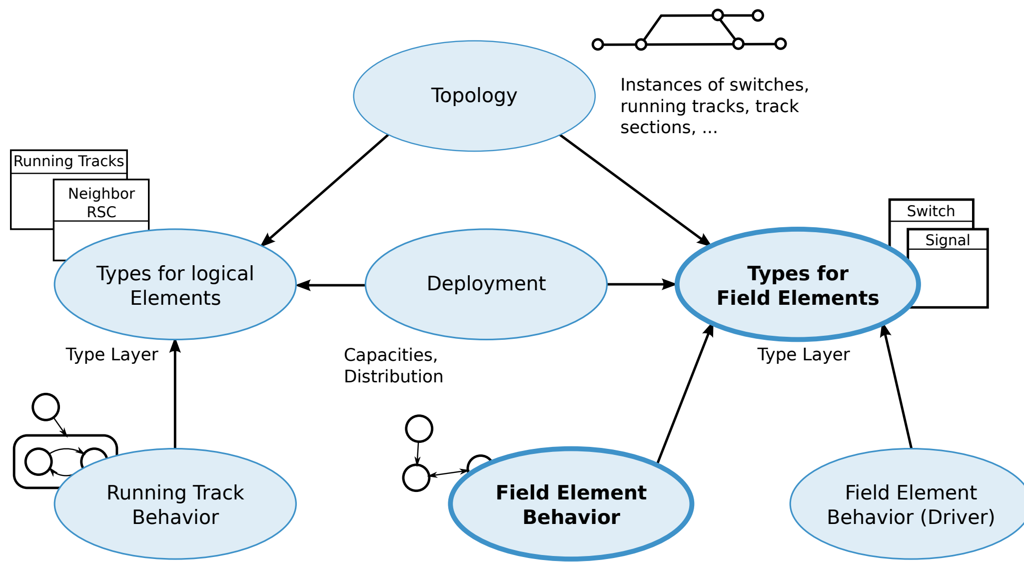


Figure 6.1: Kinds of specifications of an interlocking block design.

In addition to assistance while formulating specifications, the tool to be constructed must to be able to perform simulation and verification tasks. Furthermore, it shall assist the engineers in the documentation process, a very time consuming task. Finally, target code generation and deployment will complete the extent of functionality of the project delivery.

Since the new generation of interlocking blocks shall be developed in product lines, the separation of generic parts and specific parts is to be preserved throughout the entire development process. Therefore, the realization of given procedures in a reusable way, e. g., the switching of points or the assignment of routes, is to be separated from conceptual plannings and designs of concrete railroad lines. A further branch is about composing complex interlocking blocks of local ones, which in turn are composed of basic control units. These basic controllers are dedicated to railroad elements such as track sections, switches, and signals. Such railroad elements are mostly referred to as *field elements*.

In the first part of this chapter, the focus is on a preliminary behavior specification language, called *GALogic*, which is used to describe the behavior of the field element controllers. The language is outlined by means of grammar excerpts and example specifications. Based on this, the creation of alternative representations, here graphical ones, is described. This activity is motivated by the typical characteristics of compact textual representations. As mentioned earlier in this work, text-based model creation and maintenance is efficient, whereas exploration and communication of textually represented models is often cumbersome. Since the models and modeling

languages are very prototypical, the creation of the graphical views is performed in accordance with the first approach discussed in Section 4.2. Subsequently, advantages of such a representation as well as limitations and open issues are discussed.

Based on the prototypical graphical representations, potential use cases and scenarios to be facilitated by the product of MENGES are proposed. The basic assumption of these proposals is the decoupling of model representations from concrete files. Instead, they shall be created on demand by means of the techniques developed earlier within this work.

6.1 The behavior specification language *GALogic*

GALogic is a means for behavior specification, which abstracts any details of implementation or potential target platforms. It is situated in the middle of the bottom row of Figure 6.1. *GALogic* models describe the dynamic characteristics of given entities, which in turn are introduced by means of a language called *Feldelemente* (field elements). This language is represented by the right node in the middle row of Figure 6.1. *Feldelemente* models define abstract controlling units for field elements and describe their static properties. This includes defining their interface, declaring memories, and listing possible values of state variables. In the following the term *field element* denotes the basic railway elements mentioned above, or it refers to the abstract control units, which are related to the elements and defined by *Feldelemente* models. In the context of this chapter the term is mostly used to denote the latter ones.

The field element controllers specified in *Feldelemente*, and enhanced by instances of *GALogic*, are to be composed by means of a further, still unnamed language targeting the instantiation and deployment of such elementary controlling units to stations, lines, and further abstract organizational units. It is represented by the central node of Figure 6.1. The execution semantics of model compounds formulated in these languages is similar to an actor oriented model of computation, as proposed by Lee et al. [30], implying that each field element has to provide a function or method that is going to be invoked periodically. This feature, however, is not determined yet, too.

Let us turn to the definition of the language. Listing 6.1 shows the start rule of *GALogic*'s grammar. Skipping the import declarations, which are merely auxiliary information needed by the tooling, the *StateMachine* and *RuleBlock* parts are considered in some detail.

```
1 Model:
2     (Import)*
3     (StateMachine)*
4     (RuleBlock)* ;
```

Listing 6.1: Top level rule of the grammar of *GALogic*.

```
state machine Gleisabschnitt_beansprucht_SM {
    references beansprucht in Gleisabschnitt;
    transitions
        start nicht    --> DWeg, FLR, FWR, FWZ;
        DWeg           --> nicht, DWeg_FLR, FWZ_DWeg;
        FLR            --> nicht, DWeg_FLR, FLR_FLR, FWR_FLR, FWZ_FLR;
        FWR            --> nicht, FWR_FLR;
        FWZ            --> nicht, FWZ_FLR, FWZ_DWeg;
        DWeg_FLR       --> DWeg, FLR, DWeg_FLR_FLR;
        FLR_FLR        --> FLR, DWeg_FLR_FLR, FWR_FLR_FLR, FWZ_FLR_FLR;
        FWR_FLR        --> FLR, FWR, FWR_FLR_FLR;
        FWZ_DWeg       --> DWeg, FWZ;
        FWZ_FLR        --> FLR, FWZ, FWZ_FLR_FLR;
        DWeg_FLR_FLR   --> DWeg_FLR, FLR_FLR;
        FWR_FLR_FLR    --> FWR_FLR, FLR_FLR;
        FWZ_FLR_FLR    --> FWZ_FLR, FLR_FLR;
}
```

Listing 6.2: Example of a state machine in *GALogic*.

```
StateMachine:
    'state machine' ID ('extends' [StateMachine])? '{'
        'references' [feldelemente::StateVar] 'in' [feldelemente::Feldelement] ';'
        'transitions'
            (TransitionSet)*
    '}';

TransitionSet:
    ('start')? ('final')? [feldelemente::State] '-->' Transition (',' Transition)* ';';

Transition:
    [feldelemente::State] ;
```

Listing 6.3: Grammar of state machines in *GALogic*.

In *GALogic* behavior is expressed in a heterogeneous way: If the field element of interest exhibits sequential behavior, valid transitions of its state variables are to be listed. The group of a state variable and its transitions is called a *state machine*. Guards and effects of transition specifications are omitted, in order to allow very compact representations of those state machines. An example of such a state machine is presented in Listing 6.2, the grammar of the construct is given in Listing 6.3.

Most of the field elements are composed of several state machines, where some of them are of a multiple of the complexity compared to the example mentioned above. Additionally, those state machines do not act in isolation. The behavior of the field elements is rather determined by the concert of all of the involved state machines, meaning that the execution of a transition in state machine A may be dependent on the active state of the state machines B, C, etc. This is a further argument for separating guards and effects of the state machines, as one has to consider the Cartesian product otherwise.

This consideration leads us to the rule blocks, the syntax for specifying the guards

```
1  rule block Bedienkommando_WU references Weiche {
2    rule graph
3      --> WU [(command WU && weicheIstStellbar())] {
4          -> [istBeanspruchtInFahrstrasse()] /
5               { call weichenLage_aendern(),
6                 endlageUeberwachung := false,
7                 sollLage := !sollLage };
8          -> [(!istBeanspruchtInFahrstrasse() && !sollLage)] /
9               { call weichenLage_aendern(),
10                endlageUeberwachung := false };
11       }
12 }
```

Listing 6.4: Example of a rule block in *GALogic*.

and actions of transitions in a compound way. For convenience, a decision tree-like syntax was chosen. Consider the example in Listing 6.4: the statement `rule block` encapsulates the major component `rule graph` and optional declarative parts such as formal parameters, local predicates, and local procedures, which are absent in this example. The rule graph contains the decision tree for determining actions to be executed in the current state of the field element, as well as computing the successor state. Syntactically a decision tree is introduced by `-->`, whereas alternative actions are indicated by a leading `->`. In addition, decision trees may occur in a nested shape resulting in a side-by-side of sub decision trees and actions.

The rule blocks are inspired by the so called *Wenn-Dann-Diagrams*, a concept that was previously developed by the industry. In addition to the overall structure of the rule blocks, which has been adopted from those diagrams, they have inherited the weakening of the tree property, allowing some kind of syntactic sugar: rule graphs may also reference other named rule graph elements, i. e., actions or sub-graphs declared within the same rule graph. This introduces the opportunity of re-using of specified objects. Such reference alternatives are indicated by leading `->` $*$.

The example in Listing 6.4 also includes `command WU`, which is an interface to a specification at a lower lever of abstraction, e. g., a device driver. In the example it acts as a presence test of the event WU. `weicheIstStellbar()` is an alias of a compound Boolean expression; `weichenLage_aendern()` is an alias of a compound action. All of them are declared in the related *Feldelemente* description of `Weiche`. A further rule block example, which contains references, is shown in Listing 6.5; the grammar excerpt defining the `rule block` statement is given in Listing 6.6.

Since the above mentioned *Wenn-Dann-Diagrams* are of a graphical shape, it is desired to provide the developers with similar diagrams. These representations may assist the engineers during the active development in form of immediate feedback. Furthermore, it might help them to get used to the new languages if they are provided with a representation they are already familiar with.

```
rule block reserviere references Gleisabschnitt () {
  rule graph
    --> [istReservierungsVorbedingung()] {
        -> Fall1 [beansprucht == nicht] / {
            reservieren1()
        };
        --> Fall2 [true] {
            -> Fall3 [beansprucht == DWeg] / {
                reservieren2()
            };
            --> Fall21 [true] {

                -> Fall211 [ beansprucht == FLR ] / {
                    reservieren3()
                };
            }
            -> Fall22 [beansprucht == FLR_FLR] / {
                reservieren4()
            };
            -> * Fall211
        }
        -> * Fall2
    }
}
```

Listing 6.5: A rule block containing reference alternatives.

```
RuleBlock:
  'rule block' ID 'references' [feldelemente::Feldelement]
    '(' (FormalParameter (',' FormalParameter)*)? ')' '{'
      ('local predicates' (GuardDeclaration)*)?
      ('local procedures' (ActionDeclaration)*)?
       'rule graph' RuleGraph
  '}';

RuleTreeOrReference:
    RuleGraph | RuleTreeReference;

RuleGraph:
    GuardedAction | GuardedRuleTree;

GuardedAction:
    '->' (ID)? '[' Be ']' '/' Action ';' ;

GuardedRuleTree:
    '-->' (ID)? '[' Be ']' '{'  (RuleTreeOrReference)+ '}' ;

RuleTreeReference:
    '->' '*' [RuleGraph] ;

Be: ... // Boolean expressions

Action: ... // atomic and compound actions
```

Listing 6.6: Grammar of rule blocks in *GALogic*.

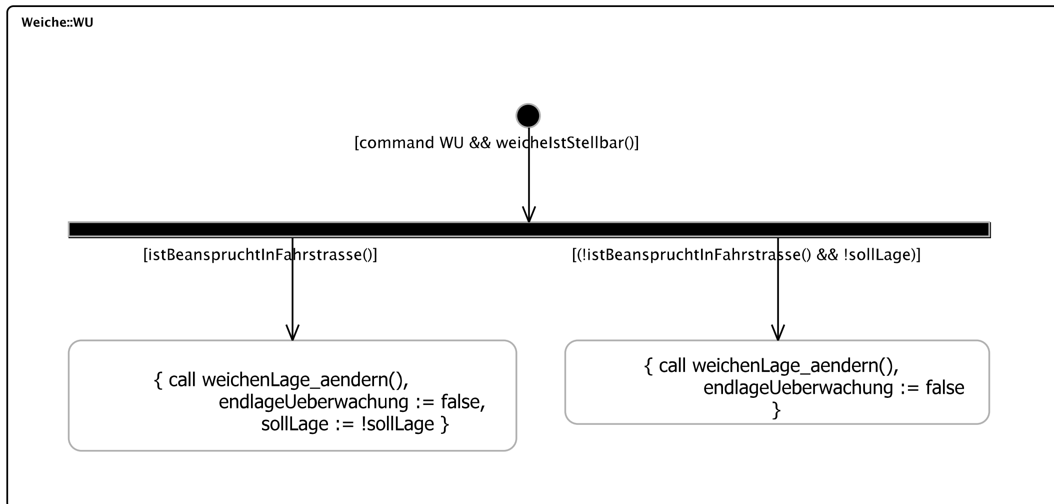


Figure 6.2: Rule graph of rule block defined in Listing 6.4 laid out manually.

6.2 Providing Graphical Views

Textual syntax and textual language development is a very promising approach for quick model prototyping, especially in the early stage of a project. The available frameworks allow to create and modify languages and appropriate tooling almost instantaneously. While a textual representation is well-suited for static information such as declared in *Feldelemente*, dynamic content might get difficult to discern, at least if the complexity increases. In this case a well-chosen graphical representation may help. Therefore, this section presents proposals for visualizing the state machines and rule blocks.

Choosing a graphical syntax

First a visual language, which represents the content to be expressed in a very intuitive and concise way, has to be chosen. For the state machines the numerous graphical Statechart languages seem appropriate. A candidate might be the *SyncCharts* language, which has been presented in the previous chapter, since a rich editor providing appealing graphical views is available (ThinKCharts).

A graphical counterpart of the rule blocks, however, is much less obvious. Therefore, the approach presented here uses a variant of the graphical representation used to draw the *Wenn-Dann-Diagrams*, the UML *Activity* diagram, has been chosen. An example of such a rule block diagram is given in Figure 6.2, representing the rule graph of rule block `WU` (Listing 6.4).

While the declarative parts (formal parameters, local predicates, and local procedures) are skipped in this draft, the focus is on the rule graph. The entry of a rule graph is marked by a symbol known as initial node in the Activity Diagram and UML State Machine language. The initial node refers to a representative of an element

of type `RuleGraph`. This reference is depicted by an arrow. From the definition of `RuleGraph` in the grammar (Listing 6.6, line 13) follows that a `RuleGraph` element may be a `GuardedAction` or a `GuardedRuleGraph`, alternatively. `GuardedAction` elements mark endings in the decision graph, i. e., decisions to be made. They are represented by rounded rectangles labeled with their effects in textual form. In contrast, `GuardedRuleGraph` elements express branches between several paths in the decision making process. They are visualized by means of a horizontal bar similar to the *fork* construct in the Activity Diagram language. Decision alternatives are indicated by outgoing arrows, which are labeled with textual guard expressions. In addition, `GuardedRuleGraph` elements may be nested, allowing to split the decision making process, i. e., the examination of guard expressions, into multiple stages. This, for example, can be used to extract common parts of the guards of a set of actions. Note that the language defined this way can be considered as an *internal DSL*, which is hosted by the UML Activity Diagram language (although an explicit specification of its semantics is still missing).

After introducing the chosen graphical rule block language, appropriate tooling support must be found. An open implementation of the Activity Diagram language is contained in the Papyrus tool chain, which has been presented in Section 3.6. Papyrus is built upon EMF, which suggests a homogeneous integration with the Xtext-based textual *GALogic* editor and common model transformation frameworks. Furthermore, the *multi-part-concept* of the Papyrus editors, i. e., the ability of accommodating multiple diagrams within a document, increases Papyrus' suitability. By exploiting this feature, a one-to-one mapping of textual specification documents and visualization documents might be possible. If a specification contains multiple rule blocks, each one is represented by a dedicated diagram and accessible via the tab list on the bottom of the Papyrus editor.

Besides the Activity fraction of the UML, Papyrus supports UML *State Machines* as well. Thus, dismissing ThinKCharts and deploying the Papyrus infrastructure for presenting *GALogic* state machines with the Papyrus state machine editor is an immediate consequence. Finally, Fuhrmann et al. [12] proposed a connection of the KIELER Infrastructure for Meta Layout (KIML) and Papyrus, which provides automatic layout for graphical UML-based models formulated by means of this tool. Therefore, Papyrus has been considered as an adequate choice for realizing initial drafts of graphical view on *GALogic* models.

Synthesizing views

The Papyrus editors are based on GMF, as outlined in Section 3.6. This implies the strict separation of the model elements to be depicted and the meta information describing the picture itself. Whereas the domain metamodel of the Papyrus editors is given by the Ecore implementation of the *UML2*, the metamodel of *GALogic* is generated by Xtext according to its grammar. In order to synthesize graphical representations, a model transformation has been realized according to the concept that is discussed at the beginning of Section 4.2. This transformation inspects the

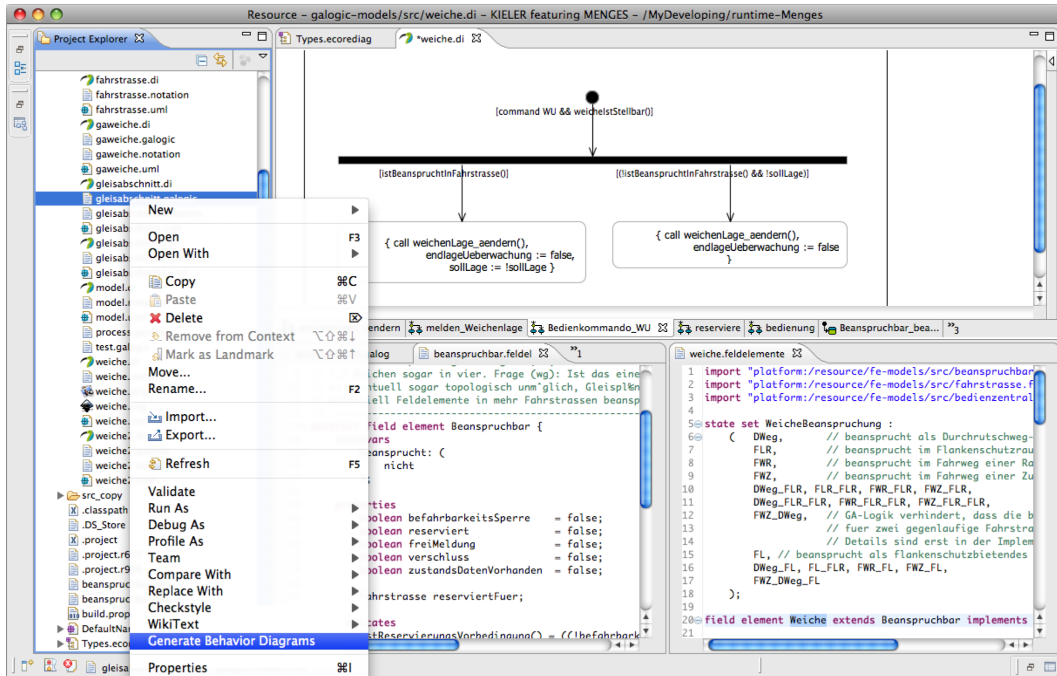


Figure 6.3: Invocation of the behavior diagram generation.

textually specified behavior model and produces a related instance of *UML2*. Due to the absence of any ability to initialize diagrams of existing models in Papyrus, the diagram description of the state machine or rule block of interest must be synthesized, too. Last but not least, Papyrus' sash window model has to be generated and linked. This model is a data structure that is maintained by Papyrus editors. It accommodates configuration data of the multi-part-editor, e. g., the ordering of the diagrams.

The transformation rules have been specified in the model transformation language *Xtend*, which was explained in Section 3.4. At the beginning the model independent parts such as the sash window model are created. Going further, the state machines and rule blocks are examined sequentially. While descending into their contents, the related semantic and notational elements that constitute the diagram are constructed and configured. This includes namings and other labeling on the semantic side, as well as decorative elements and default settings of sizes and fonts on the notational side. Furthermore, the connections between elements are added to the models.

The surrounding implementation is organized as follows: via an entry added to the context menu of *GALogic* files an event handler can be triggered, as shown in Figure 6.3. Doing so, it first loads the *GALogic* model(s) and their related *Feldelemente* specifications. Second, it applies the model transformation onto each model. Since the transformation builds up the sash window model, the notation model, and the semantic model simultaneously, and these models refer to each other in this order, the root element of the sash window model is returned. Eventually, the models have

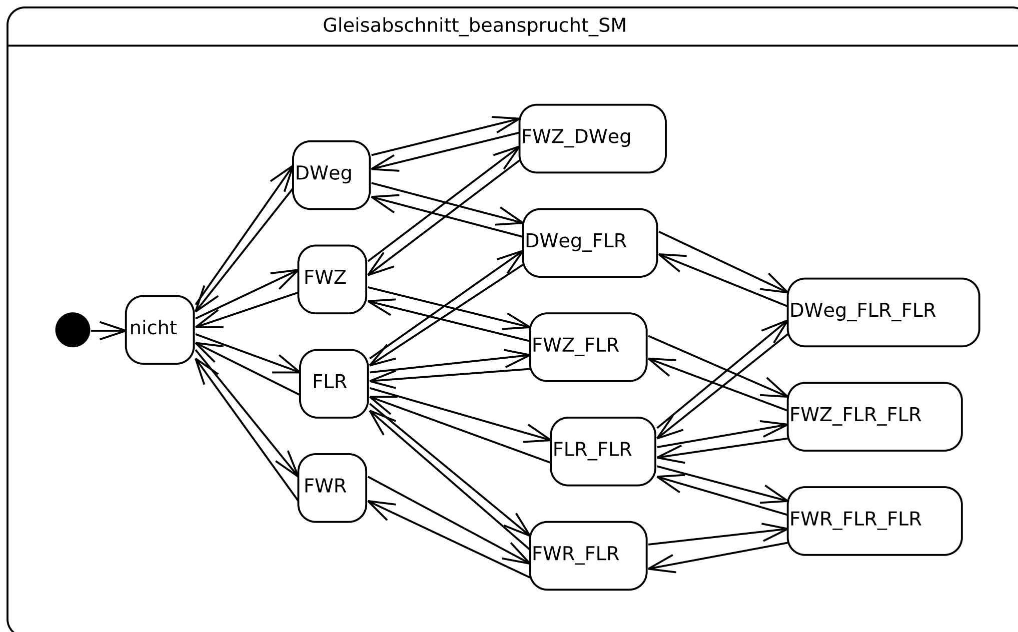


Figure 6.4: State machine defined in Listing 6.2 arranged with *KLay Layered*.

to be stored in separate files with dedicated file extensions. For convenience the newly created document is opened subsequently.

Layouting views

The remaining part of this section addresses the layout problem of the synthesized views. Arranging contents of a graphical representation is likely to be the most crucial part of the whole task. The layout of a diagram usually has a huge impact on the utility of graphical representations for given contents.

The Figures 6.4 and 6.5 show visualisations of the state machine defined in Listing 6.2, which are arranged by means of KIELER. The first diagram shows a layout that is generated by the *KLay Layered* algorithm. It implements the layer-based graph drawing approach, which has been proposed by Sugiyama et al. [50]. The basic principle of this method is the assignment of the particular nodes to layers according to its distance form a certain start node. In the diagram of Figure 6.4 the start node is given by the initial node of the state machine.

The second diagram shows a layout that is created by the *Circo* layouter, which is part of the *Graphviz*[1] library. It is based on work of Six and Tollis [46] as well as Kaufmann and Wiese [25]. This layout method tries to identify strongly connected components in the graph, and arranges the elements of such components circularly.

[1] http://www.graphviz.org

Although this diagram is appealing, it may mislead the observer since the represented state machine does not exhibit circular behavior. Therefore, the layer-based method should be preferred for these kinds of state machines. Note that the above example is a rather special one, since it has a very regular structure. Because of this structure the layouts determined by the *DOT* algorithm, which is also part of *Graphviz*, are rather dissatisfying, although it generally produces good results.

While layouting state machines is supported by KIELER, as seen above, arranging the graphical rule graphs, which in fact are activity diagrams, is still an open problem. Precisely, sizing the *fork* bar is not captured by any layout method at all. Furthermore, the left-right-ordering of the decision branches contains semantics in form of a priority of the particular alternatives. In order to address this issue a custom layouting policy has been developed, which takes advantage of the regular structure of the rule graphs. Reference branches, which are complicating the layout problem, are ignored in this initial draft. Based on this prerequisite layout data may be computed in a depth-first-search-like way.

Considering the early stage of language development within MENGES, the proposed graphical notation is not approved, so the development of a full layout algorithm targeting this syntax will possibly not pay off. Instead, the layout method above sketched has been implemented in another model transformation, which works on the notation model. It is attached to the process in form of a post-processing step of the diagram synthesis.

The transformation rules descend along the notational elements forming the rule graph. For each stage the elements are placed one by one next to each other beginning on the left-most possible position with a certain offset in the vertical direction compared to elements in the parent stage. This results in a layering reflecting the depth of the elements in the graph. As mentioned above the method acts in a depth-first-search way. Hence, placing a rule graph element causes the placement of its children, too. Nodes representing elements of type `GuardedAction` are leaf elements and need no further attention. Their size is estimated by the amount of text contained by its label during the diagram synthesis. The computation of the size and position of nodes denoting elements of type `GuardedRuleGraph` or `RuleTreeReference` is more complicated.

Consider the representatives of `GuardedRuleGraphs` first: After their alternatives have been positioned, the width of the representing black bar is set to the sum of widths of its alternative representatives and the margins enclosed. Now the placement of objects achieves a result like in Figure 6.2. However, GMF-based editors anchor connections without any configuration in the middle of the most nearby bound of the connected target and source elements. Hence, leaving the outgoing connections untouched would cause all of them to start in the middle of the south side of the fork bar, resulting in a spider net of connections and overlaid labels. To fix this, a source anchor decorator is added to each outgoing arrow, assuring straight vertical connections (on the target side the arrows are anchored in the middle according to the default policy). This leads to a connector layout like in Figure 6.2.

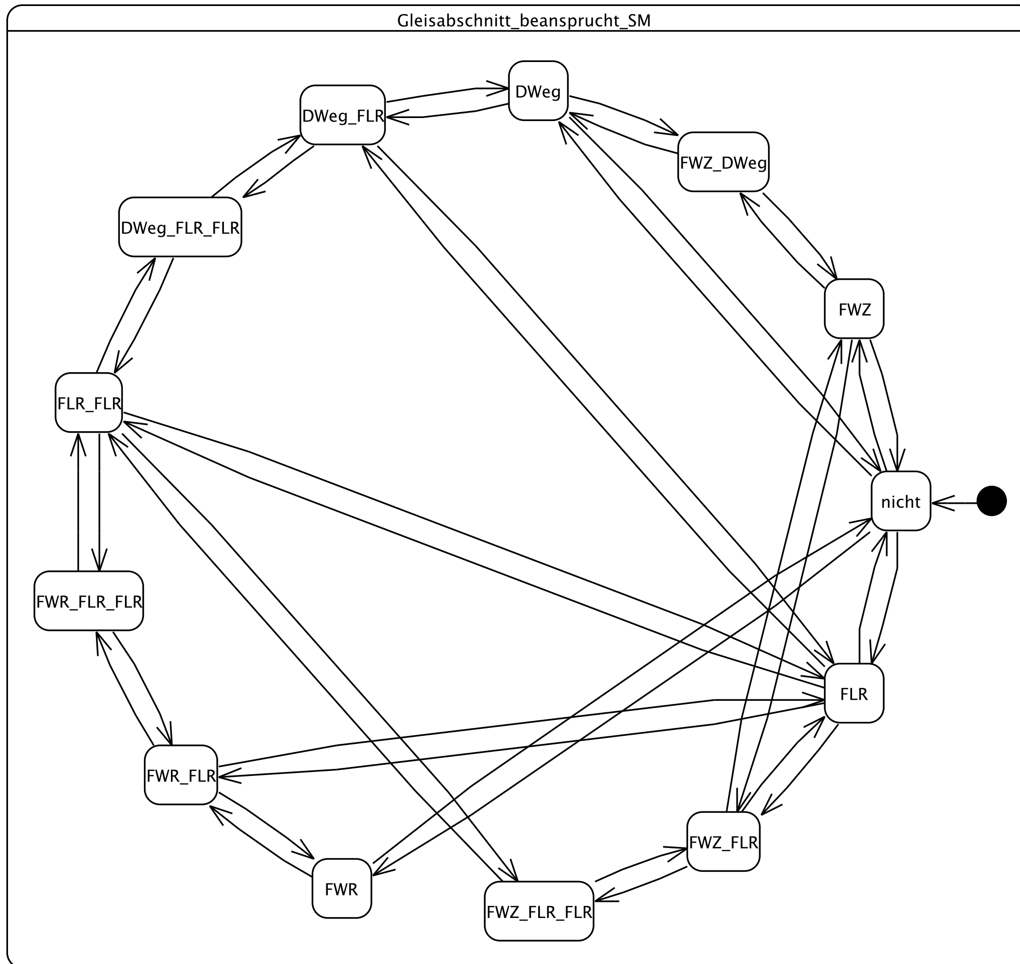


Figure 6.5: State machine defined in Listing 6.2 arranged with *Graphviz Circo*.

Based on those connector constraints the GMF runtime takes care about the label placement in an adequate way.

While arranging diagrams of rule graphs satisfying the tree property is rather straightforward, the layout computation of DAG-like graphs cannot be handled properly with this approach. DAGs arise if rule graphs contain `RuleTreeReferences`. Trying to perform the layout computation leads to some sorts of conflicts. The first kind occurs if representatives of decision alternatives have been positioned once and are considered repeatedly resulting in their re-placement. Further conflicts come up during the determination of the fork bar widths, i. e., due to accumulating the width of action or sub-graph representatives multiple times. Finally, it is unclear how to arrange the outgoing connections of the fork bars. Therefore, the layout method has been supplemented with little modifications, e. g., in order to avoid the re-placement of action nodes. By their means the method provides acceptable results on rather

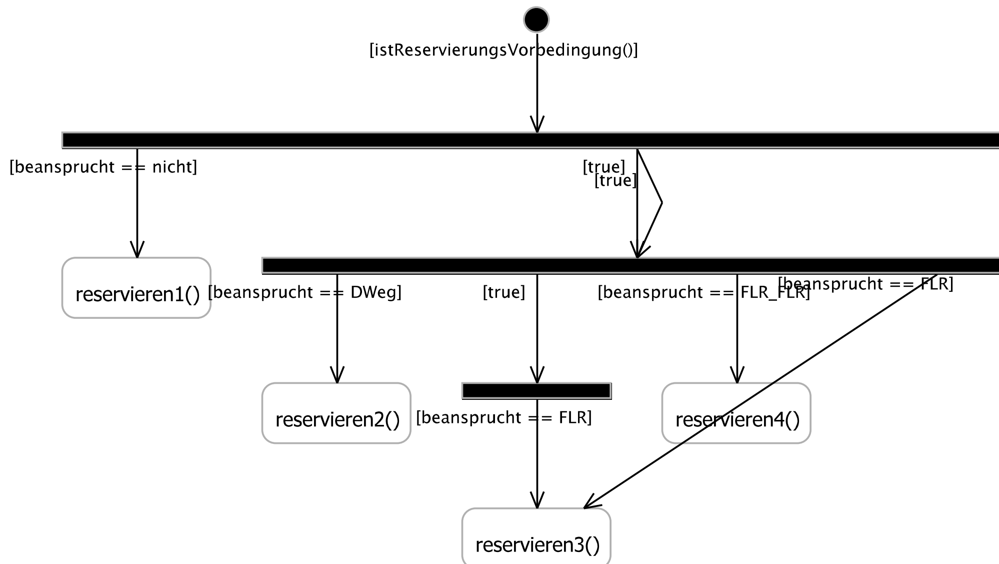


Figure 6.6: View on the rule block defined in Listing 6.5 with layout defects.

simple diagrams. These diagrams, however, should be manually post-processed, anyway, by placing bend points of e. g., edges that represent reference alternatives. An example of such a diagram is given in Figure 6.6 visualizing the rule graph from Listing 6.5.

Finally, after establishing the positions of all action and rule graph representatives as well as their connections, the initial nodes are centered corresponding to the width of the referred top level rule graph representative.

6.3 Assessment

This method for providing alternative views on models proved to be helpful in various ways. On the one hand, modelers are served with representations of their models they are already familiar with. This supplies a comfortable means to check the content they modeled. On the other hand, *graphical views* themselves can be examined and assessed. By means of examples like the pictures given in this section, the form of representation as well as the amount and level of detail in a certain graphical view may be discussed. In addition, further representations, which are based on the given proposals of graphical representations, and which address different purposes and/or different stakeholders, might be defined.

While the proposed rendering of state machines involved in *GALogic* fits very well the needs of the developers, the drawings of the rule blocks exhibit certain immature characteristics. These kinds of defects result from the design of the graphical syntax as well as the limited options of the chosen rendering tool, i. e., the Papyrus UML

Activity editor. The most obvious one, observable in Figure 6.2, is the centered alignment of the label text in the boxes representing actions, which is not alterable.

Another more general disadvantage of this approach is the lack of flexibility concerning the available structural elements and labelings. Prefabricated editors, which are built upon the common technologies, have a dedicated set of shapes and figures to be used in diagrams. The addition of further elements is not possible at all. Consequently, a dedicated editor has to be created in order to render views of models that fit the demands of the several addressees. Furthermore, the present automatic layout methods should be improved. For example the nodes of a diagram, which are assigned to a certain layer, shall be centered in its layer, rather than be oriented according to their connection, as observable in Figure 6.4, page 86.

Besides technical reasons for arguing for a full-custom editor, the syntax itself should be subjected to a revision. Notably, the substitution of the fork bar inherited from the Activity Diagram language is suggested. This construct is not captured by any diagram layout method yet and is definitely blocking the layouting of rule blocks by means of commonly approved layout algorithms.

In conclusion, the implementation effort of the method presented in this section was surprisingly low. This qualifies it as a very appropriate method in the early stage of language design. Note that the synthesized views need not necessarily be of graphical nature – such a view on a model may also be of textual shape. Nevertheless, if the graphical or textual languages evolve to fully-fledged DSLs, custom rendering tools become essential.

6.4 Towards Interactive Model Browsing & Editing

The technique proposed in the previous section generates input data for graphical editing tools. Such representations, however, are not linked to the original model data anymore. Hence, modifications of those graphical representations do not affect the models. Furthermore, representations of models generated this way are typically restricted to chosen structures or fractions of the models. Obtaining tailored overview representations on modeled systems, which are composed of numerous specifications, is hardly possible. In particular, the collaboration and interaction of specific modeled entities, e. g., cause and effect relations, are difficult to visualize. An example of such shortcomings in context of the formerly introduced means for specifying and visualizing sequential behavior is the following: Due to the loss of knowledge about the particular transitions in a *GALogic* model after the synthesis of the views, no semantic relation between view and model elements is available. Since the same holds for the rule blocks, the graphical visualization of the *described-by*-relation between state transitions and actions of rule blocks is not possible with the above approach.

In order to improve this situation, this section explores the application of the transient view approach for visualizing textually specified models that are formulated in the *GALogic* language. Therefore, various use cases of handling and exploiting such models are identified and investigated. Proposals on the realization of these use

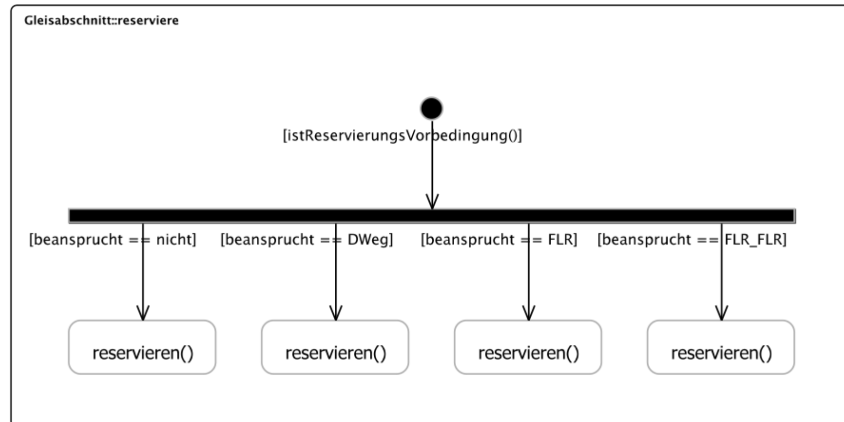


Figure 6.7: Graphical view of a rule block employing predicates and procedures.

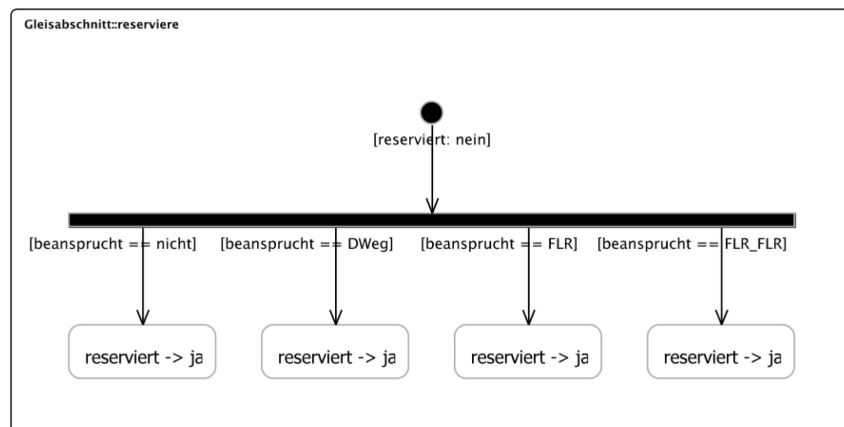


Figure 6.8: Graphical view after the expansion of predicates and procedures.

cases are explained by means of the model representations presented above.

Transient Views on MENGES-Specific Models

Transient model representations that are not subject to a predefined and inflexible metamodel seem to be a powerful means in order to overcome the shortcomings mentioned above. By means of such views it is possible to visualize chosen model elements in a desired degree of detail. In contrast to the formerly presented graphical representations, for which the degree of detail is fixed, transient graphical views can be enriched with information while browsing them, since they can be (re-)computed and smoothly updated whenever this is needed. This, however, requires for adequate arranging and formatting technologies as stressed in Section 4.3. Furthermore, transient views can be used to edit models, as proven by the solution that is presented in the previous chapter.

The use case of varying the degree of detail is demonstrated by means of the

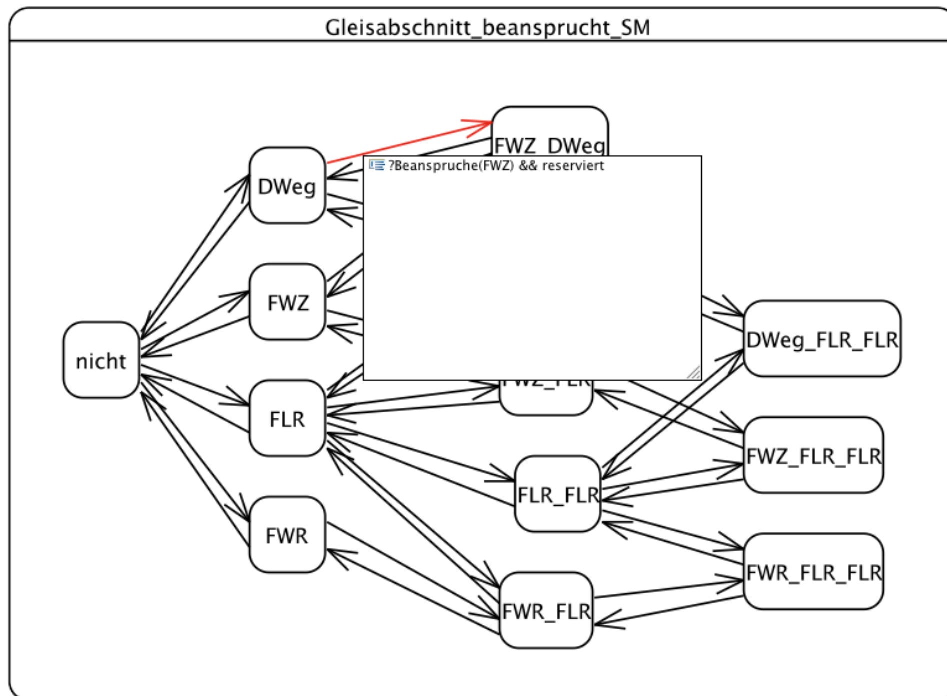


Figure 6.9: Exploring hidden details in a model diagram.

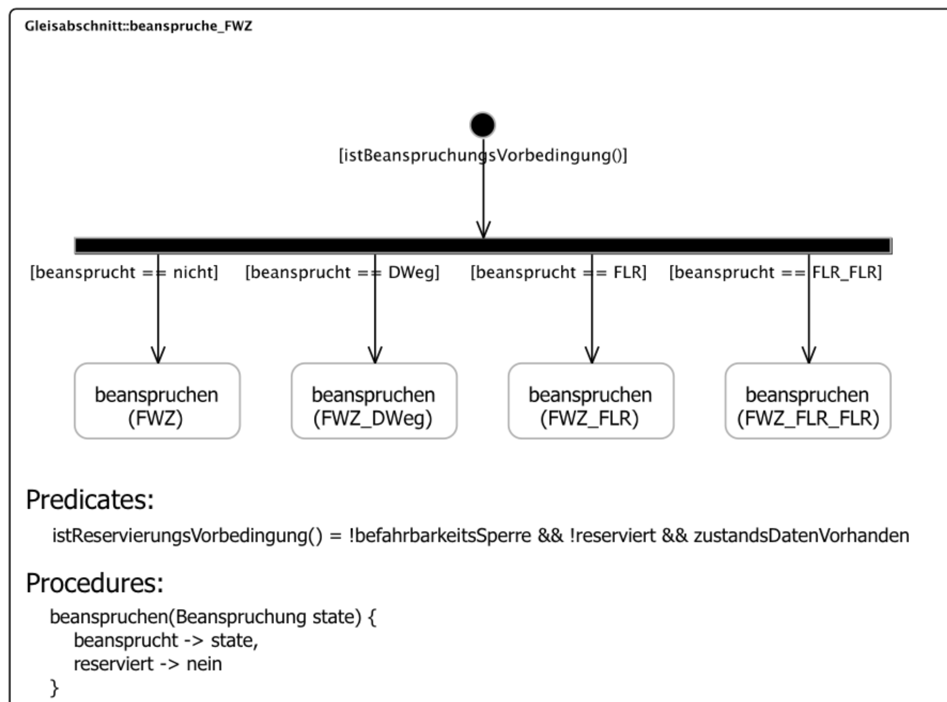


Figure 6.10: Graphical view supplemented with auxiliary information.

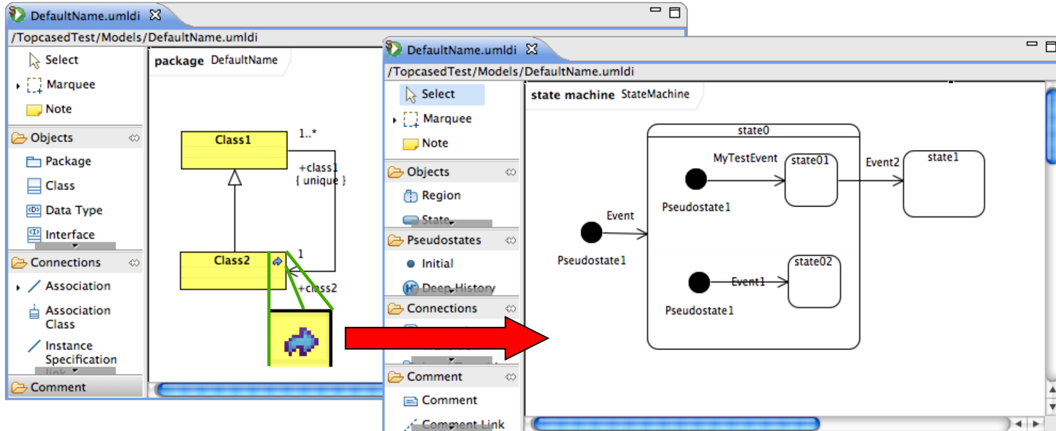


Figure 6.11: Navigation handles in graphical representations in Topcased.[2]

Figures 6.7 and 6.8. Figure 6.7 shows a graphical representation of a rule block that reflects the concrete definition of this model part. In contrast, the view depicted in Figure 6.8 has been refined such that calls of predicates and procedures are expanded in order to show the full specification content. This is assumed to be a rather simple but valuable feature while exploring and developing models onscreen. Note that the predicates and procedures used in this example are fairly short, and their expansion does not increase the diagram size. This will typically not hold in practice.

Another way of handling information that is an integral part of the model and that is, however, not properly placeable in a diagram is the dynamic display of text as shown in Figure 6.9. By means of this kind of hovering text fields, which have been formerly proposed by Scheidgen [43], graphical views can by extended such that the developer can explore and investigate the related models from different points of view. Recall in the context of the *GALogic*-based behavior specifications that guards and effects as depicted in the hovering field are not part of the state machine model. In order to provide this dynamically displayed information, mechanisms for deriving it from the rule block specifications are required. If this hurdle is cleared, such hovering fields can even be used to edit such model parts.

In the field of safety-critical systems, specifications must be confirmed by authorities. For this purpose large amounts of documentation for the crucial facts of design and implementation have to be produced. Therefore, model representations are required that do not hide details, but preserve their clarity. Hence, a way of representing behavior specifications dedicated to documenting purposes is proposed in Figure 6.10.

The creation of such views might be triggered in different ways. Starting from textual sources, a context menu entry, e. g., complemented with a keystroke combination, will be reasonable. Calling information from a graphical view or requesting

[2] http://www.topcased.org/

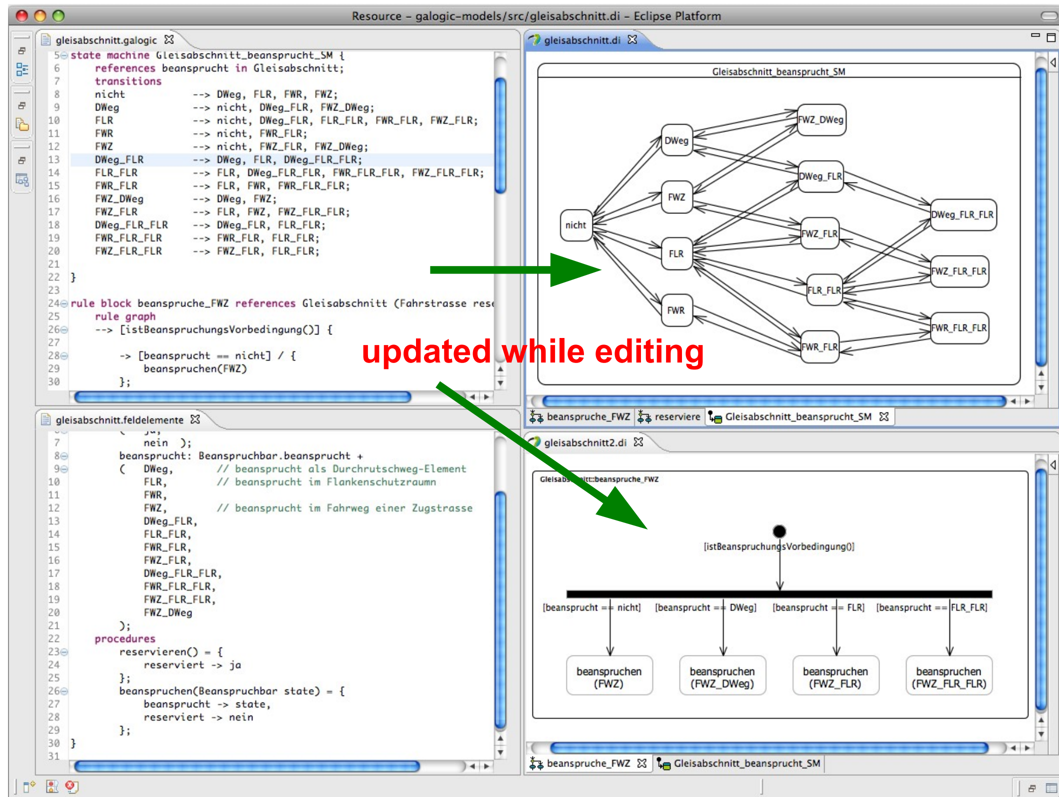


Figure 6.12: Principle of automatically updating opened views: Views on model elements that are requested by the modeler are updated, whenever depicted elements are changed.

it to change in a certain way may be enabled by graphical navigation handles, as available in the Class Diagram editor of the Topcased tool. This is illustrated in Figure 6.11. Apart from these explicit calls for particular views on models, which are performed by modelers or other stakeholders, transient views can be refreshed dynamically based on changes of the underlying model. In the context of the *GA-Logic* language this means, for example, that a graphical view of the state machine is updated whenever the related textual representation is modified. Such a scenario is depicted in Figure 6.12.

In addition to the previously proposed techniques, which basically concern a single representation of model elements, the handling of models may be improved by means that are based on the connection of model views. Precisely, this can facilitate the navigation within a compound of model elements, as well as the tracing of errors, which may occur while simulating or validating the modeled system. This applies to views displaying different aspects of model elements, as well as views exhibiting the same content in different ways. Especially in domain-specific graphical model editors, effective navigation means are mostly missing, although they might make life much

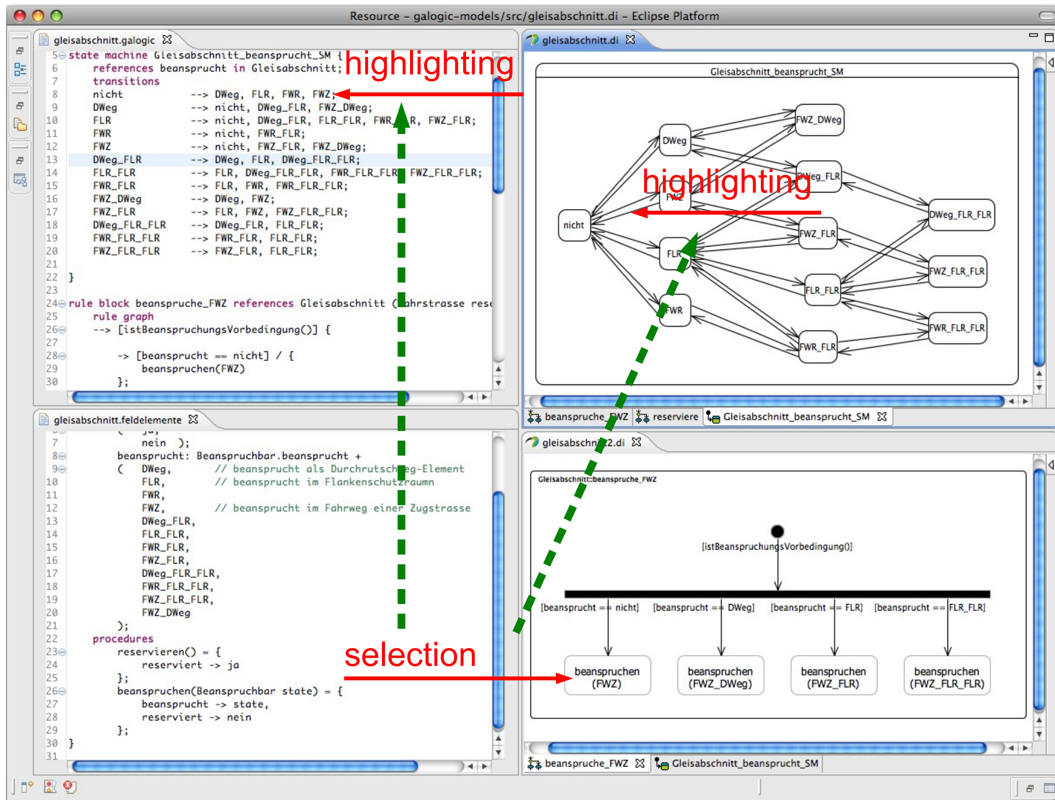


Figure 6.13: Interactive model exploration:
Views on model elements may be integrated such that they act in concert and lead the modeler through the specifications.

easier for modelers. A scenario that outlines an exemplary use case for such model navigation techniques is shown in Figure 6.13. In this setting the modeler opened the textual behavior description and its related field element definition. In addition, graphical views on a state machine and a rule block have been opened. To examine which state transitions are described by the rule block, the modeler may select an action in the rule block view. Based on this selection the related state machine(s) and their transition(s) correlating to the action are determined and highlighted in the graphical state machine view. In addition, the transition declaration in the textual base representation may be highlighted.

The capabilities of transient model representation are not restricted to model exploration activities. Similar to textual transient views, which may improve the editing of graphically formulated models as it has been realized in KIELER, graphical views may support the editing of models, too. Furthermore, advanced creation and modification methods such as *structure-based editing*, which has been proposed by Fuhrmann et al. [12], may simplify the editing of distributed models notably. For example, adding a new state and connecting it to other ones involves the modifica-

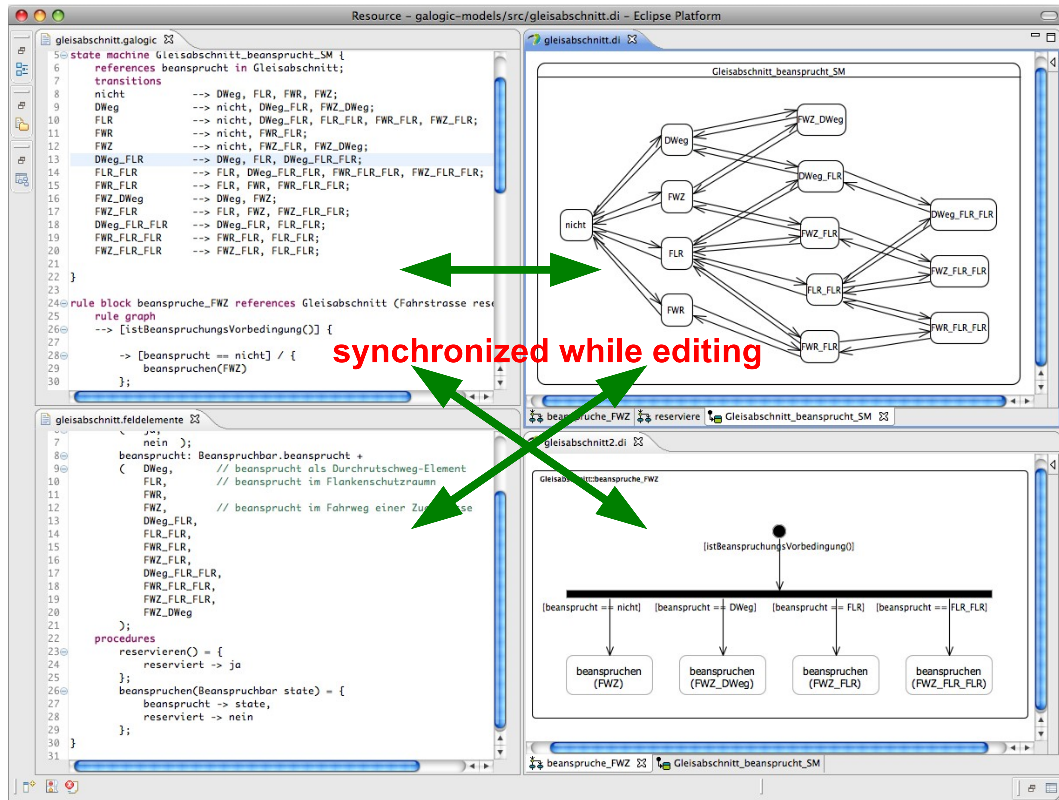


Figure 6.14: Multiform model editing by means of model views:
Model elements represented in views may be modified, removed, or new ones may be created. Such modifications can be transmitted into the source model by means of the proposed synchronization method

tions of at least two model files, since the set of states of a *GALogic* state machine is declared in the related *Feldelemente* description. Doing so in a graphical state machine view, e. g., by means of a compound editing operation, needs just a single keystroke combination in best case. The transmission of such modifications into the underlying models may be realized by means of the technique presented in Chapter 5. The principle of editing models this way is sketched in Figure 6.14.

The final idea to be discussed here is about handling field element specifications. Recall that these models are used to declare basic units that are in charge of controlling the various real world devices, which are needed in order ensure safe and reliable rail-bounded transportation. Since these controlling units share common properties and functions, an inheritance/specialization mechanism has been introduced in the *Feldelemente* language. This feature allows the derivation of a *refinement relation* amongst the *Feldelemente* models, which may also be depicted in a dedicated representation. An exemplary view on fictive field elements is given in Figure 6.15.

Besides generating such overview representations for documentation purposes, it

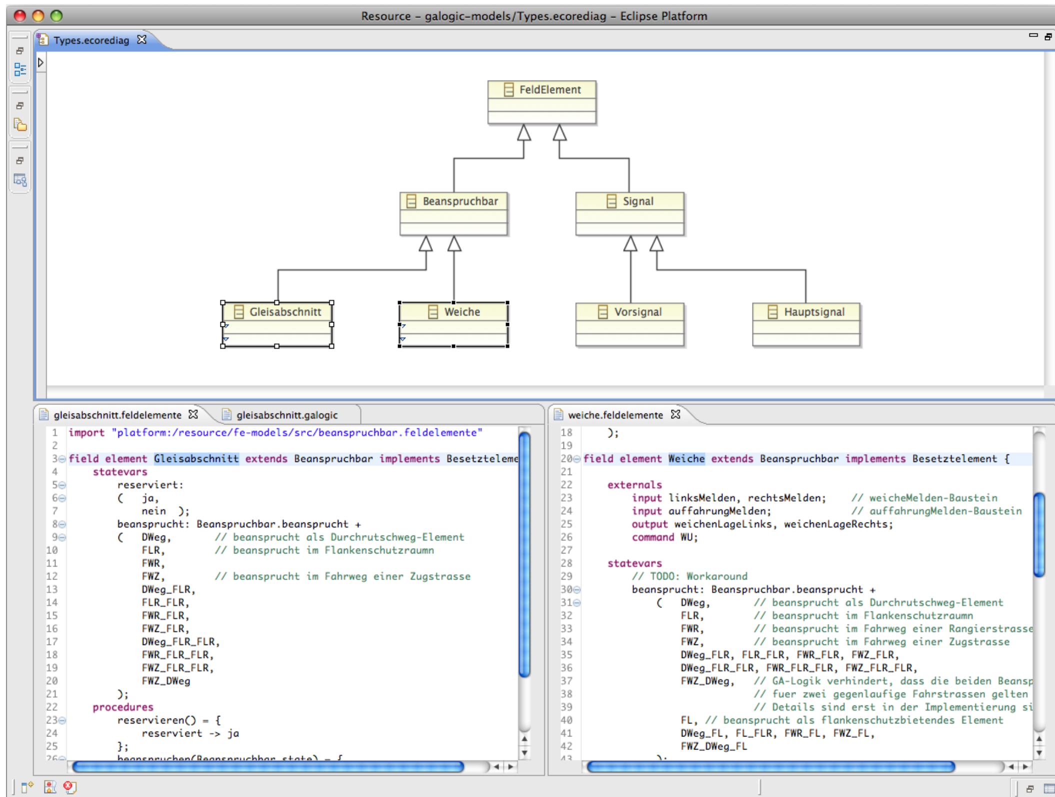


Figure 6.15: Overview on field elements:
Such a view may give overview representations and can be used to create, delete, and organize model elements, here *Feldelemente* models; representations may be composed of sets of models in a project space.

may serve as a kind of table of contents of distributed models, and may be used to navigate to the particular components. In the figure above this is indicated by choice of opened textual model views according to the selection of elements in the graphical view. Furthermore, this view might facilitate the creation of field elements. Depending on the form of persisting the elements of a modeling project, newly created parts may be saved as a dedicated model file in the workspace or added to a database.

Summary

The examples mentioned illustrate that differently shaped representations of models may increase the models utility and enrich state-of-the-practice domain-specific modeling tools. In contrast strictly linked models and representations, transient representations yield several benefits. Such representations may be extracted from arbitrary shaped content, even from a collection of model files in a project space as demonstrated by the last example. This, however, requires a notable amount of work to be spent in order to prepare the common technologies and frameworks to act in

the desired way. Since MENGES is a research project it is worth breaking this new ground.

Chapter 7

Conclusion

This thesis provides conceptional solutions concerning the problem of the integration of different forms of model representation, according to the problem description given in Section 1.2. Such representations may be of graphical or textual shape. The given solutions are of generic nature and may be applied to every domain specific modeling environment. However, some custom parts need to implemented, e. g., tailoring due to specifics in abstract syntaxes. The solutions are examined with respect to their effort to realize them, as well as their advantages and disadvantage in the practical use.

The major contribution of this thesis, is the concept of *transient-view-based model representations*. Such views are derived from chosen model content, and may be tailored to the purpose of the represented information. Such purposes may be

- the exploration of specifications,
- the comfortable model editing,
- the tracing of errors,
- the in-house documentation, or
- the documentation of system specifications for external stakeholders.

The transient view approach has been realized in KIELER. Within this experimental modeling environment, a user interface part dedicated to this purpose has been installed. Based on a graphical state chart language called SyncCharts, the textual concrete syntax called KITS has been designed, in order to represent SyncCharts models in a compact form. Besides obtaining textual representations of entire models, the modeler may choose model elements to be viewed in full detail.

In addition to the supply of such textual views a synchronization mechanism has been installed. It enables the feedback of model modifications that are performed in the textual view. This mechanisms leverages a model comparison utility, which previously has mostly been used for version control on models. Although there are some minor issues due to redundancies in the abstract syntax of the SynchCharts language, the overall result is very satisfying.

Due to this success, the transient view concept is to be employed within the industrial-oriented modeling environment that is developed by the MENGES team. Since this project is still in an early stage, alternative representations of the preliminary domain-specific languages have been proposed. With help of these representation advanced model handling features are studied.

Future Work

SyncCharts

Although the integration of SynchCharts and the textual view seems pretty mature, some work still needs to be done. This comprises the integration with the KiVi that has been developed by Müller [34] in parallel to the implementation parts this thesis is based on. The other major construction site is the EMF Compare framework. It turned out that the generic merge implementations provided by the framework are erroneous in certain situations. These mistakes have been fixed in an ad-hoc fashion. Since this is no solution for the long term, it seems worth to contribute these fixes to the framework. Furthermore, the issue concerning the short user interface freezes mentioned at the end of Chapter 5 should be investigated.

Apart from these purely technical issues, a possible extensions of the KITS view is the equipment of *undo/redo support* as it is common for usual textual editors, since this is missing in the current implementation of the KITS view. Instead, modifications that have been performed in this view and that have been transmitted to the graphical model view may only be reverted by means of the undo/redo feature of the graphical editor.

A further extension concerns the validation and quickfix support that is available in Xtext-based editors, but missing in the KITS view. This feature provides extension hooks for registering implementations of semantic model checks as well as related proposals for fixing errors indicated by such checks. This way Xtext-based DSL editors may provide assistance to modelers with respect to the adherence of certain constraints of the DSL that are not reflected its metamodel, and can be enhanced with corresponding proposals, which will fix particular constraint violations, as it is common practice in current programming language editors. The indication of exemplary errors and proposals of fixes in the KITS editor is shown in Figure 7.1.

MENGES

In order to realize the scenarios proposed in Section 6.4, basic questions need to be clarified.

1. Which graphical modeling framework shall be used?
   Apart from GMF, which is rather rigid and inflexible with respect to customiza-

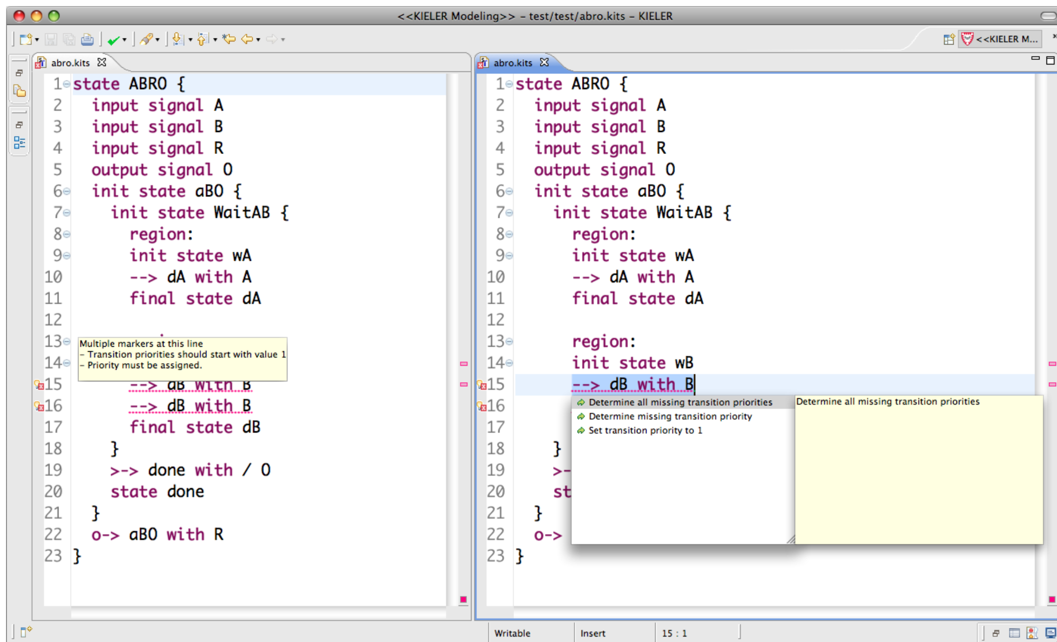


Figure 7.1: Validation and quickfix proposals in Xtext-based editors.

tions, a much more promising candidate is the newly introduced *Graphiti*[1] framework. This one is not (yet) equipped with a code generation facility, so custom parts must be implemented manually. On the other hand, this and other concepts allow a much more flexible creation and handling of graphical elements. Furthermore, in contrast to GMF, no isomorphic relation between elements of the abstract and concrete syntax of graphical representation is enforced. This even allows the composition of diagrams without maintaining any single element of the abstract syntax. An open question, however, is, whether Graphiti-based editors may be "remote controlled" as it is possible with GMF-based editors.

2. Can a graphical editor be attached to an Eclipse view?
   This question is crucial with respect to abolishing the file based generation of graphical representations. The overall goal is to achieve a user interface element that is able to *represent model content on demand* in a chosen shape, as it has been realized for different (even though similar) textual representations in KIELER (see Figure 5.16 on page 72).

3. Shall *a single* editor, equipped with different concrete syntaxes, be deployed, or rather multiple ones, each of them able to depict chosen elements?
   The answer of this question influences the capabilities of the graphical rep-

[1] http://www.eclipse.org/graphiti/

resentations in terms of displaying dependency relations between elements of models. Such dependency relations are, e. g., the use-def-relation of variables or the formerly mentioned relation between *GALogic*'s rule blocks and its state machines. An example of such a visualization is given in Figure 7.2. The decision on this question is important since graphical elements, here arrows, cannot be drawn over the bounds of the canvas of the graphical workbench parts.

4. What are adequate graphical syntaxes for such domain-specific content?
   Concrete graphical representations have to be developed, which are as rich as needed in terms of the model details to be represented. With respect to the proposals given in Section 6.2, it needs to be discussed whether the black bar in the rule block diagrams shall be approved, replaced be another element, or abolished entirely. Furthermore, a priority assignment should by added to rule block diagrams denoting the order of the examination of action guards explicitly, rather than expressing this by the arrangement in the diagram. The latter, which is currently employed, requires a specialized solution and thus limits the range of applicable layout algorithms. Furthermore, a graphical syntax for the DSL concerning the deployment issues must be designed.

After these question have been answered, concrete implementations concerning actual graphical viewers as well as the features presented in Section 6.4 can be realized. The KIELER View Management shall be employed in order to coordinate them. Since any details on these implementations are beyond the scope of this work, this thesis concludes with these remarks.

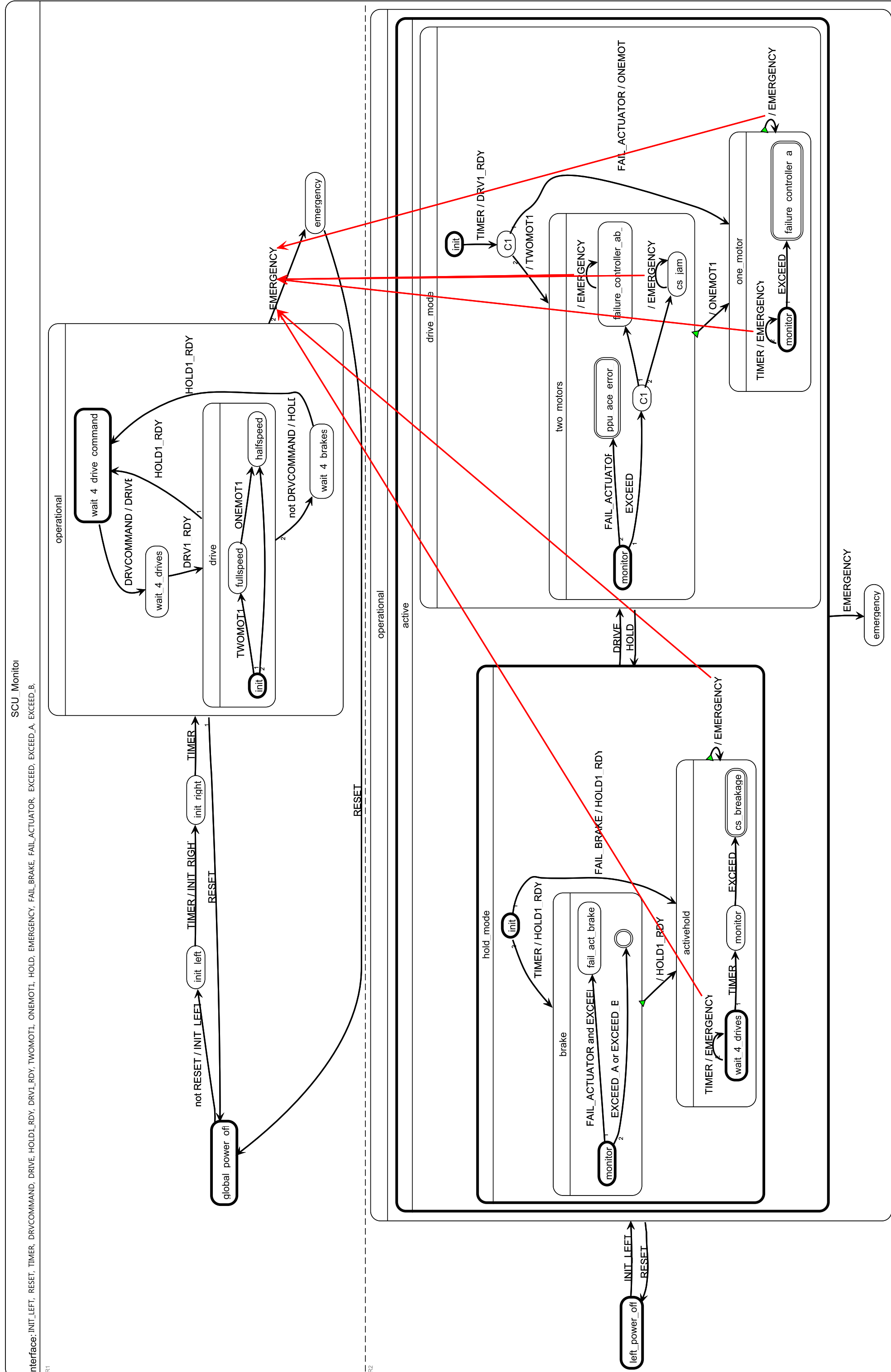


Figure 7.2: Graphical visualization of the emit-test-relation of signals, here restricted to the signals, triggering a chosen transition (Müller [34]).

Bibliography

[1] Charles André. SyncCharts: A visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR 96–56, I3S, Sophia-Antipolis, France, Rev. April 1996. 50, 51

[2] Charles André. Semantics of S.S.M (Safe State Machine). Technical report, Esterel Technologies, Sophia-Antipolis, France, April 2003. http://www.esterel-technologies.com. 51

[3] Francisco Pérez Andrés, Juan de Lara, and Esther Guerra. Domain Specific Languages with Graphical and Textual Views. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Proceedings of the 3rd International Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007)*, LNCS, Kassel, Germany, October 2007. Springer. ix, 19, 21

[4] Özgün Bayramoglu. The KIELER textual editing framework. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2009. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/oba-dt.pdf. 19, 43, 57

[5] Albert Benveniste and Gerard Berry. The synchronous approach to reactive and real-time systems. In *Proceedings of the IEEE*, pages 1270–1282, 1991. 51

[6] Grady Booch, Robert A. Maksimchuk, Michael Engle, and Jim Conallen. *Object-oriented Analysis and Design with Applications.* Benjamin-Cummings Publishing Co., Redwood City, 2nd edition, 1993. 2

[7] Grady Booch, Robert A. Maksimchuk, Michael Engle, and Jim Conallen. *Object-oriented Analysis and Design with Applications.* Benjamin-Cummings Publishing Co., Redwood City, 3rd edition, 2007. 2

[8] Tom DeMarco. *Structured Analysis and System Specification.* Yourdon Press, New York, 1978. 1

[9] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice Hall, 1998. 8

[10] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? June 2005. http://martinfowler.com/articles/languageWorkbench.html. 3, 6

[11] Hauke Fuhrmann. *On the Pragmatics of Graphical Modeling.* Dissertation, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, Kiel, 2011. ix, 23, 24, 25

[12] Hauke Fuhrmann, Miro Spönemann, Michael Matzen, and Reinhard von Hanxleden. Automatic layout and structure-based editing of UML diagrams. In *Proceedings of the 1st Workshop on Model Based Engineering for Embedded Systems Design (M-BED 2010)*, Dresden, March 2010. 84, 95

[13] Hauke Fuhrmann and Reinhard von Hanxleden. On the pragmatics of model-based design. In *Proceedings of the 15th Monterey Workshop on Foundations of Computer Software. Future Trends and Techniques for Development (2008)*, LNCS, Budapest, Hungary, September 2010. Springer. 46, 49

[14] Hauke Fuhrmann and Reinhard von Hanxleden. Taming graphical modeling. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010)*, LNCS, Oslo, Norway, October 2010. Springer. 8, 14, 49

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, Reading, MA, 1995. 39

[16] Sébastien Gérard, Cédric Dumoulin, Patrick Tessier, and Bran Selic. Papyrus: An UML2 Tool for Domain-Specific Language Modeling. In Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schätz, editors, *Proceedings of the 2007 International Dagstuhl conference on Model-based engineering of embedded real-time systems (MBEERTS 2007)*, LNCS. Springer, October 2010. 33

[17] Thomas Goldschmidt, Steffen Becker, and Axel Uhl. Classification of Concrete Textual Syntax Mapping Approaches. In Ina Schieferdecker and Alan Hartman, editors, *Proceedings of the 4th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2008)*, LNCS, Berlin, Germany, June 2008. Springer. 17

[18] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Text-based Modeling. In *Proceedings of the 4th International Workshop on Software Language Engineering (ATEM 2007)*, Nashville, USA, October 2007. 15

[19] Esther Guerra and Juan de Lara. Meta-Modelling and Graph Transformation for the Definition of Multi-View Visual Languages. In Fernando Ferri, editor,

Visual Languages for Interactive Computing: Definitions and Formalizations. IGI Global, 2007. http://astreo.ii.uam.es/~jlara/MultipleViews.pdf. ix, 20, 39, 40

[20] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987. 50

[21] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual Syntax for Models. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2009)*, LNCS, Enschede, The Netherlands, June 2009. Springer. ix, 16, 17, 18

[22] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. Closing the Gap between Modelling and Java. In Mark van den Brand, Dragan Gasevic, and Jeff Gray, editors, *Proceedings of the Second International Conference on Software Language Engineering (SLE 2009)*, LNCS, Denver, USA, October 2009. Springer. 16

[23] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992. 2

[24] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen, editors, *Proceedings of the 5th international conference on Generative programming and component engineering (GPCE 2006)*, Portland, USA, October 2006. ACM Press. ix, 15, 16

[25] Michael Kaufmann and Roland Wiese. Maintaining the mental map for circular drawings. In Stephen G. Kobourov and Michael T. Goodrich, editors, *Proceedings of the 10th International Symposium on Graph Drawing (GD 2002)*, LNCS, Irvine, USA, August 2002. Springer. 86

[26] Anneke Kleppe. Towards the Generation of a Text-Based IDE from a Language Metamodel. In David H. Akehurst, Régis Vogel, and Richard F. Paige, editors, *Proceedings of the 3rd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2007)*, LNCS, Haifa, Israel, June 2007. Springer. 11

[27] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In Jonathan Sprinkle, Jeff Gray, Matti Rossi, and Juha-Pekka Tolvanen, editors, *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM 2007)*, number TR-38 in Technical Reports, Montréal, Canada, October 2007. University of Jyväskylä, Finland. 16

[28] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007)*, LNCS, Nashville, USA, October 2007. Springer. 16

[29] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: Modular Development of Textual Domain Specific Languages. In Richard F. Paige and Bertrand Meyer, editors, *Proceedings of the 46th International Conference on Objects, Components, Models and Patterns (TOOLS 2008)*, LNCS, Zurich, Switzerland, July 2008. Springer. 16

[30] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12:231–260, 2003. 79

[31] Michael Matzen. A generic framework for structure-based editing of graphical models in Eclipse. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2010. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mim-dt.pdf. 8, 31, 33

[32] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955. 50

[33] Andreas Mülder, Holger Schill, and Lothar Wendehals. Modellvergleich mit EMF Compare, Part 1+2. *Eclipse Magazin*, 4+5, 2009. 29

[34] Martin Müller. View management for graphical models. Master thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2010. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mmu-mt.pdf. x, 100, 103

[35] Object Management Group. *Human-Usable Textual Notation Specification*, August 2004. http://www.omg.org/spec/HUTN/. 16

[36] Object Management Group. *Specification of the System Modeling Language™*(SysML®), June 2010. http://www.omg.org/spec/SysML/1.2/. 3

[37] Object Management Group. *Specification of the Unified Modeling Language™*(UML®), May 2010. http://www.omg.org/spec/UML/2.3/. 2

[38] Meilir Page-Jones. *The Practical Guide to Structured Systems Design*. Yourdon Press, New York, 1980. 1

[39] Eric S. Raymond. *The Art of UNIX Programming*. Addison-Wesley, 2003. http://www.catb.org/esr/writings/taoup/html/index.html. 4

[40] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona Polack. Constructing Models with the Human-Usable Textual Notation. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Proceedings of the ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008)*, LNCS, Toulouse, France, October 2008. Springer. 16

[41] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Englewood Cliffs, New York, 1990. 2

[42] Markus Scheidgen. Integrating Content Assist into Textual Modelling Editors. In Thomas Kühne, Wolfgang Reisig, and Friedrich Steimann, editors, *Modellierung 2008*, LNI, Berlin, Germany, March 2008. Gesellschaft für Informatik e.V. (GI). 19

[43] Markus Scheidgen. Textual Modelling Embedded into Graphical Modelling. In Ina Schieferdecker and Alan Hartman, editors, *Proceedings of the 4th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2008)*, LNCS, Berlin, Germany, June 2008. Springer. 8, 19, 36, 93

[44] Arne Schipper, Hauke Fuhrmann, and Reinhard von Hanxleden. Visual comparison of graphical models. In *Proceedings of the Fourth IEEE International Workshop UML and AADL, held in conjunction with the 14th International International Conference on Engineering of Complex Computer Systems (ICECCS 2009)*, Potsdam, Germany, June 2009. 29

[45] Matthias Schmeling. ThinKCharts—the thin KIELER SyncCharts editor. Student research project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, September 2009. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/schm-st.pdf. 50

[46] Janet M. Six and Ioannis G. Tollis. A framework for circular drawings of networks. In Jan Kratochvíl, editor, *Proceedings of the 7th International Symposium on Graph Drawing (GD 1999)*, LNCS, Stirín Castle, Czech Republic, September 1999. Springer. 86

[47] Miro Spönemann. On the automatic layout of data flow diagrams. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2009. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/msp-dt.pdf. 8

[48] Thomas Stahl, Markus Völter, Sven Efftinge, and Arno Haase. *Modellgetriebene Softwareentwicklung*. dpunkt.verlag, Heidelberg, 1st edition, 2005. 3, 5, 6

[49] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF - Eclipse Modeling Framework*. Addison Wesley, 2009. 27

[50] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, February 1981. 86

[51] Sun Microsystems. *How to Write Doc Comments for the Javadoc Tool.* http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html. 64

[52] Antoine Toulmé. Presentation of EMF Compare Utility. In *Eclipse Modeling Symposium at Eclipse Summit Europe (ESE 2006)*, Esslingen, Germany, October 2006. 28

[53] Markus Völter. Embedded Software Development with Projectional Language Workbenches. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010)*, LNCS, Oslo, Norway, October 2010. Springer. 17

[54] Reinhard von Hanxleden. SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency. In *Proceedings of the International Conference on Embedded Software (EMSOFT'09)*, Grenoble, France, October 2009. 3

[55] Mirko Wischer. Textuelle Darstellung und strukturbasiertes Editieren von Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, February 2006. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/miwi-dt.pdf. ix, 18, 19

[56] Edward Yourdon. *Modern Structured Analysis.* Yourdon Press, New York, 1989. 1

[57] Edward Yourdon and Larry L. Constantine. *Structured Design. Fundamentals of a Discipline of Computer Program and Systems Design.* Yourdon Press, New York, 1975. 1

Appendix A

Grammars of KITS

A.1 Grammar of KITS

```
grammar de.cau.cs.kieler.synccharts.text.kits.Kits with de.cau.cs.kieler.synccharts.text.actions.Actions

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
import "platform:/resource/de.cau.cs.kieler.core.annotations/model/annotations.ecore" as annotations
import "platform:/resource/de.cau.cs.kieler.core.expressions/model/expressions.ecore" as expressions
import "platform:/resource/de.cau.cs.kieler.synccharts/model/synccharts.ecore" as synccharts

RootRegion returns synccharts::Region:
    (annotations+=ImportAnnotation)*
    (
      (annotations += StringAnnotation)*
      'region' (id=ID)? (label=STRING)? ':'
      (variables+=Variable|signals+=Signal)*
      (bodyText+=TextualCode)*
     )?
    (states+=State)*;

// -------------------------------------------------------------------------------------------------

SingleRegion returns synccharts::Region:
    (
      (annotations += StringAnnotation)*
      'region' (id=ID)? (label=STRING)? ':'
      (variables+=Variable|signals+=Signal)*
      (bodyText+=TextualCode)*
     )?
    (states+=State)*;

// -------------------------------------------------------------------------------------------------

Region returns synccharts::Region:
    (annotations += StringAnnotation)*
    'region' (id=ID)? (label=STRING)? ':'
    (variables+=Variable|signals+=Signal)*
    (bodyText+=TextualCode)*
    (states+=State)+;

// -------------------------------------------------------------------------------------------------

State returns synccharts::State:
    (annotations += StringAnnotation)*
    (
      ((isInitial?='init') (isFinal?='final')?)
     |
      ((isFinal?='final') (isInitial?='init')?)
     )?

    (type=StateType)? ('state') (id=ID) (label=STRING)?

    (
      ('@' bodyReference = [synccharts::State|ID]
        ('[' renamings+= Substitution (',' renamings+= Substitution)* ']')?
       )
     |
      ('{'
          (
            (signals+=Signal                |
             variables+=Variable            |
             'onentry' entryActions+=Action |
```

```
60              'oninner' innerActions+=Action |
                'onexit'  exitActions+=Action  |
                'suspension' suspensionTrigger=Action)*
               (bodyText += TextualCode)*
               (
65              (regions+=SingleRegion)(regions+=Region)*
                )?
               )
             '}')
      )?
70    (outgoingTransitions+=Transition)*;

  // ----------------------------------------------------------------------------------------------------

  Transition returns syncharts::Transition:
75    (annotations += StringAnnotation)*
      type=TransitionType (priority=INT)? targetState=[syncharts::State|ID]
      ( 'with'
       (
         ( (isImmediate?='#')? ((delay=INT)? ('tick' | trigger=BooleanExpression))? ('/' effects+=Effect (','effects+=Effect)*)? )
80      |
         ( label=STRING )
        )
       )?
      (isHistory?='history')?;
85
  // ----------------------------------------------------------------------------------------------------

  Signal returns expressions::Signal:
      (annotations += StringAnnotation)*
90    (isInput?='input')?
      (isOutput?='output')?
      'signal' name=ID (':=' initialValue=AnyValue)?
      ((':' (type = ValueType | hostType = STRING))
        |
95     (':' 'combine' (type = ValueType | hostType = STRING) 'with'
          (combineOperator = CombineOperator | hostCombineOperator = STRING))
      )?;

  // ----------------------------------------------------------------------------------------------------
100
  Variable returns expressions::Variable:
      (annotations += StringAnnotation)*
      'var' name=ID (':=' initialValue=AnyValue)? ':' (type = ValueType | hostType = STRING);

105 // ----------------------------------------------------------------------------------------------------

  Substitution returns syncharts::Substitution:
      actual = ID '/' formal = ID;

110 // ----------------------------------------------------------------------------------------------------

  enum StateType returns syncharts::StateType:
      NORMAL = 'normal' | CONDITIONAL = 'conditional' | REFERENCE = 'reference' | TEXTUAL = 'textual';

115 enum TransitionType returns syncharts::TransitionType:
      WEAKABORT = '-->' | STRONGABORT = 'o->' | NORMALTERMINATION = '>->';
```

Listing A.1: Grammar of KITS.

A.2 Grammar of actions

```
grammar de.cau.cs.kieler.synccharts.text.actions.Actions with de.cau.cs.kieler.core.expressions.Expressions

//// we do not need to generate the transitionlabel ecore model, because we
//// only reuse classes of the official synccharts metamodel
//generate transitionlabel "http://kieler.cs.cau.de/synccharts/actionlabel"

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
import "platform:/resource/de.cau.cs.kieler.synccharts/model/synccharts.ecore" as synccharts
import "platform:/resource/de.cau.cs.kieler.core.expressions/model/expressions.ecore" as expressions

// haf: We need a Rule for transitions in order to serialize them
// here we only want to have the features isImmediate, delay, trigger and effects
// the features type, targetState, priority, isHistory are ignored and set as transient
// you need to override the rule to support transitions properly

Transition returns synccharts::Transition:
    {synccharts::Transition}
    (isImmediate?='#')? ((delay=INT)? ('tick' | trigger=BooleanExpression))? ("/" effects+=Effect ((',') effects+=Effect)*)?;


// chsch: The action rule is used in Kits.xtext for entry-, inner-, exitActions, suspensionTrigger
Action returns synccharts::Action:
    {synccharts::Action}
    (isImmediate?='#')? ((delay=INT)? ('tick' | trigger=BooleanExpression))? ("/" effects+=Effect ((',') effects+=Effect)*)?;


Effect returns synccharts::Effect:
    Emission | Assignment | TextEffect;


Emission returns synccharts::Emission:
    signal=[expressions::Signal] ( "(" newValue=Expression ")" )?;


Assignment returns synccharts::Assignment:
    variable=[expressions::Variable] ":=" expression = Expression;


TextEffect returns synccharts::TextEffect:
    code=HOSTCODE ("(" type=ID ")")?;


TextualCode returns expressions::TextualCode:
    'textual' 'code' ( '(' type = ID ')')? ':' code = STRING;


//enum DivOperator returns expressions::OperatorType:
//  DIV=":";
```

Listing A.2: Actions grammar used by KITS.

A.3 Grammar of expressions

```
grammar de.cau.cs.kieler.core.expressions.Expressions with de.cau.cs.kieler.core.annotations.Annotations

import "platform:/resource/de.cau.cs.kieler.core.expressions/model/expressions.ecore"
import "platform:/resource/de.cau.cs.kieler.core.annotations/model/annotations.ecore" as annotations

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

// -------------------------
//
//   EXPRESSIONS
//
// -------------------------

Expression returns Expression:
    BooleanExpression
    | ValuedExpression;


// Example: not D and C or ?E = 42 or not (A and (B or C))
BooleanExpression returns Expression:
    OrExpression;


// Example: A or B, A and B and C, C and B or D and not E, A and B and C
OrExpression returns Expression:
    AndExpression ({OperatorExpression.subExpressions+=current} operator=OrOperator subExpressions+=AndExpression)*;


// Example: A and B, not C and 42 <= ?D
AndExpression returns Expression:
    CompareOperation ({OperatorExpression.subExpressions+=current} operator=AndOperator subExpressions+=CompareOperation)*;


// Example: 42 <= ?A
CompareOperation returns Expression:
    NotOrValuedExpression ({OperatorExpression.subExpressions+=current} operator=CompareOperator subExpressions+=NotOrValuedExpression)
    | NotExpression;


// order IS IMPORTANT
NotOrValuedExpression returns Expression:
    ValuedExpression
    | NotExpression;


// everything that evaluates to a primitive number value
ValuedExpression returns Expression:
    AddExpression;


// Example: 1 + 2
AddExpression returns Expression:
    SubExpression ({OperatorExpression.subExpressions+=current} operator=AddOperator subExpressions+=SubExpression)*;


// Example: varA - ?B
SubExpression returns Expression:
    MultExpression ({OperatorExpression.subExpressions+=current} operator=SubOperator subExpressions+=MultExpression)*;


// Example: 2 * 4
MultExpression returns Expression:
    DivExpression ({OperatorExpression.subExpressions+=current} operator=MultOperator subExpressions+=DivExpression)*;


// Example: 2 / 4
DivExpression returns Expression:
    ModExpression ({OperatorExpression.subExpressions+=current} operator=DivOperator subExpressions+=ModExpression)*;


// Example: varA mod ?B
ModExpression returns Expression:
    NegExpression ({OperatorExpression.subExpressions+=current} operator=ModOperator subExpressions+=NegExpression)?;


NegExpression returns Expression:
    {OperatorExpression} operator=SubOperator subExpressions+=(NegExpression)
    | AtomicValuedExpression;

// Example: not A, not false, not (A or B)
```

```
// at the latter we need the parans to indicate the right binding
NotExpression returns Expression:
    {OperatorExpression} operator=NotOperator subExpressions+=(NotExpression)
    | AtomicExpression;


AtomicExpression returns Expression:
    BooleanValue
    | ValuedObjectTestExpression
    | '(' BooleanExpression ')'
    | TextExpression;


AtomicValuedExpression returns Expression:
    IntValue
    | FloatValue
//    | '(' DivExpression ')'
    | '(' ValuedExpression ')'
    | AtomicExpression;

// UPDATE (chsch): div expression has been moved above as conflict with transition delay was abolished (eg. 3 / A, whereas 3 is not a guard
      but a delay)
// Example: (2 / 4)
// note: division always has to have parantheses because the '/' sign is also used for trigger/effect delimiter
//DivExpression returns Expression:
//  AtomicValuedExpression {OperatorExpression.subExpressions+=current} operator=DivOperator subExpressions+=AtomicValuedExpression;


// Example: pre(pre(?A)), pre(pre(A)), ?A, A varX
ValuedObjectTestExpression returns Expression:
    {OperatorExpression} operator=PreOperator '('subExpressions+=ValuedObjectTestExpression')'
    | {OperatorExpression} operator=ValueTestOperator subExpressions+=ValuedObjectReference
    | ValuedObjectReference;


// Example: A, varB
ValuedObjectReference returns ValuedObjectReference:
    valuedObject=[ValuedObject|ID];


// Example: 'printf(...)'(C)
TextExpression returns TextExpression:
    code=HOSTCODE ("(" type=ID ")")?;


IntValue returns IntValue:
    value=INT;


FloatValue returns FloatValue:
    value=Float;


BooleanValue returns BooleanValue:
    value=Boolean;


// data type rule allowing any kind of value to be accepted,
// e.g. as initialValues of valuedObjects
// used in Kits.xtext
AnyValue returns ecore::EString:
    Boolean | INT |Float | ID | STRING;


enum CompareOperator returns OperatorType:
    EQ="=" | LT="<" | LEQ="<=" | GT=">" | GEQ=">=" | NE="<>";

enum PreOperator returns OperatorType:
    PRE="pre";

enum OrOperator returns OperatorType:
    OR="or";

enum AndOperator returns OperatorType:
    AND="and";

enum NotOperator returns OperatorType:
    NOT="not";

enum AddOperator returns OperatorType:
    ADD="+";

enum SubOperator returns OperatorType:
    SUB="-";
```

```
enum MultOperator returns OperatorType:
    MULT="*";

enum ModOperator returns OperatorType:
    MOD="mod";

enum DivOperator returns OperatorType:
    DIV="/";

enum ValueTestOperator returns OperatorType:
    VAL="?";


/*
   the following declarations are re-used in Actions.xtext, Interface.xtext, Kits.xtext
*/
enum ValueType returns ValueType:
    PURE="pure" | BOOL="boolean" | UNSIGNED="unsigned" |
    INT="integer" | FLOAT="float" | HOST="host";


enum CombineOperator returns CombineOperator:
    NONE="none" | ADD="+" | MULT="*" | MAX="max" |
    MIN="min" | OR="or" | AND="and" | HOST="host";


// make sure the Float rule does not shadow the INT rule
terminal Float returns ecore::EFloatObject :
    ’-’?(’0’..’9’)+ ("."(’0’..’9’)*) (("e"|"E")("+"|"-")?(’0’..’9’)+)? "f"? |
    ’-’?(’0’..’9’)+"f";


// redefine INT terminal to allow negative numbers
terminal INT returns ecore::EInt:
    ’-’?(’0’..’9’)+;


// introduce boolean values
terminal Boolean returns ecore::EBooleanObject :
    "true" | "false";


// custom terminal rule allowing to save transition label string as they are
terminal STRING returns ecore::EString:
    ’"’ ( ’\\’ (’b’|’t’|’n’|’f’|’r’|’"’|"’"|’\\’) | !(’\\’|’"’) )* ’"’;


// custom terminal rule allowing to save transition label string as they are
terminal HOSTCODE returns ecore::EString:
    "’" ( ’\\’ (’b’|’t’|’n’|’f’|’r’|’"’|"’"|’\\’) | !(’\\’|"’") )* "’";
```

Listing A.3: Expressions grammar used by KITS.

A.4 Grammar of annotations

```
grammar de.cau.cs.kieler.core.annotations.Annotations with org.eclipse.xtext.common.Terminals

import "platform:/resource/de.cau.cs.kieler.core.annotations/model/annotations.ecore"

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

// -------------------------
//
//   ANNOTATIONS
//
// -------------------------

// introduction of parsing rules for annotations
StringAnnotation returns Annotation:
    CommentAnnotation
    | KeyValueAnnotation;


// e.g.: /** semantic comment */
CommentAnnotation returns StringAnnotation:
    value=COMMENT_ANNOTATION;


// e.g.: @layouter dot; @layoutOptions "margin 5, dir top-down";
KeyValueAnnotation returns StringAnnotation:
    '@' name=ID (value=EString)?;


// needed for importing other resources
ImportAnnotation returns ImportAnnotation:
    'import' importURI=STRING;


// allow strings without quotes as they don'c contain spaces
EString returns ecore::EString:
    STRING | ID;


// custom terminal rule introducing semantic comments
terminal COMMENT_ANNOTATION returns ecore::EString:
    '/**' -> '*/';


// modified version of Terminals.ML_COMMENT as
// COMMENT_ANNOTATION is not recognized correctly with original one
terminal ML_COMMENT returns ecore::EString:
    '/*'!'*' -> '*/';
```

Listing A.4: Annotations grammar used by KITS.