On the Pragmatics of Graphical Modeling

Dipl.-Inf. Hauke Axel Ludwig Fuhrmann

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2010

Kiel Computer Science Series (KCSS) 2011/1 v1.0 dated 2011-11-24

Electronic version, updates, errata available via https://www.informatik.uni-kiel.de/kcss

Published by the Department of Computer Science at Christian-Albrechts-Universität zu Kiel

Real-Time Systems and Embedded Systems Research Group

Please cite as:

▻ Hauke A. L. Fuhrmann. *On the Pragmatics of Graphical Modeling.* Number 2011/1 in Kiel Computer Science Series. Department of Computer Science, 2011. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel.

```
@Book{Fuhrmann11,
  author = {Hauke A. L. Fuhrmann},
  title = {On the Pragmatics of Graphical Modeling},
  publisher = {Department of Computer Science},
  year = 2011,
  month = may,
  number = {2011-1},
  isbn = {978-3-84480-084-5},
  series = {Kiel Computer Science Series},
  note = {Dissertation, Faculty of Engineering,
  Christian-Albrechts-Universit\"at zu Kiel}}
```


Herstellung und Verlag: Books on Demand GmbH, Norderstedt

About this Series

The Kiel Computer Science Series (KCSS) covers dissertations, habilitation theses, lecture notes, textbooks, surveys, collections, handbooks, etc. written at the Department of Computer Science at the Christian-Albrechts-Universität zu Kiel. It was initiated in 2011 to support authors in the dissemination of their work in electronic and printed form, without restricting their rights to their work. The series provides a unified appearance and aims at high-quality typography. The KCSS is an open access series; all series titles are electronically available free of charge at the department's website. In addition, authors are encouraged to make printed copies available at a reasonable price, typically with a print-on-demand service.

Please visit `http://www.informatik.uni-kiel.de/kcss` for more information, for instructions how to publish in the KCSS, and for access to all existing publications.

1. Gutachter:	Prof. Dr. Reinhard von Hanxleden Institut für Informatik Christian-Albrechts-Universität zu Kiel
2. Gutachter:	Prof. Dr. Wilhelm Hasselbring Institut für Informatik Christian-Albrechts-Universität zu Kiel
3. Gutachter:	Prof. Dr. Edward A. Lee Electrical Engineering and Computer Sciences Department University of California at Berkeley

Datum der mündlichen Prüfung: 5. Mai 2011

Zusammenfassung

Grafische Modellierung ist attraktiv, da sie mit Hilfe von Diagrammen Modelle und deren Eigenschaften präsentiert. Diagramme eröffnen die zweite Dimension im Gegensatz zum bisher meist eingesetzten Text. Dadurch können natürliche Eigenschaften von Diagrammen, z.B. grafische Enthaltensein-Beziehungen, helfen, entsprechende Modellbeziehungen zu visualisieren. Diagramme können diverse solcher Sachverhalte *direkt* darstellen, während Text einschließlich Formeln in der Regel *indirekt* Definitionen und Erklärungen bedarf. Richtig eingesetzt können Modelle somit leichter vom Menschen verstanden werden und als gemeinsame Austauschgrundlage für Experten unterschiedlicher Fachrichtungen dienen.

Die zweite Dimension bringt allerdings eine neue Komplexität mit sich. Durch diese verbringen die Benutzer viel Zeit mit Zeichenwerkzeugen und arbeiten mit archaischen Interaktionsmechanismen, bei denen sie mit hohem Zeitaufwand grafische Elemente händisch auf der Zeichenfläche positionieren. Zudem ist es schwierig, sich in komplexen Modellen zurechtzufinden, da die Diagramme entweder sehr groß oder stark verschachtelt sind und mit den üblichen Navigations- und Visualisierungstechniken schwierig zu handhaben sind.

In der vorliegenden Arbeit präsentiere ich ein integriertes Konzept zur besseren Handhabung von grafischen Modellen. Es beinhaltet technologische Bausteine, mit denen sich die Benutzerinteraktionen auf das Wesentliche des Modellierungsprozesses reduzieren lassen: der semantische Inhalt der Modelle selbst.

Wichtigstes Werkzeug ist automatisiertes Layout von Diagrammen. Es wird konsequent eingesetzt um dem Benutzer die manuelle Positionierung abzunehmen, und erlaubt es, mehrere Diagramm-Sichten auf ein und das selbe Modell schnell und einfach zu generieren. Dadurch können dynamisch und interaktiv die Modellsichten verändert werden. Somit kann der Benutzer genau auf die Modellelemente gelenkt werden, die für ihn

momentan besonders wichtig sind. Kern dieses Ansatzes ist eine *Sichtenverwaltung*, die nach konfigurierbaren Bedingungen die aktuelle Sicht auf ein Diagramm interaktiv verändern kann.

Ich präsentiere eine Reihe von Anwendungsfällen im Bereich des modellbasierten Entwurfs, die von diesem Vorgehen profitieren. Dies beinhaltet strukturbasiertes Editieren und dynamische Fokus & Kontext Sichten während einer Simulation von Modellen. Zur Validierung und um die Verbreitung zu fördern wurden einige Beispieleditoren entwickelt und mit diesen Fähigkeiten ausgestattet. Sie sind im Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) umgesetzt, einer Eclipse-basierten Modellierungsumgebung, welche als offener Quelltext verfügbar ist.

Abstract

Graphical modeling employs diagrams to present models and their properties. These graphical depictions add the second dimension to one-dimensional text. Thus intrinsic properties of the diagrams, such as graphical spatial inclusion, can be used to visualize corresponding model relationships in a natural way, such as containment relations. Therefore graphical models are usually more *direct* while text—including formulas—tends to have more *indirect* features, meaning the used symbols require prior definition and explanation. If used correctly, graphical models can be easier understood by humans than textual ones. They can be used as a common ground for information exchange and discussion for different domain experts.

However, the second dimension introduces new complexity. It causes users to waste a lot of time with drawing tools using archaic interaction mechanisms where they have to manually place graphical items on the canvas. Creating and maintaining of diagrams is very effort prone. Additionally it is difficult to navigate in complex models. They are either very big or have a complex nesting structure and are thus difficult to handle with the usual navigation and visualization techniques.

In this thesis I present an integrated concept on handling graphical models. It comprises technological stepping-stones that each raise the level of abstraction of the user interaction with graphical models in order to concentrate on the semantic models.

A key enabler for improved designer productivity is the automatic layout of diagrams. It gets employed consistently in order to free the user from the burden of manual placing and routing. Additionally it allows to synthesize multiple graphical views onto the same model easily and quickly. Thus views can be dynamically changed to the user's needs. The user can be focused automatically to the model elements that are currently most important. The core of this approach is a *view management* that uses

configurable conditions to synthesize new model views interactively.

I present several use cases in model-driven engineering that benefit from this approach. This includes structure-based editing and dynamic focus & context views during simulation runs. For validation and to foster dissemination, several example editors with such capabilities have been developed in the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER), an Eclipse-based modeling environment available as open source.

Preface

by Prof. Dr. Reinhard von Hanxleden

Model-driven engineering (MDE) is by now an established process for the design of complex embedded systems. Starting with an abstract model of the system under development (SUD), which can also contain the environment of the SUD to enable closed-loop simulations, the model gets refined to a concrete implementation. Main motivations for model-based design are increased productivity, because developers can concentrate on the original abstract functionality, and reduced error-proneness, because implementations can be mainly synthesized, partly even by certified code generators. Often graphical/visual formalisms are employed in model-based design, especially on the higher abstraction levels that are more relevant to the developer.

The dissertation of Dr. Fuhrmann relates to model-based design, and covers practical aspects such as creation, modification, analysis and simulation of—primary graphical—system models. This topic is highly relevant as especially in complex systems, which are predestinated for model-driven engineering, the practical manageability of the system models is very effort-prone with the paradigms established today. The creation of models usually requires a time-consuming manual layout and is performed at a very elementary level of nodes and edges. The navigation in models quickly results in a confusing number of windows on the screen. This contradicts the original MDE goal of increased developer productivity.

The topic examined by Dr. Fuhrmann has been largely neglected in computer science so far, but touches a multitude of established research fields. Next to model-based engineering, which is the subject matter of a large community and also established in (not necessarily embedded) software engineering, this affects human-machine-interaction, graph layout theory and linguistics. The central notion in the work of Dr. Fuhrmann is *pragmatics*, also known from linguistics. There the pragmatics together

with syntax and semantics forms the *semiotics* of languages. Computer science concentrates in previous research mainly onto semantics and syntax of modeling, cf. the UML (Unified Modeling Language). The pragmatics, defined in Chapter 3 of the dissertation (also referencing an earlier conjoint work of Dr. Fuhrmann) as "all practical aspects of handling a model in its design process," is traditionally considered only marginally. Accordingly, little progress has been done so far. This field is currently mainly left to (commercial) tool vendors, who present different ad-hoc approaches for selective improvements of pragmatics (e.g. alignment tools for assisting manual layout), see the appendix of the dissertation.

The fundamental scientific contribution of the dissertation of Dr. Fuhrmann is a systematic introduction of the pragmatics into model-driven engineering. The foundation is a taxonomy derived from the *model-view-controller* paradigm, which is well-established in software engineering and comprises the editing (model), representation (view) and the interpretation (controller) of models. Dr. Fuhrmann develops within this taxonomy a set of approaches towards how pragmatics can systematically assist different aspects of model-driven engineering. The dissertation illustrates the potentials and problems emerging from the strict approach of separating view and model, e.g. considering automatic layout, filtering and structure-based editing. The work also discusses a set of practical solution approaches, which are prototypically implemented and validated with the Kiel Integrated Environment for Layout Eclipse RichClient (KIELER). The dissertation documents the experiences gained during development and concrete usage of KIELER and concludes with a critical overall assessment, which also provides help with the practical appliance and adoption of the presented results and indicates further research topics.

This work has strong engineering aspects and merges a set of different disciplines of computer science. Incorporating a multitude of research results developed by these disciplines, Dr. Fuhrmann succeeded in developing a novel and independent design approach, the one of pragmatics-aware modeling. This work presents a broad spectrum of novel modeling approaches, however, it does not end with individual results. The MVC-based taxonomy of pragmatics developed here forms a coherent frame for the presented approaches, which also sets the stage for further research oppor-

tunities.

Finally, as—now former—advisor of Dr. Fuhrmann I want to take this opportunity not only to highlight his significant scientific contributions, but also to compliment his excellent management and leadership skills in building up the KIELER project, which under his supervision ramped up from 0 to 35 committers. He was a very valued member of the RTSYS group, and with his can-do attitude a big motivator also for the rest of the team. He was and still is always willing to go the extra mile to make things work. So it is only fitting that, as a last contribution related to his dissertation, he played a key role in getting the Kiel Computer Science Series started, despite the high demands now placed on him from his new position in industry. Thanks for everything, Hauke.

Reinhard von Hanxleden
Kiel, November 2011

Contents

List of Figures

List of Acronyms

API	Application Programming Interface
ASCET	Advanced Simulation and Control Engineering Tool
ATL	Atlas Transformation Language
CASE	Computer-Aided Software Engineering
CHESS	Center for Hybrid and Embedded Software Systems
CS	Computer Science
CT	Continuous Time
CVS	Concurrent Version System
DE	Discrete Events
DND	Drag-and-Drop
DSL	Domain Specific Language
EMF	Eclipse Modeling Framework
EMOF	Essential MOF
EPL	Eclipse Public License
FSM	Finite-State-Machines
GEF	Graphical Editing Framework
GEMS	Generic Eclipse Modeling System
GME	Generic Modeling Environment
GMF	Graphical Modeling Framework

GPL	General Public License
GUI	Graphical User Interface
HMI	Human-Machine Interface
HTML	Hyper-Text Markup Language
IDE	Integrated Design Environment
JNI	Java Native Interface
JSON	Java Script Object Notation
JVM	Java Virtual Machine
KAOM	KIELER Actor-Oriented Modeling
KEG	KIELER Editor for Graphs
KEV	KIELER Environment Visualization
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KIEL	Kiel Integrated Environment for Layout
KIEM	KIELER Execution Manager
KIML	KIELER Infrastructure for Meta Layout
KiTE	KIELER Textual Editing
KiViK	KIELER Visual Kompare
KiVi	KIELER View Management
KIePto	KIELER leveraging Ptolemy Semantics
KLoDD	KIELER Layout of Data Flow Diagrams
KSBasE	KIELER Structure-Based Editing
M2M	Model-to-Model

M2T	Model-to-Text
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
MDSD	Model Driven Software Development
MDT	Model Development Tools
ME	Micro Edition
MIC	Model-Integrated Computing
MoC	Model of Computation
MOF	Meta Object Facility
MoML	Model Markup Language
MVC	Model-View-Controller
NP	Nondeterministic Polynomial
OGDF	Open Graph Drawing Framework
OMG	Object Management Group
OS	Operating System
PN	Process Networks
QVT	Query/View/Transformations
RCP	Rich Client Platform
RTSYS	Real-Time and Embedded Systems
S	Synchronous
SBasE	Structure-Based Editing
SC	Synchronous C

SCADE	Safety Critical Application Development Environment
SDF	Synchronous Data Flow
SR	Synchronous Reactive
SUD	System under Development
SVG	Scalable Vector Graphics
SWT	Standard Widget Toolkit
TCS	Textual Concrete Syntax
UI	User Interface
UML	Unified Modeling Language
VCS	Version Control System
VLSI	Very-Large-Scale Integration
VMS	View Management Scheme
W3C	World Wide Web Consortium
WYSIWYG	What-You-See-Is-What-You-Get
XML	Extensible Markup Language
XSD	XML Schema Definition

Chapter 1

Introduction

The main task in software programming is to command the computer to do the right thing. The programming mechanics of computers has undergone quite an evolution: From manually stamping programs on punch cards over type writers editing non-reversible text to the main method still used today—text editor and keyboard. While different IDEs might offer various support levels for large software artifacts, the basic mechanics of writing or changing a line of code is rather standard and efficient. Hence, editing text has been established for many decades.

The introduction of graphical models has added the second dimension to one-dimensional text. However, this new freedom comes at a heavy price: We are back to the early times of mechanical typewriters with rather archaic user interactions. Graphical layout has to be manually defined by placing and routing of nodes and edges. Deleting graphical objects, like using white-out on a typewriter, creates new white-space that might not be large enough to insert new expressions, i. e., new graphical constructs. Manually creating more space in a complex diagram is like using scissors and glue. In fact, in large, industrial projects it is not uncommon that highly-paid engineers use scissors and glue to create large hand-crafted posters from print-outs to help navigate through complex models. One estimate from industrial users puts the time spent with unproductive editing/formatting activities at about 30% of overall developing time[1].

Graphical views on models are manually defined and hence static like a type-written piece of paper. Creating multiple different views, e. g., for different levels of abstraction, onto the same model requires much manual

[1] L. K. Klauske (Daimler Center for Automotive IT Innovations), personal communication, Oct. 2009.

editing work. Often one ends up working with one single abstraction level or changing syntax from graphical to e. g., structural text or table views to get more detailed or more abstract representations. Although abstraction might play an important role for Model-Driven Engineering (MDE) [Kra07], so far, graphical aspects of models certainly do not. Instead of unfolding their potential as a vivid means of communication they usually are reduced to no more than syntactic sugar. When trying to communicate with the computer through graphical models, the computer will not answer in the same language. For example, model transformations typically loose the graphical information and result in a model without a graphical view, which is like typing in text and getting a punch card as answer. If one believes that a diagram communicates the meaning of a model better than another representation, and if one wants this to be widely accepted by domain users that are not necessarily computer scientists, then one has to teach computers to truly master this language.

This thesis presents an approach to bridge the gap between Model-Driven Engineering and graph drawing theory to enable the automatic processing of graphical models and fundamentally enhance the user interaction mechanisms.

We extend the traditional definition of the term *pragmatics*—the question of how elements of a language should be used—to all practical aspects of handling a model in its design process, such as editing and browsing. The main problem with pragmatics in state-of-the-practice modeling IDEs is the widely accepted way of user interaction with diagrams which is of two-faced nature: What-You-See-Is-What-You-Get (WYSIWYG) Drag-and-Drop (DND) editing. DND here encompasses all manual layout activities that a modeler has to perform, such as positioning or setting sizes of graphical objects (nodes) or setting bend points of connections (edges). We do not distinguish whether such actions are real drag-and-drop operations with the mouse or are performed by keyboard.

When working with graphical models, it is useful to have an immediate graphical feedback on editing operations, hence WYSIWYG is not the problem. However, DND adds a lot of extra mechanical effort on editing diagrams. To quote a professional developer 15 years ago [Pet95], as we still find the same problems today: “I quite often spend an hour or two just moving

boxes and wires around, with no change in functionality, to make it that much more comprehensible when I come back to it."

With the standard editing paradigm one often ends up with exactly one static view for a subset of a model where the abstraction level is once decided—e. g., level of detail or subset of displayed nodes. To get a different view requires to start the editing process all-over.

In this thesis I propose a framework for automation of much of the manual efforts that had to be done before.

The Kitchen Story Let us think about a suitable image, an analogy to point out the contributions of this thesis and its potentials. Assume you are the owner of a bistro and now your task is to create a meal. You have a set of tools for that; pots, pans, rasp, stove, stirrer, a vegetable peeling knife etc. Preparing that meal will require some additional things like commodities and a recipe, unless you make up the recipe spontaneously in your mind while you cook. Obviously the success of your cooking undertaking depends on your kitchen skills or at least on how detailed the recipe is written and how well you follow it.

A substantial part of the preparation process is what you usually do not see in a cookery show on TV: manual preparation of vegetables and meat, washing, peeling, chopping and cutting. It is all effort prone and usually boring.

Now assume someone sends you an amateur cook into your kitchen who is capable of assisting you in that process. Given a recipe, he or she will prepare that meal for you. The benefits are quite obvious.

Now assume we make your cook smarter; you or someone else gives him or her a set of recipes and you may wish for a cooking style for your dish like french, italian or indish. The cook will choose a recipe by his or her own and serve you that dish.

Another step is to claim meta requirements to your meal. Maybe you have dietary requirements. You want a vegetarian dish or a light meal with little calories or something without lactose or little carbohydrates. Your cook will consider your wishes.

Even further your cook may consider the local season or holidays and choose a proper Christmas dinner or something refreshing on a hot day.

Wouldn't that be nice?

Somebody might comment on this story that all the fun of cooking is gone. This might be true. So if you like cooking, do it in your free time with and/or for your friends and family. However, remember that you are the owner and manager of your bistro and you need to consider both, the cooking and the management. With a chef in charge you may concentrate on higher tasks like choosing your kitchen style and concrete recipes and explore local habits on what kind of meal is desired to what occasions and put it into rules for your chef.

Now let us map this case back to MDE. To create graphical models you have an IDE with diagram editors and different tools like object creation tools, manual moving and resizing tools and so on. So far creating a diagram involves a lot of manual effort like placing of nodes and routing of edges. There we bring into charge a system capable of automatic layout of your diagrams, which presents interfaces to plug-in new layout algorithms like you add new recipes to your chef's cookbook. While automatic layout is state-of-the-*art*, it is far from being state-of-the-*practice* in MDE and is little integrated into the tools.

We build upon this functionality and define a process that consistently uses this basic layout mechanism. We introduce a *view management* system that, given certain conditions, will present a certain view onto your model automatically. Relations between these conditions and the resulting views may be given in an abstract form neglecting detailed decisions about layout details. It is like stating the rule that on Thanksgiving in the US a turkey roast with cranberry sauce would be suitable.

Like the chef chooses ingredients and a style of kitchen to cook them, meta requirements may be given to the system to optimize the layout results like a most compact diagram or a focus of certain aesthetics criteria like edge crossings or direction of flow. The system may choose layout algorithms and their parameters and especially the parts of the models to be presented in a view automatically according to the given context and hence present optimal views.

Using this view management technique results in new methods for standard use cases in an MDE process like editing of diagrams or visualizing of functional model simulation. Structure-based editing, text-to-diagram syn-

chronization or focus&context simulations are only some of the presented approaches that significantly will help in developing systems with graphical models and especially in taming complexity in large scale applications. Bon appetit.

1.1 Model-Driven Engineering

This work was inspired by members of the Real-Time and Embedded Systems research group of the Christian-Albrechts-Universität zu Kiel, students, and practitioners who had to work with up-to-date functional modeling environments to create real systems, e. g., for the avionics or railway domain [FvH09a, HFPvH06].

Our research target so far are *reactive embedded systems*, where we employ tool support mainly for the following tasks:

1. Creation and maintenance of *functional behavior models* of the target system.

   This comprises the editing process itself and handling of model files, which includes methods like version management and model comparison.

2. Reading of models.

   Understanding existing models requires a lot of diagram browsing. It is just like reading source code when learning about the functionality of some piece of a program. The source is the documentation and comments should help to understand it.

3. Analysis of models.

   Correctness of a model as well as learning or teaching the semantics of a modeling language can very well be shown by a simulation of the target system. The model is either interpreted or generated code is executed on the development system and interacting with a simulation of the environment. Visualization either in the original model or in a separate environment animation helps to understand the system state resp. environment state.

In all of those tools, introduced below and in Appendix A, we found ourselves wasting a lot of time with low level editing operations and the lack of proper mechanisms to visualize complex models, as we will discuss in the following.

First, we will discuss the state-of-the-practice, the approaches that can be found in almost all off-the-shelf modeling environments. We also examine which situations left us so hungry for enhancements that we started this work. In Chapter 2 we will give an overview of the state-of-the-art in this area, where different related communities try to contribute single answers.

Model-Driven Engineering (MDE), or alternatively Model Driven Software Development (MDSD), denotes software development processes where *models* are central artifacts which represent software entities on a high abstraction level [Est08]. "Model" is a very generic term, used in many sciences, see Stachowiak [Sta74] for an underlying theory. However, MDE is more about the processes around models, and only first approaches on the formalization of MDE are upcoming [Fav04].

The goal behind modeling is to increase productivity by maximizing compatibility between systems, simplifying the process of design, and promoting communication between individuals and teams working on the system. Graphical models are appealing and help to provide a common base for experts from different domains, especially if standardized like the UML. The graphical aspect aims at introducing intuitive language semantics and better displaying an abstraction of a system. A two-dimensional canvas gives more freedom to clearly present a view on a system than the one-dimensional textual representation, whether in a low-level programming language or a high-level specification scheme. However, such freedoms demand new design decisions from the developer, who now has to spend effort into the graphical representation in order to reap its benefits.

In textual programming most developers avoid to go back to assembler programming unless the application requires squeezing out last performance optimizations by manual tricks. So higher programming languages are ruling the development processes and one might ask why graphical representations are not yet standard practice. One problem is that there exists a growing set of modeling languages and development environments for them. Well known in the control engineering domain are Matlab/Simulink[2], LabView[3], and the Safety Critical Application Develop-

[2] http://www.mathworks.com/

[3] http://www.ni.com/labview/

ment Environment (SCADE)[4]. Control flow is better expressed with Statecharts introduced by Harel in 1987 [Har87], and numerous variants since then [PS91, Bee94]. The Ptolemy project aims at heterogeneous modeling of actor-oriented models which can be of control or data flow nature [EJL+03].

The best known initiative for MDE aiming at standardization is the Model-Driven Architecture (MDA), which is a registered trademark of the Object Management Group (OMG). Some of their standards are the Unified Modeling Language (UML) [Obj05] and Meta Object Facility (MOF) [Obj06]. The UML also defines a Statecharts variant. However, the focus of the UML is on syntax, and generally lacks precise semantics [FSKdR05]. Consequently, the UML is often regarded as too general, with respect to both its weak semantics and its bewildering multitude of languages (the UML 2.0 offers seven structural languages and six behavioral languages). This has lead to the concept of Domain Specific Languages (DSLs) [vDKV00]. It is by now standard practice to create new DSLs and to build custom editors and tools for these. Often, this is still done manually; alternatively, one may employ a framework that supports the generation of new graphical languages as proposed by Lédeczi et al. [ALABM+01] and denoted Model-Integrated Computing (MIC). The proposed Generic Modeling Environment (GME) or the Eclipse Platform with the Graphical Modeling Framework (GMF)[5] allow to synthesize customized graphical interactive drag-and-drop editors for new DSLs from some basic specifications. The creator of a concrete editor is called the *toolsmith*. The benefits of this generative approach are the short implementation times from an editor specification to an initially running diagram editor with a common set of features. However, the drawback is a relatively complex development process with a steep learning curve and difficulties when it comes to customization of the generated code. Therefore new approaches emerge, which go back to a purely API centric development of a graphical editor like Graphiti[6]. With both developments there now emerges a variety of new graphical languages from the user community, each for a special purpose or a special domain.

With this diversity of graphical formalisms without a real standard, the

[4] http://www.esterel-technologies.com/

[5] http://www.eclipse.org/gmf/

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

different technologies get developed and evolve rather independently. This includes different approaches to the pragmatics of model handling, i. e., how models are created, edited, visualized, inspected, simulated, compared and so on.

The tools considered here include Matlab/Simulink/Stateflow of The Mathworks as well as SCADE and E-Studio of Esterel Technologies, LabVIEW of National Instruments, Harel's Statemate, IBM Rational Rose Realtime and Software Architect, Visual Paradigm, Web-based tools such as Oryx [DOW08] and different Eclipse based tools such as the UML2Tools, MDT/Papyrus and the Ecore Tools. In the following we will discuss the major properties of the tool user interaction that are relevant for this thesis. Screenshots and some further overview and characterization of the tools are given in Appendix A.

1.2 Problem Statement

The problem tackled in this thesis is the common way of user interaction with development tools in an MDE design process. This is elaborated further in the following, addressing in turn (1) the *low-level mechanics* of design tools in MDE, (2) *fixed hierarchy* that retain the abstraction level of diagrams, (3) *lacking view management* that misses the separation of models and their views and thus limits the flexibility of diagram reuse, and (4) *limited data visualization* during model execution that renders interactive debugging and analysis of models very effort prone.

Problem 1: Low-Level Mechanics

What-You-See-Is-What-You-Get (WYSIWYG) editing was first introduced 1974 with the text editor *Bravo* of Xerox PARC [Mye98]. Taylor comments on WYSIWYG for typesetting and states its two-faced nature [Tay96]:

> "Why has WYSIWYG succeeded so spectacularly, while other typesetting approaches have languished? I think WYSIWYG's main appeal is that it appears to offer its users superior cybernetics – i. e., feedback and control. To the extent that you can trust its authenticity, the screen gives immediate feedback. Acting on that feedback, the user then has immediate control. And people like having feedback and control.[...]
>
> It is worth remarking in this context that while WYSIWYG may have won the hearts and minds of designers through "superior cybernetics," the degree of control that such programs offer may be more illusory than real. Or perhaps it is more accurate to say that desktop publishing programs let you fiddle interactively with the details of your typography until the cows come home, but they do not let you control the default behaviors of the composition algorithms in a way that efficiently and automatically delivers the kind of quality typography that was formerly expected of trade compositors."

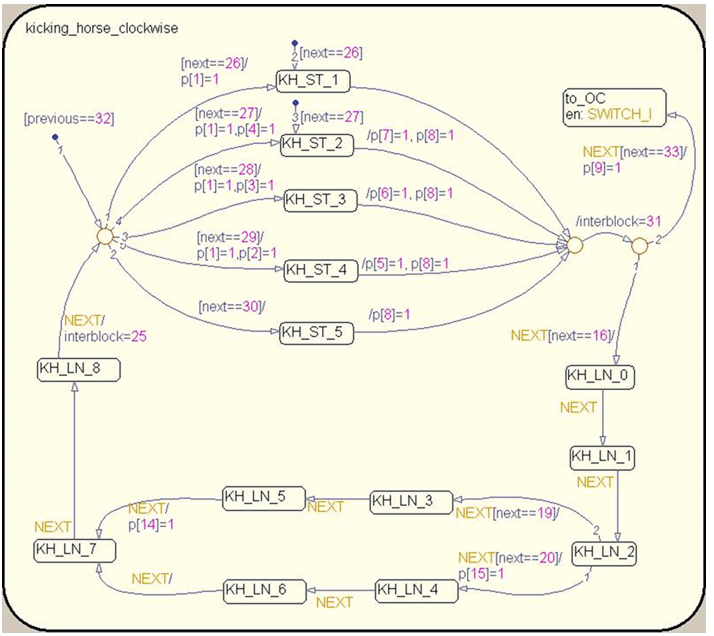
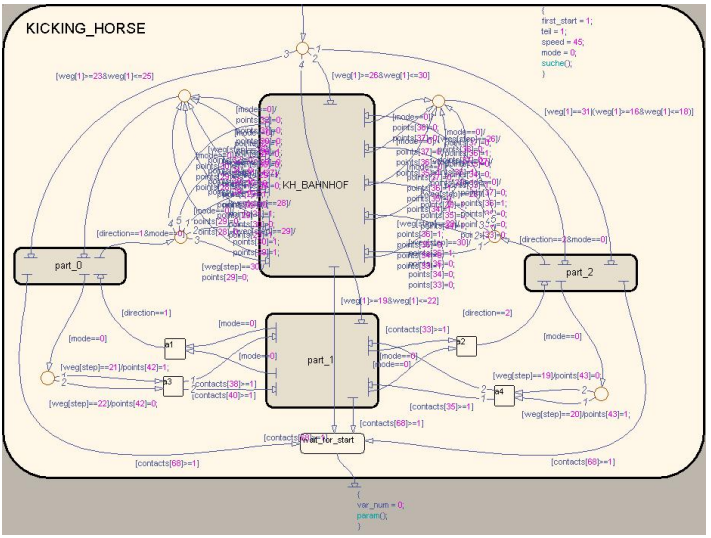


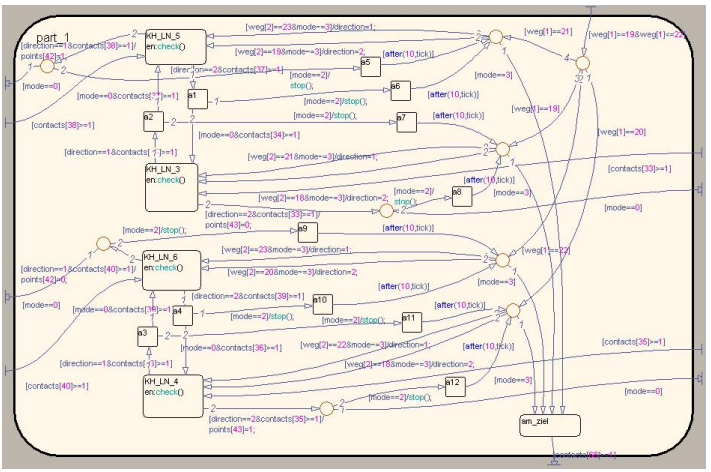
Figure 1.1. Well-done manual layout. The placing is related to the real track scheme of the railway installation and hence carries some additional advice how to read it.

While typesetting tools may or may not have improved, this two-faced property holds the same still today for graphical modeling tools. The direct graphical feedback of WYSIWYG editors enables the editing process for a broad range of users due to its immediate response without any technical barriers. However, the editing process itself involves the effort-prone manual placing and routing of nodes and edges in a diagram. Hence, we want to distinguish between the positive aspect of immediate feedback, to which still the notion of WYSIWYG Drag-and-Drop (DND) fits (see later).

The *mechanics* of development denote the manual work the developer has to perform to achieve the desired effect such as creating or editing a graphical model. It means the actual user interaction between human



(a) Overlapping elements make this manual layout unusable.



(b) Quickly done manual layout. A low level of abstraction makes the diagram hard to grasp.

Figure 1.2. Different qualities of manual layout in a railway example.

and machine: The keystrokes on a keyboard with special sequences or key-combinations, mouse movements and clicks, drags and hovers.

The term Drag-and-Drop originates from editors where the IDE contains a toolbar or *palette* from where one drags new graphical objects to the canvas and drops them there. Additionally, existing objects are moved and resized by such dragging operations. We will still use this term when there are slightly different mechanics, e. g., when the keyboard is involved to do pixel-by-pixel movement of nodes. Sometimes, DND is also referred to as *freehand* editing in the literature [Min06]. We will use both terms interchangeably.

Usually there are multiple sequential atomic mechanical steps necessary to perform one higher-level operation. For example for the task of *adding a new node* into an existing diagram, one needs to add a new state at some position in the diagram, either by dragging it from the palette into the target location with the mouse or by appropriate keyboard commands. Additionally one must adapt the original layout of the diagram: there must be new empty space by moving existing objects around, resizing parent objects and redirecting or splitting any relevant references or connections. The latter task set only consists of *enabling steps* [GP96] and usually takes much more effort than simply adding a new object to an empty canvas.

We do not state that an automated process of placing and routing of diagrams can always be better than a manual layout. Often, a well-done manual layout can present a diagram in an optimal light. Consider a railway example modeled in Stateflow, presented in Figure 1.1 and Figure 1.2. Figure 1.1 shows a proper abstraction level and a placing of nodes that correspond to the track scheme of a railway installation. This gives the user some advice how to read the diagram if he or she is familiar with the track scheme. This would be very difficult for an automatic algorithm to achieve. However, you could still question whether such *secondary notation* should be applied to models at all. Figure 1.2a and Figure 1.2b are diagrams for the same railway application. Here, the developers have chosen a different level of abstraction and only provide a quick-and-dirty layout—which is almost unreadable due to violations in essential aesthetics criteria like overlaps. Unfortunately, due to the lack of time, we see diagrams of the latter quality far more often than of the first. It appears from industrial projects that this

is also true for many models in companies for productive use.

To summarize, usually the only way to create a graphical model is to manually "draw" it. This implies that the developer also manually applies the layout, i. e., sets the coordinates for all nodes and bendpoints for edges, etc. Afterwards only this single *view* of the model is available. If a different view is required, the layout must be changed again manually.

Problem 2: Fixed Hierarchy

For each diagram the level of abstraction is important for the comprehension of the model. If the developer has no choice about what elements get visible in the diagram because the tool enforces a specific pattern, e. g., a one-to-one relationship between model elements and diagram elements, then the level of abstraction is fixed. However, some tools allow choices and hence allow to optimize the diagram to the current use case.

A fixed view is especially disadvantageous when it comes to the visualization of compositional relationships, which get expressed by one entity being contained by another. This is a common pattern, for example

- in state diagrams such as Statecharts where one state can contain substates,
- actor-oriented data flow diagrams where multiple entities can be composed to some operator or subsystem in order to be reused in a higher level data flow or
- many languages where requirements for modularization introduce *packages* to bundle related elements, which is much used in the UML.

There are two ways how editors commonly handle these hierarchies.

Hidden Hierarchy Only one level of hierarchy is displayed on the canvas. The element containing child objects may have some indication that it is a composed object (a special icon maybe), but the contents are hidden in this view. The children constitute a separate view, usually on a different canvas in a different window. Navigation between these views must be performed by the user.

Visible Hierarchy All elements of all hierarchy levels are shown on the canvas. Elements containing sub-elements are enlarged and the children are visible within the body graphical element of the parent.

The approaches have different properties: The first one is modular, because the graphical layout of the interior of a node does not influence the size of the node. However, it might be harder to gather the meaning of a diagram, if the contents of composite nodes are relevant for its understanding. Then the user manually has to navigate between the levels of hierarchy.

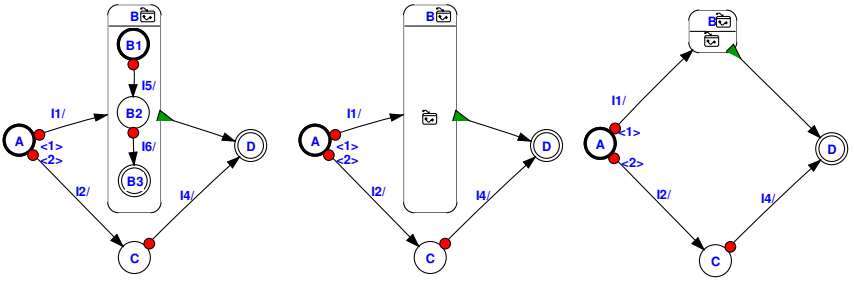
The second approach presents all elements in one view and, hence, for smaller diagrams it might be easier to understand the meanings as all hierarchy levels are revealed. Nevertheless, the goal of abstraction by introducing hierarchy is not fulfilled as no low level information is hidden. Editing operations might get effort-prone because changes in layout in lower hierarchy levels can affect higher levels. An example is shown in Figure 1.6 on page 20, which combines state machine elements with lower-level data flow elements with a visible hierarchy. Obtaining that special design and layout required a developer effort that is usually not acceptable in typical productive environments.

Some tools allow "hard-coded" mixtures of both approaches: for each hierarchical parent element one may choose whether the children are displayed embedded in the same view or get hidden from the user. Sometimes this choice can be done exactly once at the creation time of the parent. However, there exist viewers that allow to change this choice later on, e. g., Esterel Studio or Eclipse GEF based editors. This *folding* or *unfolding* only happens within the single static view. The parent can be visualized in multiple different ways:

1. Large bounding box with embedded children elements (Figure 1.3a).

2. Large bounding box while hiding all children (Figure 1.3b).

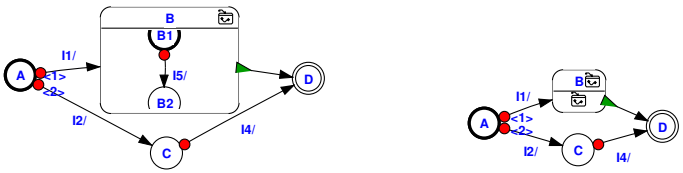
   When folding a parent, this does not really help to reduce the complexity of the diagram, because the size is still the same and hence you might end up with a lot of big empty boxes. Eclipse uses this way to fold its *compartments*.



(a) Fully visible hierarchy

(b) Hidden hierarchy with large parent

(c) Hidden hierarchy with small parent



(d) Visible hierarchy with medium size parent (smaller than children)

(e) Hidden hierarchy with small parent and manually optimized layout; the most compact representation.

Figure 1.3. Alternatives for hierarchy visualization (E-Studio).

3. Small bounding box with embedded children elements. Scroll bars are added to the body of the parent to navigate within the children (Figure 1.3d).

   This allows to reduce the size of the parent to even smaller bounds than the bounding-box of its children. Eclipse and E-Studio support this. Although this avoids to influence the size of the parent when its children change, it is very inconvenient to manually navigate within a parent to understand its children and produces awkward results in printouts.

4. Small bounding box with just a name or textual references to the children (Figure 1.3c).

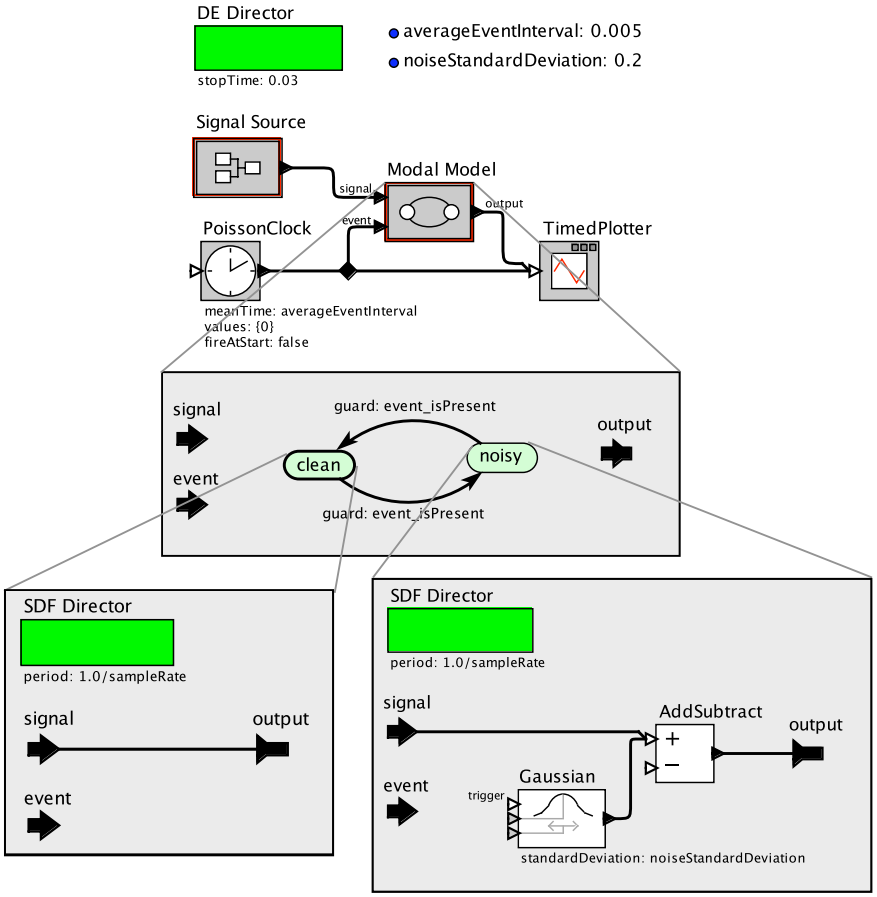


Figure 1.4. Ptolemy in general only has hidden hierarchy. However, for presentation of models in papers, the developers usually use this special side-by-side rendering of hierarchical elements [Lee09].

The last option seems to be fairly practical as in folded mode an entity will use less space on a canvas. There might be a different view of this

hierarchy level where this space gain might be well used to create a more comprehensible overall layout to make the diagram look less complex (Figure 1.3e). Unfortunately, this different view still must be created manually. The developer has to push together all elements in order to fill the emptied space. To unfold the parent again, these beautifying steps have to be undone again manually. Little of the tools we have seen so far allow to save multiple views persistently, hence these manual steps have to be performed again to switch between fold/unfold states of graphical objects with optimized layouts.

The Ptolemy II Editor Vergil only shows hidden hierarchy. This makes it difficult to present complex systems where the behavior is distributed to multiple hierarchical levels. Therefore the Ptolemy developers usually create hand-crafted images like Figure 1.4, where the composite actors are rendered somewhere next to their parents as in an exploded view drawing [Lee09]. This has the disadvantage that the containment relation is no longer an intrinsic property of the drawing itself. So far no tool support for this exists yet.

Problem 3: Lacking View Management

Most existing modeling tools have a set of manually created static views—e. g., one per composite element as discussed above—of the model to present. In order to investigate and explore the model, the user needs to navigate between these views. In the existing tools considered here this is done manually by the user. Either the user chooses parent objects from a list or a tree view, or he or she can navigate in the graphical view from the top element to sub-elements and dig through to the desired elements. New elements are opened in the same canvas that was used before, hence one immediately loses the context of the node one is delving into. Otherwise new elements get opened in new windows with each having its own canvas. This way a parent and its children can be viewed side-by-side where the zoom level of each window needs to be reduced in order to fit onto the screen. Obviously this does not scale: for keeping the overview of a whole complex model one might end up with a screen cluttered full of model windows, each so small that you can hardly guess

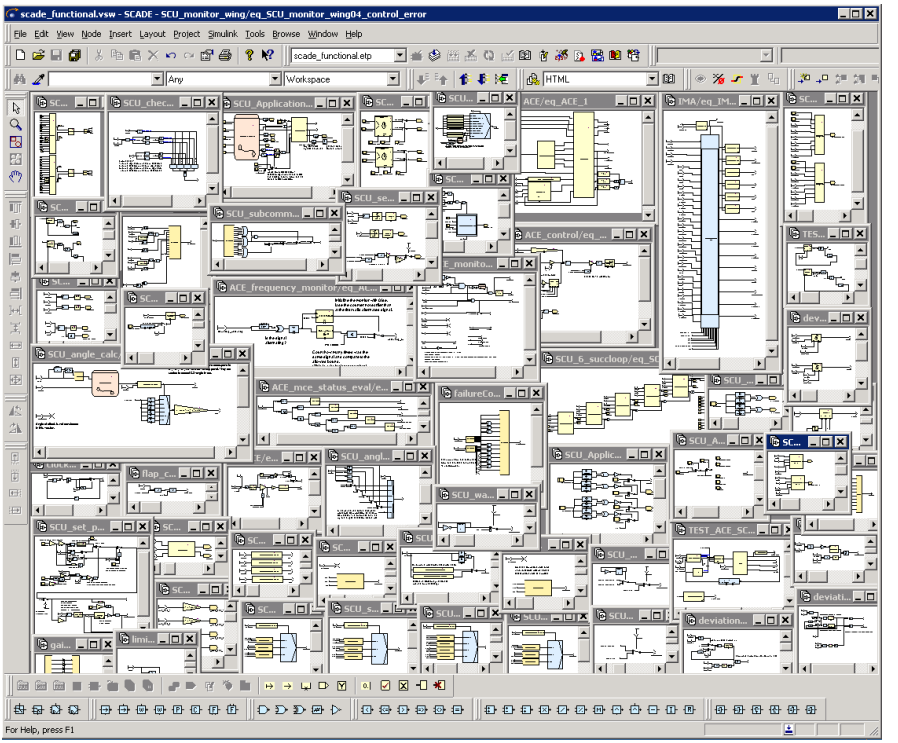


Figure 1.5. Illustration for the lack of proper view management: showing the whole system entails loosing details, windows get too small to be usable.

what they comprise. Figure 1.5, which stems from an avionics application [FKRvH06] implemented in SCADE of Esterel Technologies, illustrates this point. In this view details are lost and interaction (e. g., editing) is hardly possible. At the other end of the spectrum, if one would present just one of these model windows that covers one specific part of the system, the context is lost. This is shown in Figure 1.7, where in many tools like in SCADE, Ptolemy or Eclipse, a tree is employed to compensate this context loss. From our experience, modelers typically try to select the two or three

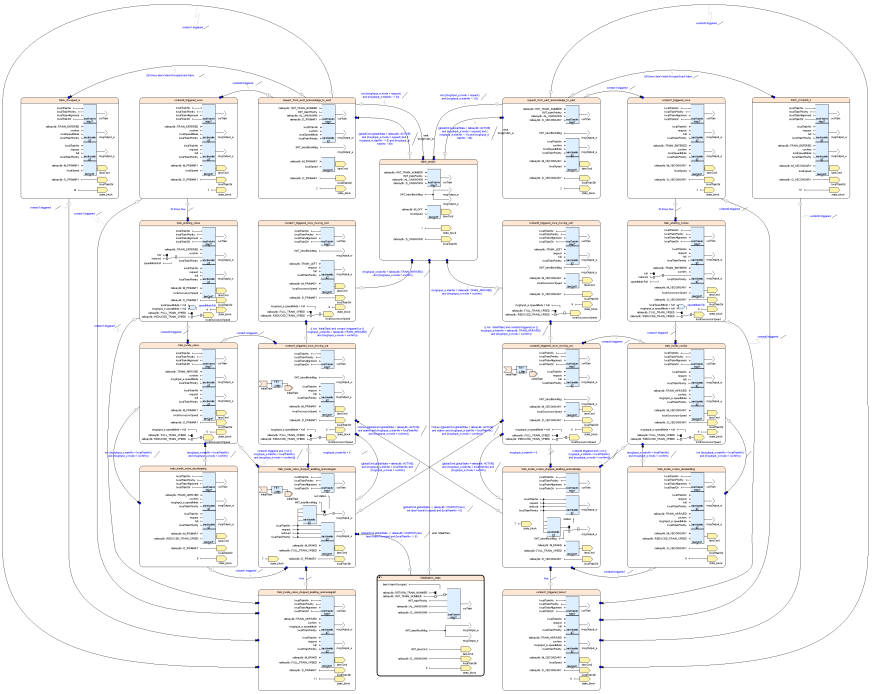


Figure 1.6. Big state diagram with embedded data flow and visible hierarchy. The mechanical drawing part of this consumed approximately 50 person hours.

currently most relevant model windows, and spend a fair amount of time to arrange windows accordingly on the screen, if this is possible at all (cf. Appendix A).

Problem 4: Limited Data Visualization

Only for state machines it is common practice to use the diagram view itself for graphical feedback about the internal data of a simulation run. In that case it seems quite natural to highlight active states. However, when it

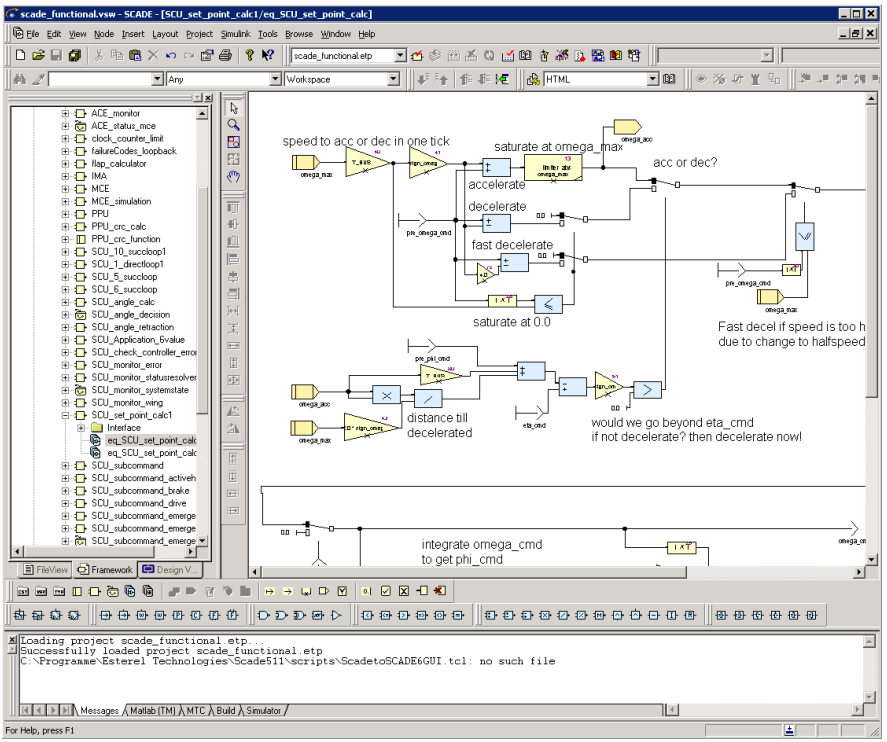
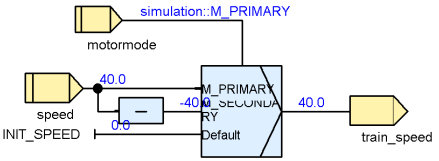


Figure 1.7. Usual focus to a single subsystem loses the context. Often tools try to compensate with presenting the context in an additional tree view of available subsystems.

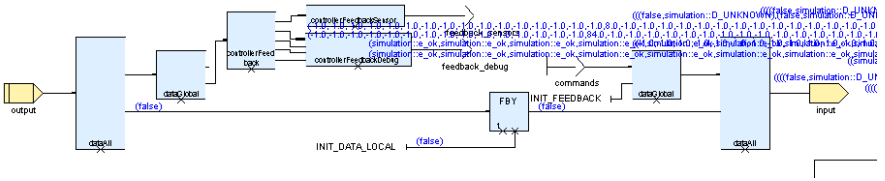
comes to more complex data, the solutions become unsatisfactory.

SCADE is an actor-oriented data flow language that allows to display the data in a simulation instant in the graphical view. Small blue text labels are placed near the connections, which present the actual values as shown in Figure 1.8. This is only useful for small data tokens, because big data types such as arrays or structures usually break the fix layout.

Therefore most tools offer ways to present the data in separate views.



(a) Data (blue labels) shown the diagram is useful for small data tokens.



(b) For big data types (e. g., arrays in this real-world railway application [HFPvH06]) the presentation breaks basic aesthetic criteria (e. g., overlapping)...

(c) ... and results in infeasible views. Here, the diagram itself is the black part in the left (same as the cutout above). The data are the blue lines, which protrude the canvas.

Figure 1.8. A simple strategy to display data in the diagram in SCADE

For example the Ptolemy II Editor Vergil can open scope plotters in separate windows. It also supports a 3-dimensional animation of the environment as shown in Figure 1.9. External data views are also available in SCADE, Matlab/Simulink and LabVIEW.

With this solution the problem remains of connecting the data displays to the actual model elements. Especially when there are many interesting data sources, the view can get cluttered with many such views.

Other approaches are

Mouse Hover Simulink allows to display data on mouse hover over a specific connection. This is similar to the label overlay as it leaves the layout of the diagram untouched but presents the feedback in it. It avoids

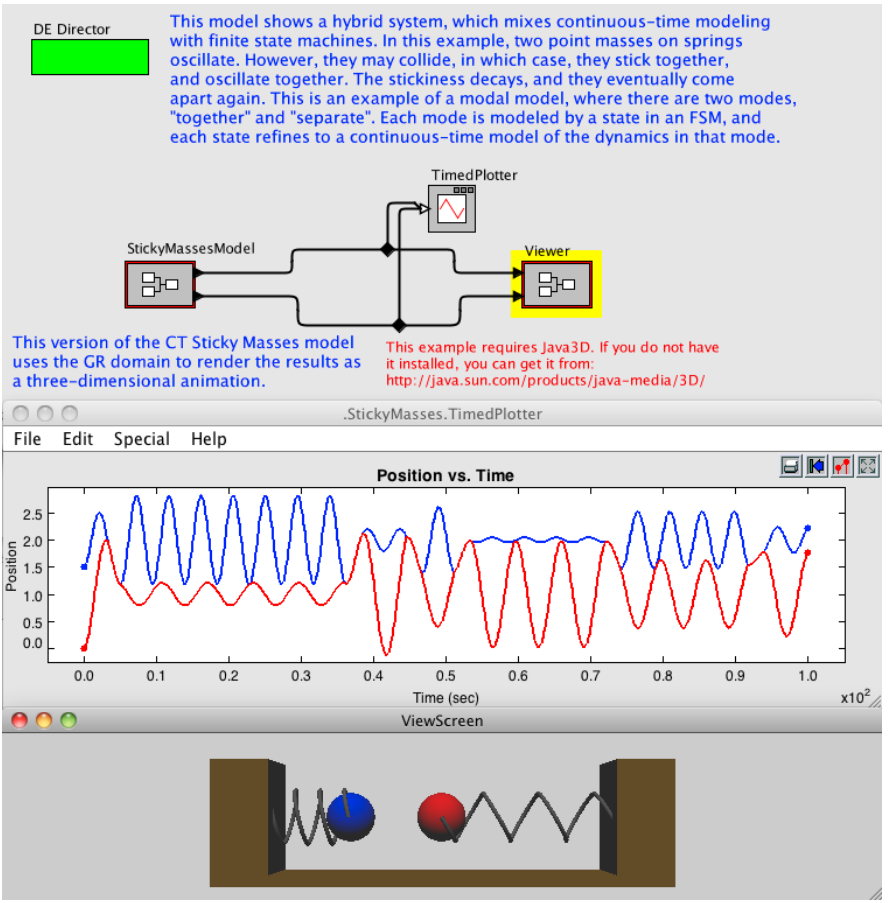


Figure 1.9. Usually external views show plotters or even graphical animations such as Ptolemy II.

the overlapping of data labels but adds the effort of manual mouse navigation.

Special Data Nodes Most data flow languages offer special nodes in the

language syntax itself. They usually do not have a semantic meaning for the model but are used as data sinks that use their icons to display values as text or scope. In this case the drawback is that the nodes enlarge the diagram and the choice of values to be observed is fixed. Therefore the abstraction level of displayed data is only chosen once and therefore might display either too much or too little information for a given use case.

To summarize, modeling tools commonly rely on archaic mechanisms for presenting models, such as the generic window management capability provided by the operating system or explosive zooming. Means to re-use the diagram itself for graphical feedback are limited. There are little built-in, smart navigation or data display facilities.

1.3 Contributions

The major contributions of this thesis are threefold:

1. An interpretation of the notion of pragmatics, orthogonal to syntax and semantics, to denote all practical aspects of handling models in a design process of MDE. Additionally, there is a categorization of modeling pragmatics, based on the Model-View-Controller pattern. It maps the use cases of the MDE process to the three different aspects of MVC to support separation of concerns in the discussion, analysis and implementation of modeling pragmatics. This is introduced in Section 3.1 and used to structure Chapter 3.

2. A proposal on pragmatics-aware modeling, which systematically employs automated layout to enhance design productivity. As a central aspect a *view management* component is introduced that allows to dynamically change or synthesize views of models. This strictly separates models from their views and automates many issues involving the view, while the user can concentrate on the model. It employs means of automatic layout combined with filtering techniques to show only relevant parts of models. Together with other graphical effects this enables reuse of the diagram of a model. It gives enhanced graphical feedback and helps to trace model properties to assist in a specific use case or interest of the user. View management enables to define such use cases in an abstract manner. Examples are presented such as structure-based editing, text synchronization and focus & context views in different situations, e. g., an interactive simulation run of a functional model. These concepts are presented in Section 3.2–3.4.

3. A report on how this pragmatics-aware modeling proposal has been put to practice with KIELER, the Kiel Integrated Environment for Layout Eclipse Rich Client. Its goal is the validation and dissemination of the concepts presented in this thesis. A focus is laid on genericity to be applicable to a wide range of modeling languages. The generic implementation of the key concepts is presented in Chapter 4. This furthermore encompassed the development of support projects:

- ▻ A set of graphical model editors has been developed and integrated into KIELER to validate the approach. The Thin KIELER SyncCharts Editor (ThinKCharts) is the major demonstrator. Other examples demonstrate the genericity of the approaches by opening the range of validation to other specialized editors, specifically for actor-oriented data flow models and some UML languages. Furthermore, the integration of KIELER automatic layout into the Ptolemy II graphical Editor Vergil validates the approach and APIs beyond the scope of Eclipse. This is presented in Chapter 5.
- ▻ A generic extensible infrastructure connects layout algorithms with concrete diagram editors. Initial layout libraries or algorithms have been integrated, such as the GraphViz library and a customized layer based algorithm supporting port-constraints.
- ▻ A twofold framework to model execution in Eclipse has been developed. On a low level a data exchange and user interface infrastructure is implemented where simulation components can be plugged into. On a higher level the semantics of a concrete modeling language can be defined by a mapping to the modeling system Ptolemy II that works as a simulation engine in the background. This supports the visualization approaches that work in the context of model execution. The latter two subprojects are presented in Chapter 6.

1.4 Related Publications

Parts of this thesis were already published in research papers. The following provides an overview ordered by topic. The publications are also noted in Figure 1.10 for categorization (see Chapter 3).

Background

[FKRvH06] Hauke Fuhrmann, Jens Koch, Jörn Rennhack, and Reinhard von Hanxleden. Model-based system design of time-triggered architectures—an avionics case study. In *25th Digital Avionics Systems Conference (DASC'06)*, Portland, OR, USA, October 2006

This paper motivated some of the state-of-the-practice descriptions in Section 1.1.

[HFPvH06] Stephan Höhrmann, Hauke Fuhrmann, Steffen Prochnow, and Reinhard von Hanxleden. A versatile demonstrator for distributed real-time systems: Using a model-railway in education. In Amund Skavhaug

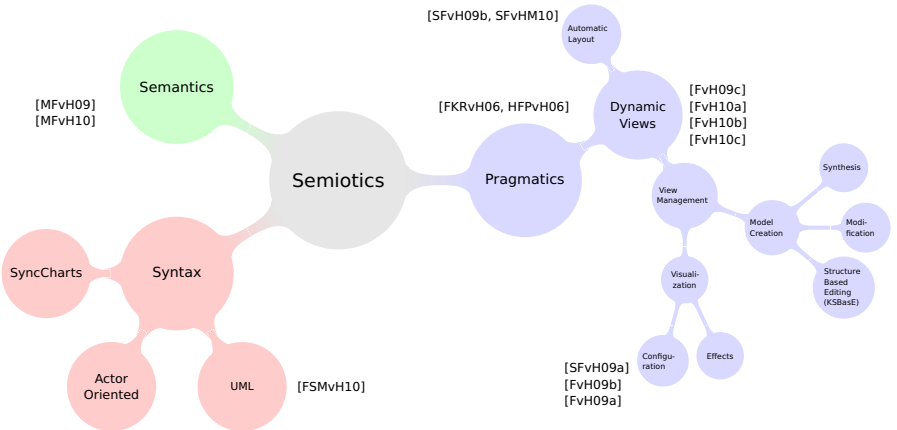


Figure 1.10. Overview of related publications in the KIELER project.

and Erwin Schoitsch, editors, *Proceedings of the ERCIM/DECOS Dependable Embedded Systems Workshop at Euromicro 2006*, Cavtat/Dubrovnik, Croatia, August 2006

This work illustrates further the problems of graphical model-based design presented in Section 1.1.

Foundations

[FvH10a] Hauke Fuhrmann and Reinhard von Hanxleden. On the pragmatics of model-based design. In *Foundations of Computer Software. Future Trends and Techniques for Development—15th Monterey Workshop 2008, Budapest, Hungary, September 24–26, 2008, Revised Selected Papers*, volume 6028 of *LNCS*, 2010. doi:10.1007/978-3-642-12566-9_7

In this paper we present the ideas and concepts of our enhancements to graphical modeling and introduce the interpretation of the notion "pragmatics." This is the main part of Chapter 3. It was earlier presented in a technical report [FvH09c].

[FvH10c] Hauke Fuhrmann and Reinhard von Hanxleden. Taming graphical modeling. In *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, LNCS, Oslo, Norway, October 2010. Springer

This paper presents the foundations and implementation of the project: the term "pragmatics", meta layout and view management (Chapter 3) and parts of the KIELER implementation (Chapter 4). A longer version was published as a technical report [FvH10b].

Use Cases of View Management

[FvH09a] Hauke Fuhrmann and Reinhard von Hanxleden. Enhancing graphical model-based system design—an avionics case study. In *Conjoint workshop of the European Research Consortium for Informatics and Mathematics (ERCIM) and Dependable Embedded Components and Systems (DECOS) at SAFECOMP'09*, Hamburg, Germany, September 2009

The aerospace case-study of focus and context of Section 4.4 was the topic in this paper, and was first presented in a technical report [FvH09b].

[SFvH09a] 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'09)*, Potsdam, Germany, 2 June 2009

In this paper we present some foundations of automatic layout (subsection 3.2.1) and the visual comparison of graphical models (Section 4.7).

[FSMvH10] 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

In this paper we present approaches and problems of applying the concepts to the UML as discussed in Section 5.3.

Support Projects

[SFvHM10] Miro Spönemann, Hauke Fuhrmann, Reinhard von Hanxleden, and Petra Mutzel. Port constraints in hierarchical layout of data flow diagrams. In *Proceedings of the 17th International Symposium on Graph Drawing (GD'09)*, volume 5849 of *LNCS*, pages 135–146. Springer, 2010. doi:10.1007/978-3-642-11805-0_14

This paper addresses automatic layout of actor-oriented models with port constraints as presented in subsection 3.2.1 and the integration into the Ptolemy/Vergil editor (Section 5.4). This work also appeared as a technical report [SFvH09b].

[MFvH10] Christian Motika, Hauke Fuhrmann, and Reinhard von Hanxleden. Semantics and execution of domain specific models. In *2nd Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe 2010) at conference INFORMATIK 2010*, GI-Edition – Lecture Notes in Informatics (LNI), Leipzig, Germany, September 2010. Bonner Köllen Verlag

This work presents the execution framework in general and the usage of Ptolemy as semantics specification in particular in Section 6.1. It has also been published as a technical report [MFvH09].

Also the corresponding parts of the related work in Chapter 2 are based on the related work sections in these papers.

Theses

The work presented in this dissertation builds on the following theses (diploma theses, bachelor's theses, master theses, student research projects), which were supervised by the author and categorized in Figure 1.11:

▻ Steffen Jacobs. Konzepte zur besseren Visualisierung grafischer Datenflussmodelle. Student resarch project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, February 2007. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sja-st.pdf`

This thesis evaluated effects of view management.

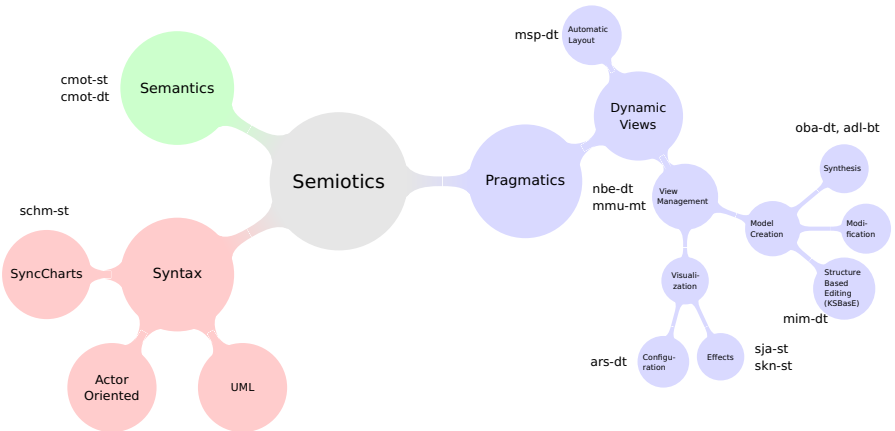


Figure 1.11. Overview of related theses in the KIELER project.

▻ Steffen Jacobs. Automatisierte Validierung von Modul-Konfigurationen in der Integrierten Modularen Avionik. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, January 2008. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sja-dt.pdf

The topic of this thesis was the validation of special aspects of aerospace applications. It introduced Eclipse as a common platform and target for the KIELER implementation.

▻ Arne Schipper. Layout and Visual Comparison of Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2008. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ars-dt.pdf

The thesis contains a first implementation of the KIELER Infrastructure for Meta Layout (KIML) (see Section 4.2) and approaches to visual comparison of graphical models.

▻ 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

This thesis presents the basic SyncCharts editor implementation with a rudimentary implementation of the attribute-awareness mechanism of diagrams.

▻ 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

The topic of this thesis was the development or adoption of layout algorithms for actor-oriented data flow languages with port constraints, hyperedges and orthogonal routing. The outcome was mainly a layer-based algorithm implementation.

▻ 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

The topic of this thesis was the design of a textual SyncCharts specification language and the synchronization with a corresponding graphical view.

▷ Nils Beckel. View Management for Visual Modeling. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, October 2009. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nbe-dt.pdf

This thesis aimed at a generic implementation for view management.

▷ Christian Motika. Modellbasierte Umgebungssimulation für verteilte Echtzeitsysteme mit flexiblem Schnittstellenkonzept. Student research project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, October 2007. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-st.pdf

This thesis presented the implementation of an environment simulator for a railway system that later got used in the KIELER Execution Manager (KIEM) project (see Section 6.1).

▷ Christian Motika. Semantics and execution of domain specific models—KlePto and an execution framework. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2009. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-dt.pdf

This thesis contains the implementation of the KIEM and KIELER leveraging Ptolemy Semantics (KlePto) (see subsection 6.1.2) with a case study of SyncCharts as simulated language.

▷ Adriana Lukaschewitz. Transformation von Esterel nach SyncCharts in KIELER. Bachelor thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2010. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/adl-bt.pdf

The topic of this thesis was to re-implement a model-to-model transformation from the Esterel synchronous language to SyncCharts, porting it from the KIEL context [PTvH06] to KIELER.

▻ 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

This thesis describes the implementation of the structure-based editing framework.

▻ Stephan Knauer. KEV – KIELER Environment Visualization – Beschreibung einer Zuordnung von Simulationsdaten und SVG-Graphiken. Student research project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2010. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/skn-st.pdf

This thesis aimed at a better integration of environment visualization (KEV) into KIELER. It based on the prior *modelgui* application by Steffen Jacobs and me.

▻ 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

This thesis improved view management.

Implementations of the topics in this thesis are part of the open source KIELER project. User and API documentation, source code as well as executable binary distributions are published on-line: http://www.informatik.uni-kiel.de/rtsys/kieler.

The implementation of the Ptolemy layout integration (Section 5.4) is published as open source part of the Ptolemy tool suite by UC Berkeley: http://www.ptolemy.org or http://ptolemy.eecs.berkeley.edu/.

1.5 Outline

This thesis is organized as follows. This section so far gave a brief introduction to Model-Driven Engineering in Section 1.1 followed by the problem statement in Section 1.2. I clarified my contributions in Section 1.3 and categorized related publications in Section 1.4.

The related work is discussed next in Chapter 2.

Chapter 3 introduces the central proposals on how to tame graphical modeling. Section 3.1 introduces pragmatics, which together with syntax and semantics constitutes the field of semiotics. Figure 1.12 illustrates this further, and also presents a graphical, somewhat abstract overview of chapters 3–6. Variations of this picture will be used throughout this thesis to help orient the reader. Terminology and a categorization approach is also given in Section 3.1, where I categorize the concepts with the Model-View-Controller (MVC) pattern (also discussing Figure 1.12). Section 3.2 covers the *view* of MVC, it explains what role diagrams play and lays the foundations for the following stepping stones presented as specific graphical effects that can be applied to diagrams. From pure automatic layout in subsection 3.2.1,

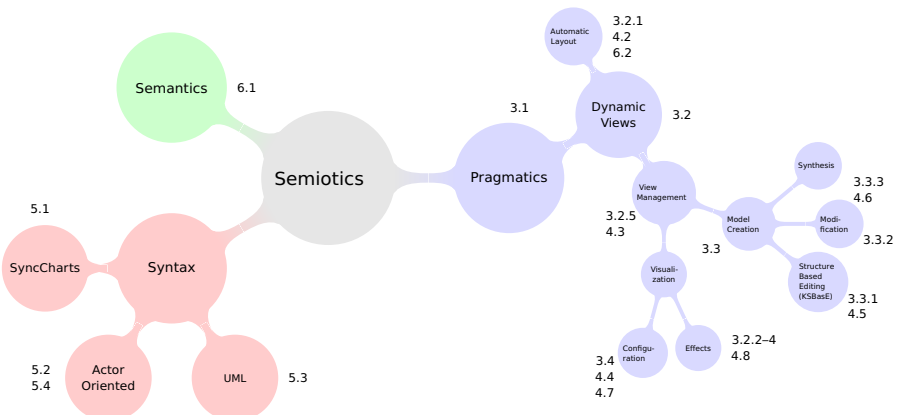


Figure 1.12. Overview of the KIELER projects, categorizing the sections.

we go to abstraction mechanisms like smart filtering in subsection 3.2.2. subsection 3.2.5 presents how *view management* logic employs the former concepts to dynamically and interactively present custom views on models. *Meta layout*, presented in subsection 3.2.6, interacts and parametrizes layout algorithms to customize and optimize such synthesis of different diagrammatic views on graphical models. In the following Secs. 3.3 and 3.4 we examine use cases that build upon the automatic layout functionality to create and maintain the *model* and to *control* it in simulations or analyses respectively.

Chapter 4 illustrates these concepts with the open source Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER). Next to the main projects for meta layout (Section 4.2) and view management (Section 4.3), it discusses multiple fields of application, such as model simulation (Section 4.4) and editing—structural (Section 4.5) and textual (Section 4.6)—and comparison (Section 4.7) and the environment animation (Section 4.8). It ends with an experimental evaluation in Section 4.9.

Demonstrator editors get presented in Chapter 5. The main case-study is performed with the Thin Kieler SyncCharts Editor (ThinKCharts), presented in Section 5.1. Actor-oriented data flow languages and the UML are discussed in Section 5.2 resp. Section 5.3. Integration of a layout algorithm into the external Ptolemy II editor Vergil is presented in Section 5.4 to validate some basic API even beyond the scope of Eclipse.

The KIELER implementation builds upon the existing Eclipse platform to leverage as many synergies with existing projects as possible in order to concentrate on the mentioned topics. However, the platform had to be extended by certain side aspects next to the main contributions of this work, presented in Chapter 6. Section 6.1 introduces the execution framework that allows to validate the approaches in the case of model simulation runs. Section 6.2 discusses the connection to concrete automatic layout algorithms, namely the GraphViz library and the development of a customized layouter that supports port constraints to enlarge the family of supported languages.

The thesis concludes in Chapter 7 and gives some hints to further ongoing work.

Chapter 2

Related Work

As this work is an interdisciplinary task, there is a large body of related work emerging from related communities.

2.1 MDE and Software Visualization

The MDE community employs means of user experience enhancements largely orthogonal to the ones presented here [SGD05, Est08].

Model transformations are a well-established technique to achieve consistency between model entities produced at different model life-cycle stages. GenGEd [Bar02] modifies visual languages using graph grammars and graph productions with predefined production sequences. We here instead propose responsive model manipulation by interactively triggering in-place model transformations like QVT [Obj04] or, as presented in this thesis, *Xtend* from the Eclipse Model-to-Text (M2T) project.

There are multiple recent approaches to creating model-to-model transformations *from examples* instead of complex transformation languages [BLS+09]. It would be interesting to combine such approaches with the structure-based editing framework presented in Section 4.5 to give the user very natural ways to define custom editing operations him- or herself. Also transformation languages based on triple graph grammars [BEK+06] could augment structure-based editing by graphical views on the transformations themselves.

The field of *Human Centered Software Engineering* [GGB+05] also addresses usability and productivity. However, these approaches mainly focus on the question of how to make the best user experience with a given

product. In contrast, we here try to enhance the development process itself with novel tool support.

Another related community focuses on software visualization [Die07], which mainly presents what we call *effects* on graphical views (cf. subsection 3.2.5). However, Charters et al. address themselves the fact that software visualization effects have been investigated to their limits and are demonstrated in standalone isolated tools only. Their future depends on whether the community will manage to better integrate the approaches [CTM03]. This is exactly one goal of this thesis. We also employ the notion of *focus & context* by Card et al. [CMS99], see subsection 3.2.4. Musiel and Jacobs [MJ03] apply this technique to UML class diagrams, using notions of *level of detail* and a rudimentary specialized automatic layout algorithm. The approach to view management presented here generalizes such ideas by orchestration of software visualization concepts (effects) with the context (triggers) in which they should be applied to dynamically synthesize graphical views on models.

Developments of Minas et al. show similar goals for the generative creation of diagram editors like in Eclipse. However, their work is implemented in the open source but standalone diagram editor generator framework *DiaGen* since 1995 [VM95]. DiaGen uses graph grammars to specify editors and create the target diagram editors. More recently, *DiaMeta* allows to use meta models instead of graph grammars, similar to Eclipse-based approaches like GME or GMF [Min06]. However, the authors see advantages on the graph grammar side such as extracting syntax directed rules to create models [HM10]. Next to these main approaches, Minas et al. address some questions about usability and especially regarding layout. Maier and Minas introduce simple rule-based Statechart layouters that still focus freehand editing [MM08] and recently enhanced these by an approach where they also try to separate the layout algorithm from a concrete diagram syntax [MM09]. Quite similar to ideas of the KIELER Infrastructure for Meta Layout (KIML) presented in Section 4.2, they present a metamodel for graph layouts and map this to metamodels of concrete diagram syntaxes. An interesting feature is the composition of multiple different layout rules to a whole layout result. This, unlike in KIML and Eclipse GEF, includes the internal placement of primitive node attributes like node name labels or

class diagram attribute lists. From different perspectives this can be seen either as an advantage or as a drawback. Their approach still focuses on an iterative manual editing with much user interaction, while we prefer a complete layout from scratch, which can take more variables into account and therefore produces better quality layout results. Furthermore, their graph layout metamodel seems to consider only plain graphs and does not yet support advanced graph features such as ports or hyperedges as KIML. Only a small set of plain graph layout algorithms has been implemented from the literature into their layout rule system.

The KIEL project [PvH07] evaluated the usage of automatic layout and structure-based editing in the context of *Statecharts*. It provided a platform for exploring layout alternatives and has been used for cognitive experiments evaluating established and novel modeling paradigms. However, it was rather limited in its scope and applicability, hence, it has been succeeded by the KIELER project, which is the context of this work presented in Chapter 4.

2.2 Automatic Layout

Automatic layout problems for arbitrary diagrams are often NP-complete, and diagram quality is difficult to measure [PCJ96, CP96, WPCM02, Pur02, Eic05, Völ08]. However, the graph drawing theory community emerged with sophisticated algorithms that solve single layout problems efficiently with appealing results. There exist open layout library projects with multiple sophisticated algorithms, such as the Open Graph Drawing Framework (OGDF) [CG07], Graphviz [GN00] and Zest, which is part of the Eclipse Graphical Editing Framework (GEF). There are also commercial tools such as yFiles (yWorks GmbH) and ILOG JViews [SV02]. Among the OGDF algorithms is a specialized algorithm for class diagrams [GJK$^+$03], which should not be drawn with plain layout algorithms because of the different notation standards for the edge types *association*, *generalization*, and *dependency*. Other approaches for layout of class diagrams have been proposed by Eiglsperger [Eig03], Seemann [See97], and Eichelberger [Eic05].

Spönemann et al. have especially considered actor-oriented data flow

languages with port contraints. As data flow diagrams can be structurally mapped to graphs, basic algorithms for layout of such diagrams can be taken from the area of graph drawing. There are several approaches to the general problem of graph drawing [TDB88, DETT94, DETT99, KW01, JM03], of which a selection is presented in the following.

Layered approach: The *layered* or *hierarchical* layout method first eliminates directed cycles in the graph, then determines a layering of vertices and optimizes this layering with respect to vertex positions. This approach is used for the KIELER Layout of Data Flow Diagrams (KLoDD) implementation, therefore it is covered with more detail in Section 6.2.2.

Force-directed approach: This approach creates a model of physical forces and minimizes the energy of the model [Ead84]. One variant consists in assigning springs with appropriate forces to each pair of adjacent vertices; such methods are called *spring embedders*.

As planarity of graphs is a topic which is well studied in graph theory, many drawing methods expect a planar embedding as input. If the graph to be drawn is not planar, it is first processed in a *planarization* phase. Methods which build on planarization are the following.

Topology-shape-metrics approach: This method computes a bend-minimal orthogonal representation of the graph in an *orthogonalization* phase, and determines final coordinates for vertices and bend points during *compaction* [Tam87, TDB88].

Visibility approach: A *visibility representation* is constructed, which maps vertices and edges to horizontal and vertical segments; these are in turn replaced by drawings of their corresponding elements [TT86, DETT99].

Augmentation approach: The graph is augmented by vertices, edges, or both, to get a graph with specific properties, e. g., one in which all faces have exactly three edges [Sch90, FPP90], or a biconnected graph [CON85]. In their basic variants, these algorithms usually yield a straight-line drawing.

Mixed model approach: This approach extends methods of straight-line drawing from the augmentation approach to construct orthogonal or quasi-orthogonal drawings [Kan96, GM98].

The main specialties that make layout of data flow diagrams more difficult than layout of general graphs are ports and hyperedges. Previous work on layout with port constraints includes that of Gansner et al. [GKNV93] and Sander [San94], who gave extensions of the hierarchical approach to consider attachment points of edges. These methods are mainly designed for the special case of displaying data structures and are not suited for the more general constraints of data flow diagrams. A more flexible approach is chosen in the commercial graph layout library yFiles (yWorks GmbH), which supports two models of port constraints and hyperedge routing for the hierarchical approach[1], but no details on the algorithm have been published [WEK01]. Other unpublished solutions to drawing with port constraints include ILOG JViews [SV02] and Tom Sawyer Visualization[2]. Handling of hyperedges in hierarchical layout has been covered by Eschbach et al. [EGB06] and Sander [San04]. Sugiyama et al. [SM91] and Sander [San96b] have shown how to draw general compound graphs, but due to the presence of external ports (see Section 6.2.2), our requirements for structural hierarchy are different.

The topic of visualization of hardware schematics is quite related to the drawing of data flow diagrams. While traditional approaches for the layout of schematic diagrams follow the general *place and route* technique from VLSI design [AKSM85, Lag98], more recent work includes some concepts from the area of graph drawing [Esc08]. However, these concepts are not sufficient for the needs of our application, since they do not address our scenarios for port constraints, but concentrate on partitioning and placement for large schematics and hyperedge routing.

[1] yFiles Developer's Guide, `http://www.yworks.com/`
[2] Tom Sawyer Software, `http://www.tomsawyer.com/`

2.3 Visual Comparison

We will consider automatic layout algorithms in different contexts. One of them is visual comparison of graphical models presented in Section 4.7.

Beginning with the diff tool [HM76], the comparison of content initially took place at the textual level. The first steps in comparing non-flat file data were taken in database applications, but those worked only on relational data. Chawathe et al. elaborated the ability to compare hierarchically structured information, satisfying a rising demand resulting from the immense growth of the amount of structured data in general [CGM97]. This method is also applied to documents written in the Extensible Markup Language (XML) by Ohst et al. [OWK03]. First implementations emanated from the XML content of model files and used the XML elements as a base for structural comparison. The representation was in a tree structure. Applications working with this paradigm are The Compare Utility (Spark Systems) or the XML Differencing tool (Stylus Studio).

Considering actual models, an interesting approach is used by CoObRa [SZN04]. The model elements themselves are considered as objects in a Version Control System (VCS). Every operation applied by the user to the model elements in the IDE is mapped to an operation on the object in the VCS. Only these change operations are saved, so this mechanism saves storage space and the difference computation between the versions is derived for free. A client-server concept is used to enable multiple developers to work on one project.

A generic approach in comparing and merging uses the similarity based SiDiff [SG08, TBWK07]. Input models are transformed to an internal data structure. The structure-based diff is then performed with these data, leading to a generic description of the differences. Depending on the type and semantics of the input models, the output must be interpreted in an appropriate manner to obtain the differences in the domain of the original model.

Both of the above concepts were implemented as plug-ins in the round trip engineering tool FUJABA[3], still with the lack of an appropriate graphical representation. A similar method is used by EMF Compare of Brun and

[3] http://www.fujaba.de

Toulmé [Bru08, Tou07], employed in the Eclipse Modeling Framework (EMF)[4]. EMF Compare is limited to display the changes as structured data in a tree-like view, and again, no graphical facilities are given. However, as further explained in Section 4.7, we can build on EMF Compare to *compute* differences, and enhance it with means to graphically *visualize* the differences.

Another work focusing on Eclipse is a plug-in suite of Mehra et al. [MGH05]. The input model is mapped to Java objects on which the comparison is performed in a generic way similar to EMF Compare and SiDiff. They also provide support to display the differences in a graphical way. However, there are some drawbacks, as the two versions are drawn into one diagram with a re-computation of the layout and ugly overlappings may occur.

The aforementioned SCADE Model Diff is designed to analyze differences between two SCADE models or two versions of a model. The differences are represented in terms of *added*, *deleted*, *changed*, or *moved* elements. Only the semantics are taken into account, no layout information. The results are presented in several ways, in a *diff tab* showing all the differences in a list, in a *diff window*, displaying two tree structures side by side, or in a *location window*, exhibiting two graphical models—SCADE models—with highlighted differences. Furthermore it is possible to generate a textual report of the changes. Unfortunately the tool is limited to the SCADE own data flow and behavioral languages, and it does not provide advanced features like automatic zooming/panning/folding.

Girschick presents an algorithm to detect and display changes in UML class diagrams [Gir06]. This algorithm is specific to UML class diagrams. Reports of changes are shown in an HTML page, including a graphical view of the merged diagram and a textual description of the changes. The view of the diagram is a merged version of both, which can lead to unaesthetic overlappings when much has been changed between the two versions.

Another example is the plug-in for Pounamu, a meta-CASE tool developed by Zhu et al., using a more generic approach that is not limited to one kind of model [ZGH+07]. The method to display changes is more

[4] http://www.eclipse.org/modeling/emf/

interactive and the user can, when checking out a newer version, see every change in the diagram immediately and accept or reject it. The graphical representation works with two layers on top of each other, one for each diagram.

Overall, there still appears to exist little work on how these differences are presented best to the user, especially when they are supposed to appear in the diagram itself. There are several works considering graphical languages, perception and representation, correlation between syntax and semantics, secondary notation, etc. [Pur02, PHG06]. However, a sound solution for a graphical comparison still appears to be lacking.

2.4 Execution Semantics

Some visualization techniques proposed in this thesis are used in the context of model execution resp. simulation.

There exists a range of modeling tools that also provide simulation for their domain models. To just mention some of the popular ones: *Ptolemy II* is a framework that supports heterogeneous modeling, simulation, and design of concurrent systems. For integrated simulation purposes Ptolemy provides the *Vergil* graphical editor. But there also exists the possibility to embed the execution of Ptolemy models into arbitrary Java applications as described by Eker et al. [EJL+03].

With Matlab/Simulink/Stateflow of Mathworks and SCADE of Esterel Technologies the user is able to integrate control flow and data flow model parts in their own Statecharts dialect and data flow language.

The Topcased project is based on the Eclipse framework and targets the model driven development with simulation as the key feature in validating models [CCG+08]. Other simulation supporting frameworks are Scilab/Scicos from INRIA, Hyperformix Workbench from Hyperformix, StateMate from I-Logix/Telelogic, or Uppaal from Uppsala University.

Most of these tools are specific and follow a clear semantics. This allows such tools to provide a tailored simulation engine that can execute the models according to this concrete semantics. Ptolemy supports heterogeneous modeling and different semantics for and within the same model. However,

Ptolemy has fixed concrete and abstract syntax and, hence, cannot be used directly to express arbitrary DSLs, where one reason to create them is to get a very specific language notation. Therefore, we investigated Ptolemy further as a generic semantic backend in combination with the Eclipse modeling projects as elaborated in Sec. 6.1.2.

As outlined by Scheidgen and Fischer [SF07], two fundamentally different concepts can be emphasized for specifying model semantics:

- Model-Transformation into a *semantic domain* (denotational) or
- provision of a new action language (operational).

In the first case semantics is applied to a metamodel by a simple mapping or a more complex transformation into a domain for which there already exists an explicit semantical meaning. This could be for example pure mathematical formulas. Another tool could be able to handle such domain and for example evaluate such formulas.

The second concept applies semantics by extending the metamodel with semantical attributes and operations on the same abstraction level. For this a meaning additionally has to be defined, e. g., in writing generic model simulators that interpret this information based on formal or informal specifications. For example a class of a *state* in a state machine metamodel usually only covers structure information, e. g., the name of the state, whether it is an initial or final state and what transitions it is connected to. To execute a state machine, states would require a notion of *being active*. Therefore one could augment the metamodel by a corresponding boolean attribute to the state class and operations that read and set that attribute during simulation time. The *M3Action* framework for defining operational semantics is illustrated by Eichler et al. [ESS06] or Scheidgen and Fischer [SF07]. Chen et al. present a compositional approach for specifying the model behavior[CSN07].

We follow the first approach by utilizing existing languages only and describe structure-based transformations as explained in more details elsewhere [Mot09].

Although defining a new high-level action language for transforming a *runtime model* during execution retains a stricter separation between the

different abstraction levels, the work presented here follows the more natural approach. That is, leveraging a semantic domain and specifying model transformations with necessary inter-abstraction-level mapping links to the model in question. This entails the following advantages:

1. There is no need to define any new language to express semantics on the meta model abstraction level.
2. There is a quite direct connection for meta model elements and their counterparts in the semantic domain which allows easy traceability.
3. Abstraction levels can be retained by carefully choosing an abstract semantic domain and advanced techniques for model transformation (e. g., a generative approach for the transformations as well).

Chapter 3

Taming Graphical Modeling

So far we have seen that the established means for modeling tool user interaction mainly focus on freehand editing and single static views on models.

This chapter presents an integrated practical approach to graphical modeling that consistently builds upon automatic layout. It comprises technological stepping stones that build upon each other, where each of them raises the abstraction level of user interaction with graphical models. We will examine different use cases of MDE that benefit from this approach.

First, the next section will introduce some terminology and will explain how this chapter is organized exploiting the Model-View-Controller (MVC) paradigm.

3.1 Pragmatics and Model-View-Controller

In linguistics the study of how the meaning of languages is constructed and understood is referred to as *semiotics*. It divides into the disciplines of syntax, semantics and pragmatics [Mor38]. These categories can be applied both to natural as well as artificial languages, e. g., for programming or modeling. In the context of artificial languages, *syntax* is determined by formal rules defining expressions of the language [Gur99] and *semantics* determines the meaning of syntactic constructs [HR04]. "Linguistic *pragmatics* can, very roughly and rather broadly, be described as *the science of language use*" [HM77]. This also holds for MDE with its artificial languages, as discussed in the following. However, first we clarify some more terminology specific to MDE mainly taken from the modeling linguists Atkinson and Kühne [AK03], Favre [Fav04, Fav05] and Gurr [Gur99].

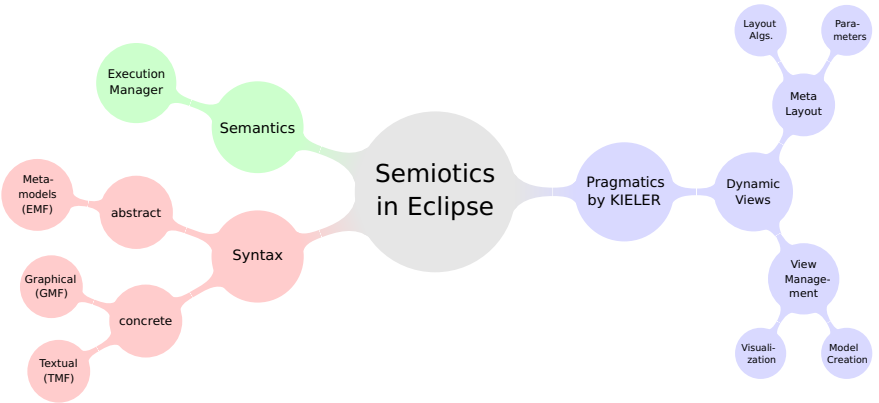


Figure 3.1. KIELER focuses on *pragmatics* and enhances the use of syntax and semantics of models which are defined by modeling platforms such as Eclipse

The main artifacts in MDE are *models* with two main concepts: A model *represents* some software artifact or real world domain and *conforms* to a *metamodel* or *grammar* [HM10], defining its *abstract syntax*. Additionally, the *concrete syntax* is the concrete rendering of the abstract concepts. Concrete syntax can be textual or displayed in a structured way, for example a tree view extracted from an XML representation of the abstract syntax. To be comprehensible, also a graphical syntax is very often used; the UML is one example.

According to Gurr, a *visual language* is any language that is expressed to the reader's visual sense. Therefore, diagrams as well as textual or mathematical models are visual languages. One special characteristic of diagrams is that they exhibit intrinsic properties, and these properties directly correspond to properties in the represented domain [Gur99], such as containment relations of boxes.

A *graphical model* is a model that can have a graphical representation, like a UML class model. A *view* onto the model is a concrete drawing of the model, sometimes also *diagram* or *notation model*, e. g., a UML class diagram. The abstract structure of the model leaving all graphical information behind

Figure 3.2. Different Representations of a Class Model: Diagram, Text and a Tree View (created with yUML (`http://yuml.me`) and Eclipse)

is the *semantical* or *domain model*, or just *model* in short. E. g., a class model can also be serialized as an XML tree. Hence, the *model* conforms to the abstract syntax, while the *view* conforms to the concrete syntax. Figure 3.2 shows three different views of the same class model.

A view can represent any subset of the model, which in some frameworks is used to break up complex models into multiple manageable views. Hence, there is no fixed one-to-one relationship between model and view.

State-of-the-practice approaches still have only little generic answers on how to specify *semantics* [SF07, MFvH09], but handle *syntax* of models very well, both abstract and concrete. They provide code generators to easily provide model implementations, syntax parsers and textual and graphical editors with common features like the Eclipse GMF[1].

Pragmatics

The third field of linguistics, *pragmatics*, traditionally refers to how elements of a language should be used, e. g., for what purposes a certain state-

[1] `http://www.eclipse.org/modeling/gmf/`

ment is eligible, or under what circumstances a level of hierarchy should be introduced in a model. It denotes the "relation of signs to (human) interpreters"[Mor38] and therefore Gurr calls it the "real semantics" of a language [Gur99]. Pragmatics addresses questions on how to avoid ambiguities or misleading information in graphical representation, for example by wrong usage of layout conventions, termed *secondary notation* by Petre and Green [GP92]. Hence, Petre as well as Gurr state:

> "A major conclusion of this collection of studies is that the correct use of pragmatic features, such as layout in graph-based notations, is a significant contributory factor to the effectiveness of these representations." [Gur99]

Following this conclusion, the traditional interpretation of pragmatics is slightly extended to all practical aspects of handling a model in its design process [FvH10a]. This includes practical design activities themselves such as editing and browsing of graphical models in order to construct, analyze and effectively communicate a model's meaning. We will especially discuss means that either advance or avoid such misleading or ambiguous representations.

Model-View-Controller

A non-trivial question at the onset was how to organize the subject matter. In the area of model-based design there exist extensive surveys, see for example Estefan's overview of model-based systems engineering methodologies [Est08], or the overview of hybrid system design given by Carloni et al. [CPPSV06]. An annotated bibliography by Prochnow et al. [PvH04] inspects the visualization of complex reactive systems. There exist numerous surveys on automatic graph drawing, which are considered an essential enabler for efficient modeling [DETT94, DETT99]. However, there is no existing taxonomy that focuses on the aspect of pragmatics.

Figure 3.3 shows a categorization of the subprojects of this work by the field of semiotics and further by the logical interrelationships, similar to Figure 3.1 for the Eclipse landscape in general.

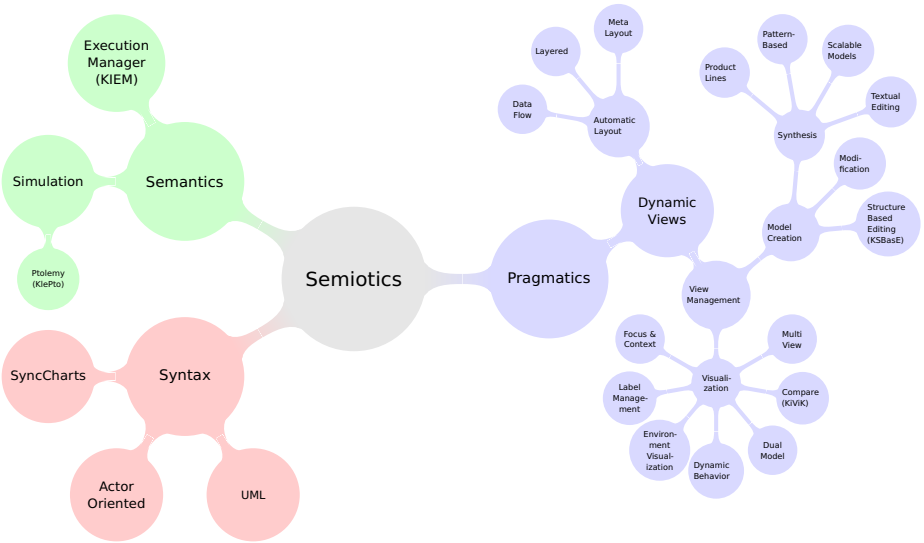
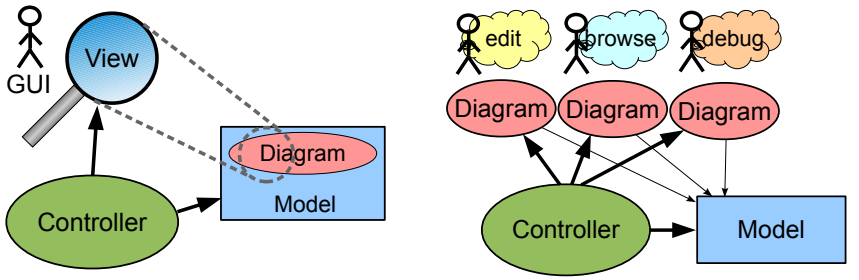


Figure 3.3. Categorization of the KIELER projects following semiotics.

However, there are already a number of paradigms well established in software engineering that could be adopted for model-based design processes, including the design of the modeling infrastructure itself. For example, the state of the practice in creating a graphical model, say, a data flow diagram or a Statechart, is to directly construct its visual representation with a Drag-and-Drop (DND) WYSIWYG editor, and henceforth rely on this one representation. Instead, we here propose to separate a model from its representation (view) and thus applying the Model-View-Controller (MVC) paradigm introduced by Reenskaug for Smalltalk [Ree79], defined as:

Models "Models represent *knowledge*. A model could be a single object (rather uninteresting), or it could be some structure of objects."

Views "A view is a (visual) representation of its model. It would ordinarily highlight certain attributes of the model and suppress others. It is thus acting as a *presentation filter*."



(a) Current practice for tool *developers*. Model: Internal data-structures including the diagram. View: GUI of the tool, showing a cutout of the diagram. Controller: Internal tool logic.

(b) Enable MVC for tool *users*. Model: Abstraction of the System under Development (SUD). Views: Multiple separated diagrams, possibly for different use cases. Controller: "Smart" modeling tool with logic to control the views.

Figure 3.4. Different ways of using the MVC pattern for MDE tools.

Controllers "A controller is the *link between a user and the system*. It provides the user with input by arranging for relevant views to present themselves in appropriate places on the screen."

This describes clearly the separation of knowledge, filtered views on it and the control by the user over these views. This pattern has been widely adopted in software engineering, e. g., for developing web-applications.

However, looking at different state-of-the-practice MDE tools (cf., Appendix A) shows that in this domain the clean separation in model, view and controller is not that clear—at most at an internal implementation level, i. e., in the scope of the tool *developer* as depicted in Figure 3.4a. Usually the (often one and only) diagram is buried in the internal data-structures for file handling etc. This can result in the problems presented in Section 1.2.

In contrast, the proposal here is to enable the MVC separation for the tool *user* as indicated in Figure 3.4b. A clean separation of a diagrammatic view on the model of the System under Development (SUD) requires a "smart" controller that manages the relationship between views and the model. The following sections will elaborate on what "smart" means.

Additionally, next to semiotics, also the MVC concept can be used as a

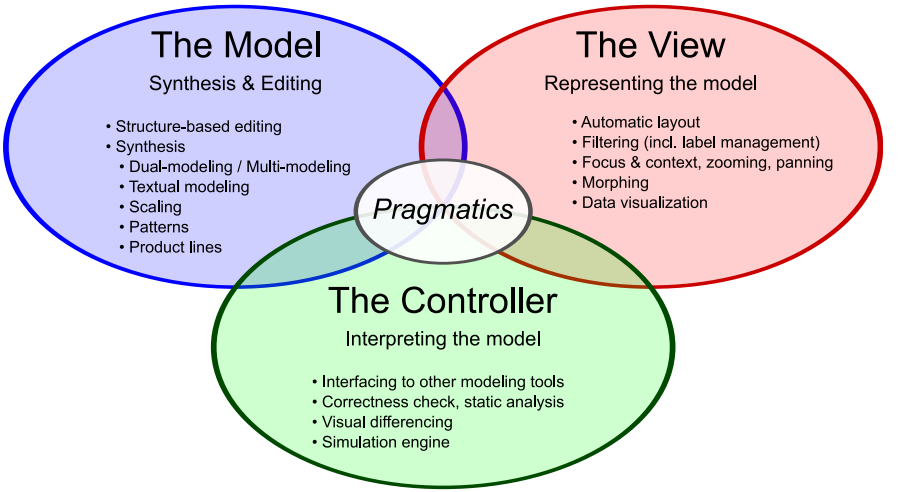


Figure 3.5. The MVC paradigm applied to the pragmatics of graphical model-based system design.

guiding principle to categorize pragmatic issues. For a first overview, see Fig. 3.5.

View Everything involved to get to a graphical representation of a model is categorized by the view. These are questions about the concrete layout of a diagram or the level of detail in which model elements are shown and how a view and any further information such as simulation data is presented to the user.

Model The model corresponds to all activities that lead to a semantic model, i. e., creation and modification of models. This includes standard editing mechanisms as well as the structure-based editing or model synthesis presented in the next sections. It also covers additional information that usually is not visible in the diagram notation that can be exploited by dual- or multi-modeling.

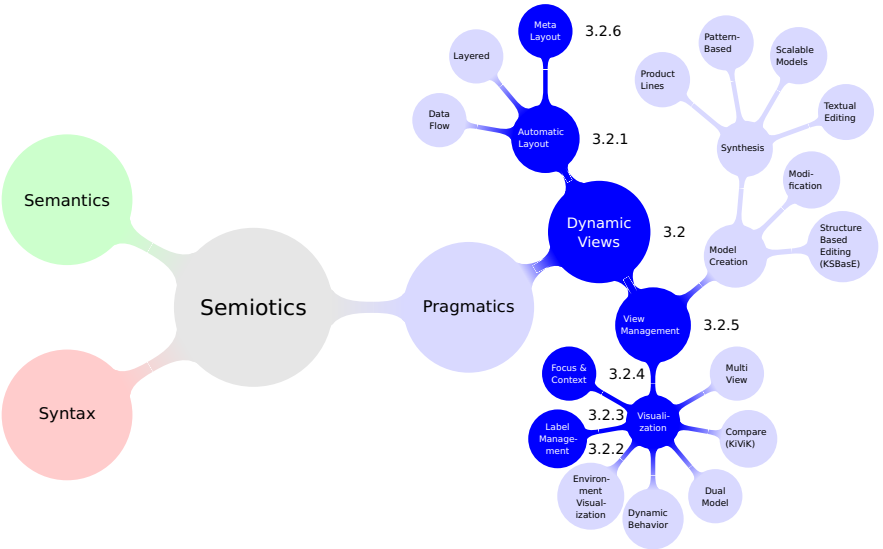
Controller The controller comprises semantic aspects. Anything that can be

used as input to configure a view, i. e., to reuse a diagram as graphical feedback. For example any kind of static or dynamic analysis such as correctness checks of a model could be used to present the results in the diagram itself.

In some cases, it may be arguable how a certain aspect should be classified. E. g., we here consider multi-view modeling to be part of the model, but it could also be classified as part of the controller. However, the MVC classification still appears helpful.

The following sections themselves are organized following this MVC categorization. However, to have an additional view onto the subject, we will also come across the categorization according to semiotics (hence applying multi-view modeling).

3.2 The View—Representing the Model



A key enabler for efficient model handling is the capability to automatically compute the layout of a graphical model. If one frees the user from the burden of manually setting the coordinates of nodes and bendpoints, sizes of boxes and positions of connection anchor points, this can open up enormous potentials. The following section explores this further.

3.2.1 Automatic Layout

> "The usefulness of the visualization of a graph is largely strongly dependent on its layout. In other words, a *good* drawing gives a high possibility of quickly and accurately communicating the meaning of a diagram, but a *bad* drawing gives a high possibility of confusion or misunderstanding." [Sug02]

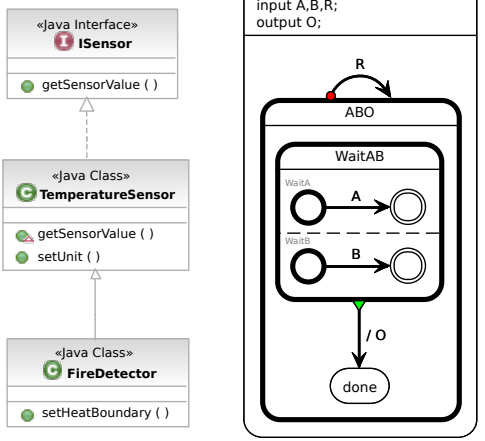
Getting a good drawing by a freehand editing process by hand is difficult and effort prone as discussed in Section 1.1.

Therefore I propose to calculate the layout automatically and thus free the user of the burden of manual placing activities. Automatic layout denotes the process of calculating and applying layout properties of a diagram automatically by an algorithm. By layout we understand all relevant properties that are required to create an actual drawing of a graphical model. In the following we will denote a layout algorithm also simply as *layouter*. Most diagram types we consider in this work are based on *graphs*, where main artifacts are nodes and edges. For example state machines can be mapped very well to graphs while message sequence charts cannot (cf. Figure 3.6c). Hence, using graph-based languages helps to reuse certain algorithms.

Automatic layout has to be appealing to the user such that he or she is willing to replace optimized manual layout with an automatically created one. Additionally this layout capability would have to be deeply integrated into the modeling tool and optimized for the respective graphical language syntaxes.

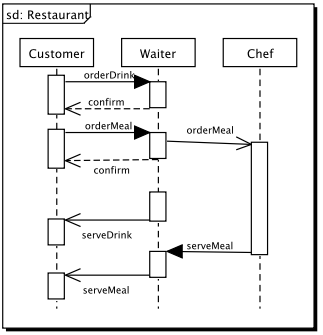
One must recognize that at this point, the automatic layouting capabilities offered by modeling tools, if they do offer any capabilities at all, tend to be not very satisfying. A major obstacle is the complexity and unclarity of this task. What are adequate aesthetic criteria for "appealing" diagrams [CP96, Eic05, PCJ96, WPCM02, Völ08]? Are there optimal solution algorithms or heuristics with acceptable results that adhere to the desired aesthetic criteria? An important aspect is the usage of *secondary notation*, which is specific to the modeling language used [GP96]. Used properly, an automatic layout does not only provide aesthetically pleasing diagrams, but can also give the viewer valuable cues on the structure of a model. For example, a standardized way of placing transition labels (e. g., "to the left in direction of flow") can solve the often difficult label/transition matching problem. Similarly a standardized direction of flow (e. g., "clock wise") can give a quick overview of the flow of information, without having to trace the direction of individual connections.

Figure 3.6 shows three examples of different graphical formalisms that pose different layout challenges. UML Class diagrams seemingly can be

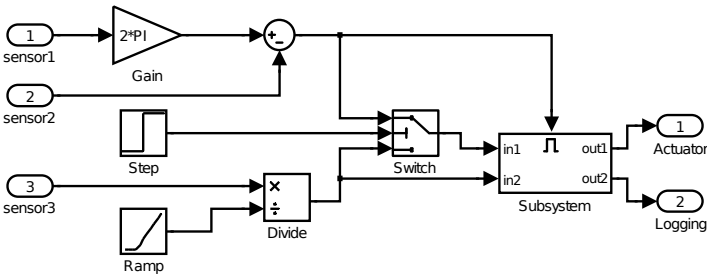


(a) UML Class diagram

(b) Statecharts (here Sync-Charts)



(c) Sequence diagrams



(d) Data flow diagrams (here Matlab/Simulink)

Figure 3.6. Different graphical syntaxes with different properties for their layout.

mapped directly to the standard graph layout problem, although sometimes hierarchy might be added by displaying packages in the diagram. While usual relations can be regarded as regular graph edges, any kind of inheritance relations as shown in Figure 3.6a have a special role. They are typically drawn from top to bottom, which is a strong constraint for the

layout algorithm. So even here one needs a specialized layout algorithm for this diagram type [GJK+03].

State machines go well together with graph layout, but introducing hierarchy requires special handling. In a diagram with hierarchy and without any inter-level connections crossing hierarchy boundaries, the layout algorithm for a flat layout can be called recursively, as elaborated in more technical detail in Section 4.2.

Small enhancements of the graphical syntax might have severe consequences for the layout. Inter-level transitions, which are possible in some Statechart dialects as UML State Machine diagrams or Stateflow of Matlab/Simulink, cannot be layed out with this approach and would require a special handling again.

Another special class are actor-oriented data flow languages [LNW03]. The notion *data flow* sometimes is used in different contexts resulting in different diagram syntaxes. We here consider languages also used in the control engineering domain such as Ptolemy, the SCADE, or Matlab/Simulink (Figure 3.6d). The connections denote flows of data and two distinct connections will likely carry different data and possibly different data types. Data are consumed by operators, and to distinguish the different incoming and outgoing data sources and sinks, an operator has special input and output *ports*. For many operators it is very important to specify explicitly which data flow is connected to which port because an alternation would also alternate the semantics. The example shows subtraction, division and switch operators which are not commutative and, hence, need their incoming flows exactly at the right input ports. The graphical representation also reflects this issue by presenting specific anchor points for the connections at the border of the operators. For the mentioned languages these ports have fixed positions relative to the operator, usually showing the data flow from left to right by positioning inputs left and outputs right. However, some special purpose ports may also be positioned on top or bottom of the operator, in general at pre-defined and static locations. These *port constraints* induce a great complexity to the problem and require special care such as by the approaches of Eiglsperger et al. [EFK00] or the modified Sugiyama layout as elaborated on in Section 5.4.

There are languages that are not even based on graphs, such as message

sequence diagrams as shown in Figure 3.6c. In this case we cannot reuse an existing graph-based layout algorithm, but have to create a customized one that respects the constraints given by the language. For example the lifelines are put next to each other and message connections are positioned between the lifelines one below the other. This reduces the freedoms of placing the elements and makes implementation of such algorithm not as complex as for standard graphs. However, such special constraints hamper reuse of existing algorithms. Hence, as the last consequence of this thesis, one should consider the ability to do automatic layout in language syntax design.

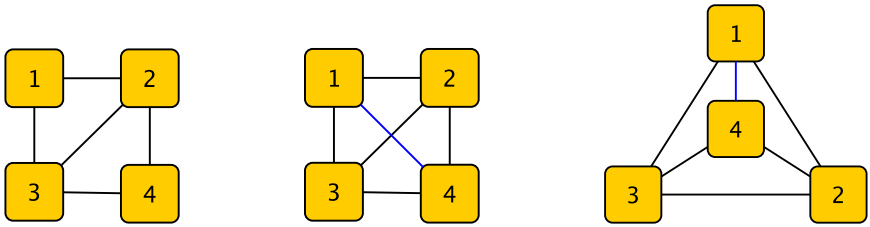
Summing this up, we cannot hope for one ultimate layout algorithm that is applicable for all languages and applications. Instead, we need a set of different layouters to cover a wide range of language syntaxes and layout styles.

Layout Stability

The approaches presented in the following sections will build upon automatic layout. We do not make any assumptions about the layout algorithm in charge. The resulting layout should be feasible for the given language. Concrete aesthetic criteria, however, might be subjective and we do not state any requirements for them. Therefore, the system should allow to parametrize existing layout algorithms and to plug-in new algorithms to the concrete languages' and use cases' needs.

There is only one categorization of layout algorithms that is important for this work and which we want to consider in the following: *static drawing methods* vs. *dynamic drawing methods*. Standard graph drawing techniques redraw diagrams from scratch whenever a new layout is required due to some change in a diagram. However, when working with diagrams interactively, the grade of *layout stability* introduced by Paulisch and Tichy [PT90] can help to preserve cognitive continuity and stability.

Figure 3.7 shows the effect of different layouts: introduction of new graphical elements can either cause a layout result that strictly adheres to the chosen aesthetic criteria, as the avoidance of edge crossings in Figure 3.7c, or it can try to reduce the changes of the layout as in Figure 3.7b.



(a) Original **(b)** Stable **(c)** Planar

Figure 3.7. Stability of layout when introducing a new edge (1,4)

Using dynamic graph drawing, sometimes also referred to as *incremental* or *interactive* [DETT99], can help to keep the *mental map* [ELMS91] of a model when recomputing the layout frequently.

Means to compensate this are mechanisms to apply the new layout result in a smooth animation such that the user sees the movement of nodes and edges. This obviously helps for rather small diagrams.

Therefore, the choice of a layout algorithm according to its stability level is rather important for the tasks proposed in the following. However, there exist only a few approaches on incremental layout updates like of Brandes and Wagner [BW97] and Branke [Bra01]. Still, also static layout algorithms have different levels of stability when they each get employed consistently from the beginning as shown in Figure 3.8.

Limits of Automatic Layout

There are limits of automatic layout of views when models become too complex. Consider for example the current UML2 Metamodel, which consists of 263 types with no compound structuring and thousands of relations and inheritances between the classes. This metamodel is available as an EMF Ecore model in the Eclipse Model Development Tools (MDT)[2], as semantical model augmented with some very small manually layouted views, but due

[2] `http://www.eclipse.org/modeling/mdt`

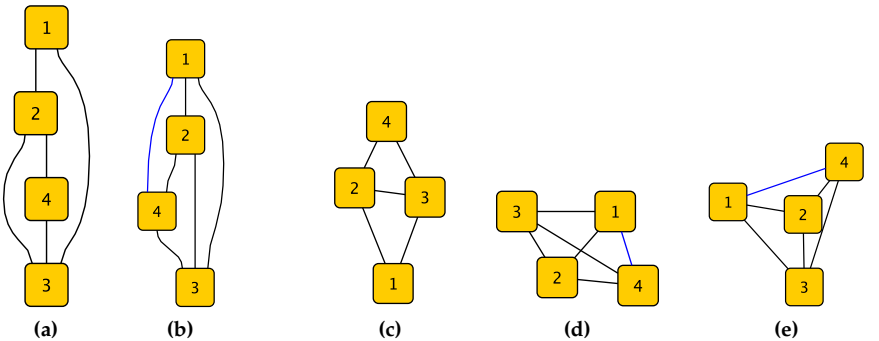


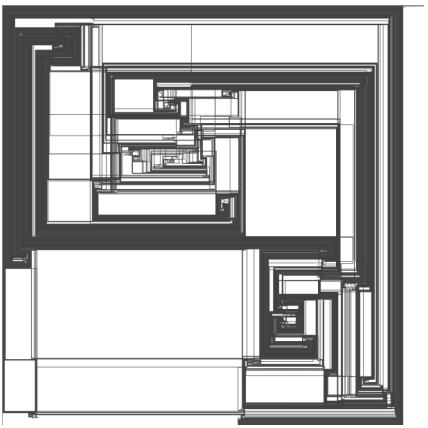
Figure 3.8. Different stability levels for different static layouters: The layer-based method (a and b) has rather good stability, while in general force directed approaches (c) are not deterministic (d and e)

to the model's complexity, there is no complete view available [Obj05]. With automatic layout, it is possible to synthesize such a view; Figure 3.9a shows the layout generated by the KIELER Infrastructure for Meta Layout (KIML) (see Section 4.2) using the Mixed-Upward-Planarization algorithm [GJK+03], which is optimized for class diagrams and respects the different types of edges. However, the result looks more like a VLSI integrated-circuit die and is hardly usable. Especially the numerous relations make the diagram unreadable. The algorithm focuses on the avoidance of edge crossings which produces big and long "highways" of edges. Standard navigation techniques like manual zooming and panning come to their limits; see also Figure 3.9b.

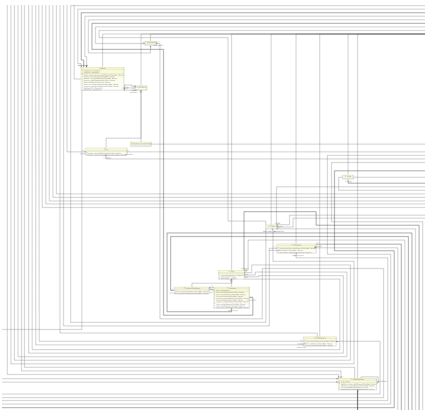
This limitation of plain layout application prompts the need for *view management* employing smart filtering mechanism, which is discussed next.

3.2.2 Filtering

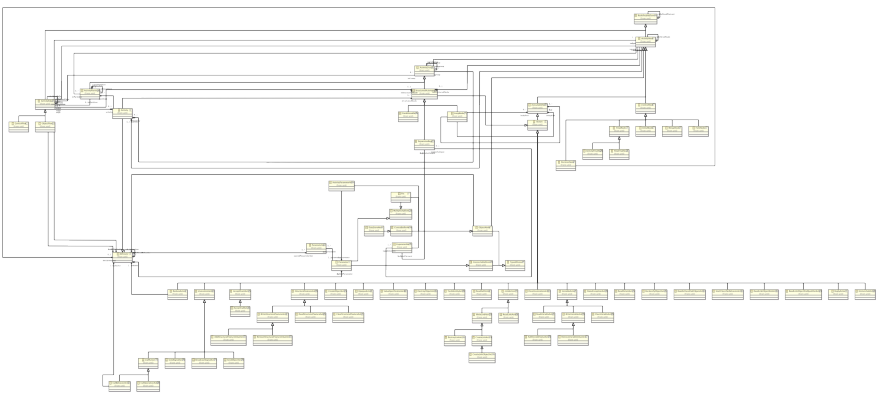
Card et al. define approaches for reducing information in a diagram: *Filtering*, *Selective Aggregation*, *Highlighting* and *Distortion* [CMS99]. A filter simply hides a set of objects in the diagram, to reduce the complexity. For



(a) Complete model.



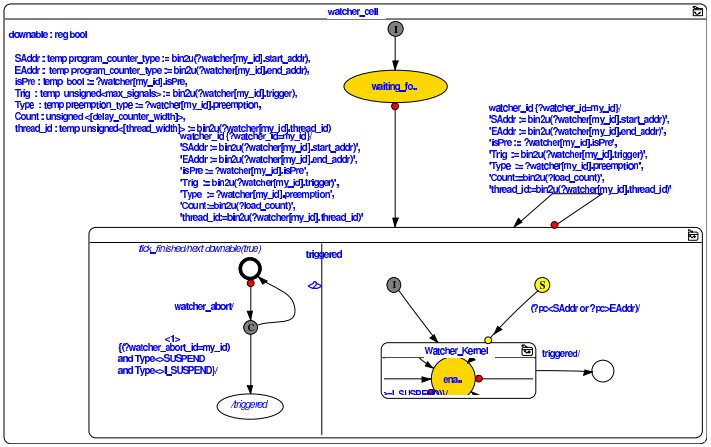
(b) Cutout with 20x zoom.



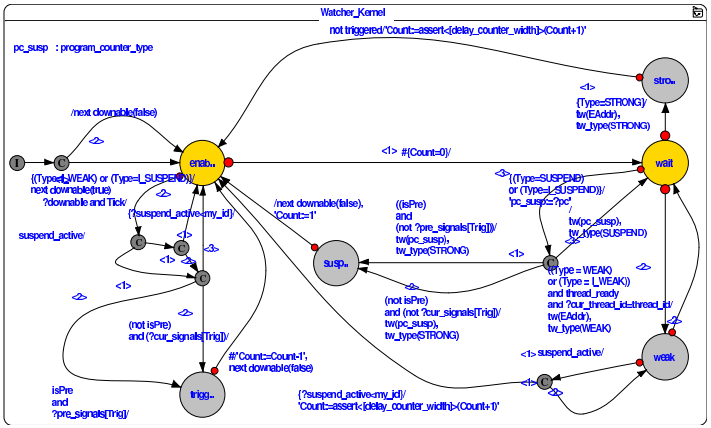
(c) A filtered view to the same model showing only Activity model parts.

Figure 3.9. Class diagram of the UML 2.1 metamodel in Eclipse. Standard navigation techniques come to their limits. Views become unusable. Filtering in view management can synthesize a feasible view.

technical scalability issues it is often not feasible to construct and inspect models with many objects—hundreds or thousands of nodes—but consis-



(a) Top-level state



(b) Inner state

Figure 3.10. Filtering in E-Studio, showing a part from a processor design [TT08]: The composite state in the lower right corner of (a) displays only a very small part of its inner life (b).

tently working with filters it can be. Only a small set of objects should be visible while all others are hidden and do not consume graphical system resources as well as cognitive effort of the user.

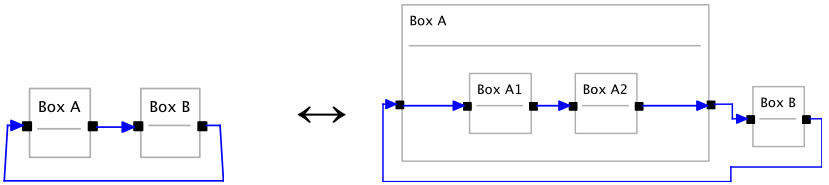
For example, to learn only about a smaller subset of the UML, e. g., Activity models, one may create a customized view on the UML metamodel that only contains elements immediately relevant to Activity models. Figure 3.9c shows such a view that is again automatically synthesized with automatic layout. However, this limited set of only 79 classes with much less edges presents a view that actually can be used very well to browse Activity models.

Filters could be applied by navigating through the model, thus a user reveals some parts and hides others. In order to work properly, we need strategies to apply this automatically to free the user of the burden to manually selecting the items to show and to hide. (This also leads to the focus and context paradigm, see subsection 3.2.4.)

Simple filters can already be found in some tools that hide objects on the canvas while the canvas size resp. the bounding box stays the same size. Hence, only the number of elements is reduced but not the size and therefore the same zoom level or paper size is required to display the model and there is hardly any chance to see more of the surrounding context as before. A rather unusual way of filtering can be used in Esterel Studio, see Figure 3.10. The hierarchy mechanism in E-Studio allows to create the relatively clearly arranged top-level diagram Figure 3.10a. However, the macrostate `Watcher Kernel` in the lower right reveals only a very small part of its contents, the rest is hidden. One has to manually open the `Watcher Kernel` state in a new canvas. Only then its whole extent can be seen as shown in Figure 3.10b. It reveals a complex inner life compared to what small part of it is shown in the parent. This feature becomes more useful in combination with automatic layout that uses the free space gained from the filters.

Dynamic Visible Hierarchy

Dynamic hierarchy is a special type of a filter where all children of some parent objects are filtered. For filters one might choose to hide items



(a) Composite `Box A` folded **(b)** Composite `Box A` unfolded

Figure 3.11. Example for dynamically visible hierarchy, here for an actor oriented data-flow language implemented in KIELER. This utilizes collapsible compartments and a layer-based automatic layout algorithm supporting port constraints.

regardless of the hierarchy level to reduce the complexity, see Fig. 3.11 for a simple example. This corresponds to the *folding features* of text or XML editors [LW99].

3.2.3 Label Management

Working with real-world applications quickly leads to the question of how to handle long labels. Label placement is a big issue in graph drawing [DKMT07] and in geography with map feature labeling [vDvKSW02]. The problem is computationally intensive, for bent edges it is NP-hard [KT97]. The labels are rather short in map labeling—city, street or river names—but in arbitrary DSLs they do not need to be, as Figure 3.10a shows. We assume to have an automatic layout algorithm that takes care of the label positioning, taking the label as is without changes. There are innovative approaches changing the diagram syntax, e. g., to replace an edge by the label itself by optical scaled down distortion [WMP+05]. However, we do not consider such invasive changes as universal option.

Instead, we try to dynamically reduce the complexity of the label to give the layouter better chances to find appealing layouts and to avoid difficulties as illustrated in Figure 3.12.

The simplest way would add a second label property that allows the user to add a custom abbreviation for a label. This could be any string, which then would be displayed instead of the real label. Some tools like Esterel

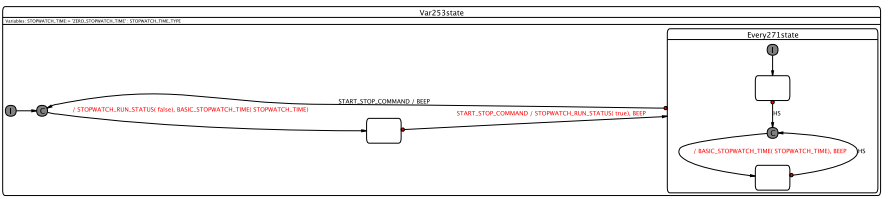


Figure 3.12. Long labels prevent good layout. Here a small part of Harel's wristwatch example [HLN+90], converted from Esterel to Statecharts [PTvH06].

Studio go this way. However, this means additional effort for the user who has to think of meaningful abbreviations. Additionally, inconsistencies can appear when the abbreviation does not correspond to the actual label.

An automatic *label filter* might use different strategies, see also Table 3.1. *Wrapping* aims to compact the label by wrapping the text, while *abbreviation* hides part of the text to actually shorten the length of the string. *Syntactically* arbitrary labels might be handled with suboptimal results. Even soft wrapping respecting identifiers will wrap compound labels inappropriately by not keeping related identifiers on one line. *Semantical* abbreviation could hide specific token types while showing only more important ones, like operators vs. variable/signal references. An example is shown in Table 3.1, where the original label contains complex boolean expressions over SyncCharts signals. A semantical abbreviation could remove all operators, such that the user can only see which signals are involved but not how they are connected in detail.

With a *label manager* in charge, the labels can be dynamically displayed with different levels of detail.

A smaller or more compact label will very likely reveal a better overall layout of the diagram. However, for some tasks it might be important to see a set of all labels in full detail while for other tasks a reduced detail level might be appropriate. So handling this issue dynamically would significantly improve the current situation.

Whether semantic or syntactic processing is done depends on whether the semantics for the diagram is available and is taken into account for the

Table 3.1. Ways to reduce label complexity temporarily, here for a Statechart transition label. The last case only shows which signals are involved but not how they are connected. This requires knowledge about language semantics.

Original			`(not SignalA) and (not SignalB) / SignalC(counter)`
Wrapped	syntactical	hard	`(not SignalA) and (n ot SignalB) / Signal C(counter)`
		soft	`(not SignalA) and ( not SignalB) / SignalC( counter)`
	semantical		`(not SignalA) and (not SignalB) / SignalC(counter)`
Abbreviated	syntactical		`(not SignalA) and (not Si...`
	semantical		`SignalA, SignalB / SignalC`

text reduction. If they are not specified, the label manager has no chance to correctly wrap the labels to meaningful results. A set of standard strategies could be provided to cover whitespace, braces or standard ways to handle assignments which are pretty common for text labels. But in general this will not provide good results. For abbreviation this is even worse, because the system can have only simple strategies for generic abbreviation, as taking only some prefix of the label or completely replace the whole label by a short unique ID.

If the semantics is known, the label manager needs to be customized—i. e., configured by some user-provided *label management schemes*—on how to work on the specific language. In Table 3.1 a label is wrapped only at operator boundaries and abbreviated by hiding all braces and operators and showing only a list of referenced identifiers. For other languages similar schemes could be added easily. The label manager framework should have an interface so these schemes could be user-provided.

3.2.4 Focus and Context

In classical modeling environments, the user typically has the alternatives of either seeing the whole model without any detail, or seeing just selected parts of the model. Figure 1.5, put earlier on page 19, from an avionics

application, shows what may happen if one does try to see the whole system. To find a way out of this dilemma, we note that when working with a model, it is common that there are parts of the model of particular interest for the current operation or analysis, which we refer to as the *focus*. Other objects next to it comprise the *context*, which might be important information to understand the focus objects but may be displayed with less detail. This leads to a *focus and context* approach where filters are employed to hide irrelevant objects [KM02, MJ03].

While *optical fisheye* views [LA94] result in deformation of the context, which makes it less comprehensible, automatic layout opens the door for *graphical fisheye* [SB92, LR96]. Here the objects in the focus are enlarged while the context objects are simply sized down while the automatic layout will still keep edges and boxes free of distortion.

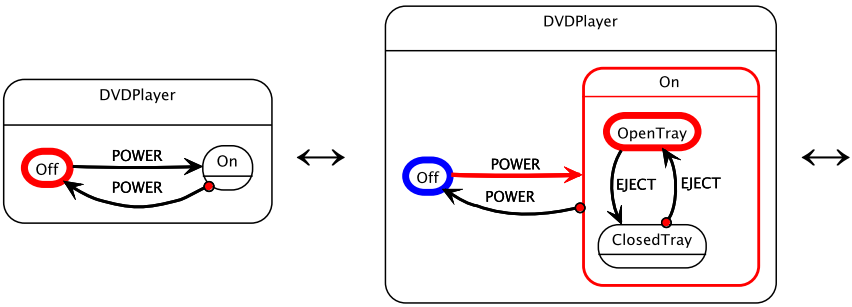
An example is shown in Figure 3.13 for SyncCharts, where the focus is naturally put onto the currently active states. A more detailed example is discussed in Section 4.4.

Alternatives for Focus and Context

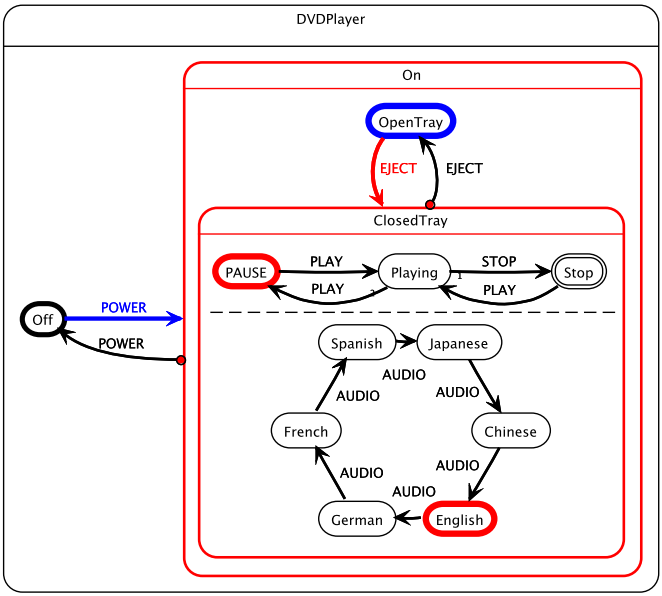
Experience showed that the specific interpretation of what objects form the focus and which the context is not always optimal. Sometimes it is difficult to follow the reasons of the view change, e. g., the switch from state `On` to `Off` in Figure 3.13. Signals emitted in the collapsed state can cause this change but are immediately hidden and, hence, the user cannot follow the causal event chain. This calls for a more general approach for applying focus and context techniques. Even for this specific diagram language one can come up with various other schemes to select the focus objects.

- One could show an intermediate step between the transition where the former active and the new active states are both in focus. (I. e., red *and* blue in Figure 3.13, which makes a difference when leaving a composite state).

- One might define the focus by active transitions instead of states—this could also filter parallel regions that do not change configuration.



(a) View with focus on state `Off` **(b)** View with focus on states `On/OpenTray`



(c) View with focus on states `On/ClosedTray`

Figure 3.13. Semantical graphical focus and context: Active states are in the focus (red) and shown with full detail. Inactive are the context (black), where composite states get collapsed. The states/transitions that were active in the last step are also highlighted (blue).

- The context does not necessarily go up to the top level, but might be limited to some number of hierarchy levels.
- *Meta focus:* one could specify a more abstract focus, e. g., "focus on signal *S*," which would set the concrete focus on transitions/states that reference *S*.

Concrete implementations are further discussed in Section 4.4. However, so far we see that a static hard-coded scheme for focus & context is not sufficient. The opportunity to customize the visibility of elements should be given to the user. This rises the need for *view management*, which is discussed next.

3.2.5 View Management

Setting the focus to active states in state machines is a rather natural choice. However, we have seen that this is not the only possible way to do it. Furthermore, considering diagram types other than Statecharts, it is not that obvious how to select the focus of the diagram, because there might be no such thing as an active state—e. g., in actor-oriented data flow diagrams sometimes all operators are active in every step—or there is no visible step-wise simulation at all—e. g., in structural diagrams such as UML class diagrams. To broaden applicability, it appears natural to upgrade layout information and directives to "first-class-citizens." This means that the view of a model becomes part of the state of a model, which can be controlled by the user, the modeling tool, or the model itself. An engine for *view management* could for example categorize graphical entities in focus and context, maybe even multiple levels of context by setting different *levels of detail* as denoted by Musial and Jacobs for the UML [MJ03]. These and other aspects of view management are depicted in Fig. 3.14.

The view manager needs to listen to *triggers*, or *events*, upon which it might change between the dynamic views, showing the user some objects in the focus and others in the context. These triggers might be *user triggers*, induced manually by the user, e. g., manually clicking on fold/unfold buttons at parent nodes or manually changing the focus by selecting a different node. They could also be *system triggers*, produced by the machine

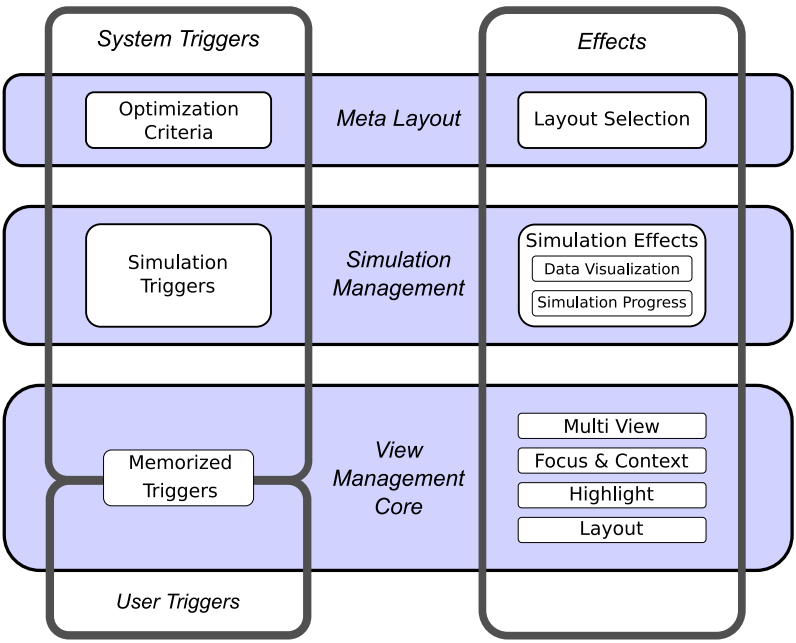


Figure 3.14. Aspects of view management.

by some automatic analysis, semantical information, progress of time (real or logical), etc. *Memorized triggers* can for example be trigger annotations stored persistently with a model.

Obviously, this view manager can hardly be one monolithic application that carries all information and is applicable for all types of DSLs and application environments. We need a way to efficiently specify both the *triggers* to listen to and the *effects* that shall be performed. We denote this specification the *View Management Scheme* (VMS) or *combination*. This VMS needs to be provided by the developer, either by the application developer for application-specific schemes or by the tool creator for more general schemes applicable for a whole DSL. For a practical user interface this VMS should be expressed by a simple syntax, maybe close to some general

purpose scripting language. It would require expressions to

1. address different user triggers (mouse clicking, keyboard events),
2. specify custom system triggers,
3. address different system triggers,
4. address different visualization effects (folding, unfolding, filtering, layout triggering, choose layout algorithms), and
5. address graphical diagram objects or their properties, either specific objects (e. g., "State A") or classes of objects (e. g., "a node of type *state*") or specific *patterns* of such objects.

An example for an advanced VMS is given in pseudocode in Listing 3.1. It specifies a more differentiating focus & context scheme that also takes the recent activity history into account. Only active states are shown in full detail, while recently active states are also shown and states that haven't been active for some time get shown only with reduced detail or are even completely hidden.

A view management language would be able to execute such scheme, where all `Effect` procedures were primitives for this language, implemented in the underlying host language.

Some of the items can be implemented using standard techniques, such as addressing model elements. A set of predefined user triggers and visualization effects could be provided. It is not that obvious how to specify custom system triggers. Most of them will be very semantic-specific for a certain DSL. For example the trigger "a state has become active" (i. e., the `activeStateChangeTrigger` in Listing 3.1) in a Statechart would require interaction with the simulation engine and, hence cannot be implemented only with the knowledge about the certain DSL meta-model and the modeling and visualization framework. Therefore an interface to the "outside" is required, the respective lower level programing environment of the modeling tool.

Such a view management engine could be employed to handle the ideas of semantical focus and context in a general way. It should also allow, via

Listing 3.1. Example of a View Management Scheme

```
viewManagementScheme "active State Focus&Context"
    triggered by activeStateChangeTrigger
        foreach active state and its ancestors
            setLevelOfDetail(state,FULL)
            highlightEffect state red
        foreach not affected state
            switch(number of steps ago it was last active)
                case 1:  setLevelOfDetail(state,FULL)
                         highlightEffect state lightBlue
                case 2:  setLevelOfDetail(state,MEDIUM)
                         highlightEffect state blue
                case 3:  setLevelOfDetail(state,LOW)
                         remove highlightEffect
                default: setLevelOfDetail(state,LOWEST)
                         remove highlightEffect
        autoLayoutEffect and zoomToFitEffect

// functions that can be reused in other VMS
setLevelOfDetail(state,level)
    switch(level)
        case FULL:   filterEffect expand composite state
                     filterEffect show all labels
                     foreach connected transition
                         setLevelOfDetail(transition,FULL)
        case MEDIUM: filterEffect collapse composite state
                     filterEffect show all labels
                     foreach connected transition
                         setLevelOfDetail(transition,MEDIUM)
        case LOW:    filterEffect collapse composite state
                     filterEffect hide all labels
                     foreach connected transition
                         setLevelOfDetail(transition,LOW)
        default:     filterEffect hide state

setLevelOfDetail(transition,level)
   switch(level)
      case FULL:   labelManagerEffect full label
      case MEDIUM: labelManagerEffect abbreviated label
      default:     filterEffect hide label
```

user triggers, to quickly navigate manually through a model, using for example *semantic zooming and panning* where one considers the structure of a model to navigate through it. For example, one would not just change the zoom on a linear percentage scale as is commonly the case, but could also change the zoom by hierarchy level.

This far we have used view management to specify *what* can be seen in a diagram *in what detail*. As yet we take automatic layout as one straight operation which we cannot influence. Hence, we cannot control *how* to display the graphical elements. This will be discussed in the following under the notion of *meta layout*.

3.2.6 Meta Layout

For a given graphical DSL there might be different layouts conceivable for the graphical representation. There may be different automatic layout schemes available, either one single algorithm with different parametrization options, or completely different layout algorithms. Each layout algorithm results in a different layout style. The process of selecting and combining different already existent layout algorithms is denoted as *meta layout*. This should be integrated into the view management.

Note that this is somewhat contradictory with the concept of having a *normal form* [PvH06], where models with the same domain model will have the same graphical representation. The motivation for normalization is to limit ambiguity and subjectiveness when creating or analyzing diagrams. However, it may be hard to find one layout algorithm that provides optimal layout results for all possible applications—even within one DSL. So we may soften the idea of normalization by varying degrees. One could apply different layouters

1. to different models,
2. within one model, in different unconnected hierarchy levels (see Figure 3.15),
3. within one model, in different unconnected areas of the same hierarchy level and

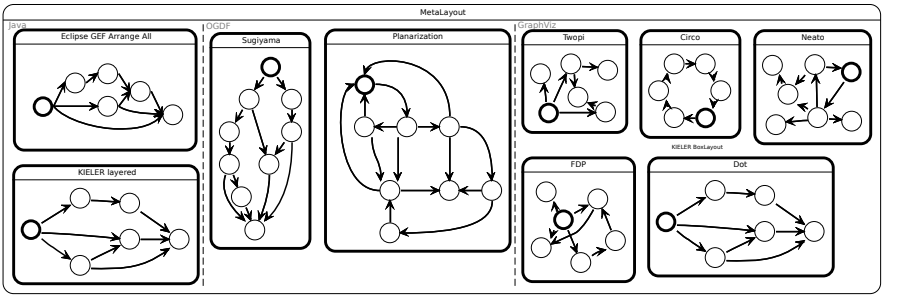


Figure 3.15. Meta layout in KIELER: Employ different layout algorithms in one diagram.

4. within one model, in different connected areas and/or hierarchy levels.

Problem of certain Layout Scopes

Conceptually it is straight forward to apply different layout algorithms to cases 1, 2 and 3 above. Figure 3.15 shows an example where different hierarchical elements themselves are arranged using one layouter, while each of their contents is arranged by other layouters. Therefore the inner layout results are independent of siblings. The outer layout only depends on the resulting bounding boxes of the inner layouts. This can be handled by calling the layout algorithms depth first in the containment tree.

Case 4 above is more challenging. In the case of connected hierarchy levels, this phenomenon is also called *inter-level* edges. Even on plain graphs this can happen if the user has different sets or *clusters* of nodes and there exist connections between nodes of different clusters. The main approach on clusters or compound nodes is to use one dedicated layout algorithm that arranges all connected areas, instead of dividing the graph and conquering it with multiple different algorithms as done above. A survey on this topic is given by Kaufmann and Wagner [KW01]. So far, the approaches do not yield optimal results and are only available in very few layout libraries. GraphViz dot, for example, supports clustered graph layout, but still no

real compound graphs. Furthermore, the free allocation of different layout styles is constrained.

Therefore, languages containing inter-level edges make the problem much harder and limit the freedoms of different layout styles in one diagram. The user resp. toolsmith has to provide sufficient layout algorithms that support these complicating features.

Layouter Choosing Strategies

Having multiple layouters and different regions in the diagram, a question arises: When to apply where what layout? This is answered by *layout choosing strategies*.

The simplest strategy is plain user decision. The user manually annotates each part of the model with the specification of which layouter should be used. This way the user would be able to select the best layouters according to his or her personal subjective aesthetic criteria. Additionally, the user could consider application and system specific properties when choosing the layouters.

For a better benefit, the modeling tool could assist in choosing the right layouter settings by trying to optimize the layout result. The optimization criteria should be provided by the user while the machine should be able to work with them. Possible criteria are

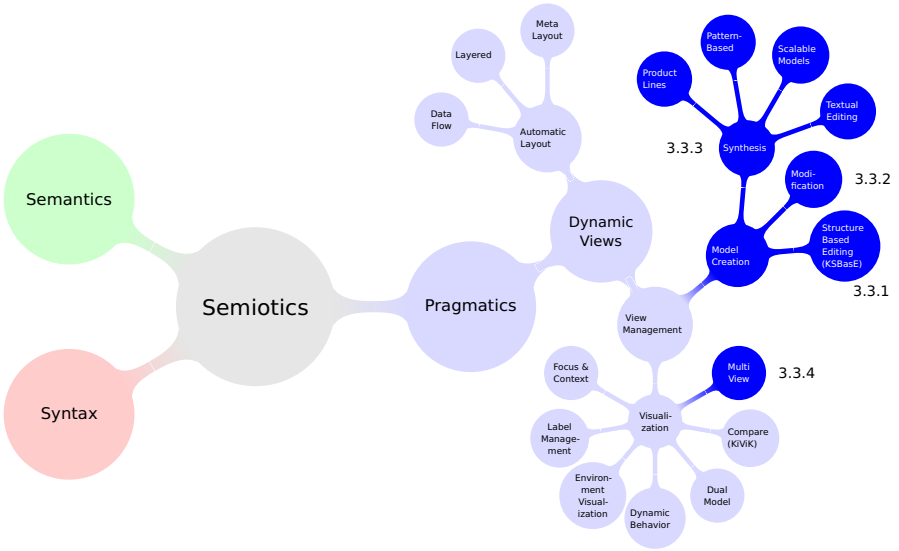
- *syntactic aesthetic criteria* such as link crossings, link lengths, diagram area, aspect ratio [KMS94, Pur97];
- *semantic aesthetic criteria* such as alignment, symmetry or zoning [Pur02];
- prescribed *development patterns*; or
- *model element types*, e. g., graph-based vs. port-based (cf.subsection 6.2.2) or plain graphs vs. compound graphs as discussed above.

While some layout algorithms might be implemented with the explicit aim to optimize one or multiple criteria, others are not. Hence, for a given criterion and two different diagrams, also two different layouters could reveal the optimal results. Therefore the modeling environment needs built-in *metrics* to measure the adherence to all of the optimization criteria. A first

brute-force approach would simply try all available layouters and choose the one with the best results according to the metrics. Another layout choosing strategy would select the layouts on the context in which the diagram is used. Maybe there is a different layout sufficient for simulation than for editing and even another one for printouts. Furthermore, within a simulation the engine can have different states that require different layouts.

So far we discussed how the *view* of a model can be presented. In the next section we outline how to handle the *model*, i. e., how to create or maintain models. These more abstract or more user-centric features are still based on the above concepts of automatic layout and view management.

3.3 The Model—Synthesis and Editing



A graphical model is nice to look at, but can be effort-prone to create or change. Common editors have the paradigm of What-You-See-Is-What-You-Get (WYSIWYG) Drag-and-Drop (DND) or *freehand* interaction as discussed in Section 1.1. In general it is desirable to immediately get visualized effects of editing steps in WYSIWYG. However, the way of interaction—DND—is the source of plenty of additional manual editing efforts.

I here advocate to try to avoid the tedium induced by DND editing as much as possible, to put the system back into the focus instead of its graphical representation. The basic enabler is the aforementioned capability for automatic layout (Sec. 3.2.1). Here again, one issue is the preservation of the mental map of the modeler. In the context of model editing, there exist different schools of thought. One direction argues that the appearance of a model after an edit should be changed only minimally, to preserve

the mental map [CMT02]. The other approach is to try to give models a uniform appearance, that "the same should look the same," proposing a *normal form* that is independent of the modeler and the history of the model (see also Sec. 3.2.6). There the issue of mental map preservation is addressed during the editing step by a morphing animation of the model.

Creating or maintaining a model usually involves many successive small editing steps. Therefore employing layouters with a good *stability* (cf. 3.2.6) can help to preserve the mental map. However, running a layouter from scratch might result in a more optimzed layout in the end, independent from the editing history. Therefore, the choice of layout algorithms should still be in the hand of the developer or development team—the corresponding philosophy could be prescribed by development guidelines provided by a company.

3.3.1 Structure-Based Editing

The idea of structure-based editing comprises only structural decisions of the developer, which are (1) to select a position in the model topology and (2) to select an operation to apply to the model. This changes only the structure of the model, i. e., its topology, sometimes also referred to as the *domain* or *semantic* model.

The graphical representation also can be updated immediately. The automatic layout has to be applied to create a fresh *view* of the new structure of the model after the user operation. The complexity of the model and the performance of the layout algorithms determine whether it is feasible to apply the layouter after every small editing step in order to get immediate visual feedback. Therefore we eliminate the DND style editing but possibly keep the WYSIWYG nature of the editor. This immediate visual feedback appears valuable enough to put a premium on fast layouting algorithms, even if this might give slightly sub-optimal results.

Structure-Based Editing for Graph-Based Models

For DSLs that are based on graphs some experience from the Kiel Integrated Environment for Layout (KIEL) project was gained, which applies this

paradigm to Statecharts [PvH07]. Graphically they consist of states (nodes), transitions (edges), hierarchy and parallel regions. In this case only a small set of different structural operations are required to create or modify the charts. For a selected state these are only

1. *create a new following state* and
2. *add a region to the state*, as shown in Figure 3.16

For transitions the operations are only

1. *create* a transition and
2. *reverse* a transition.

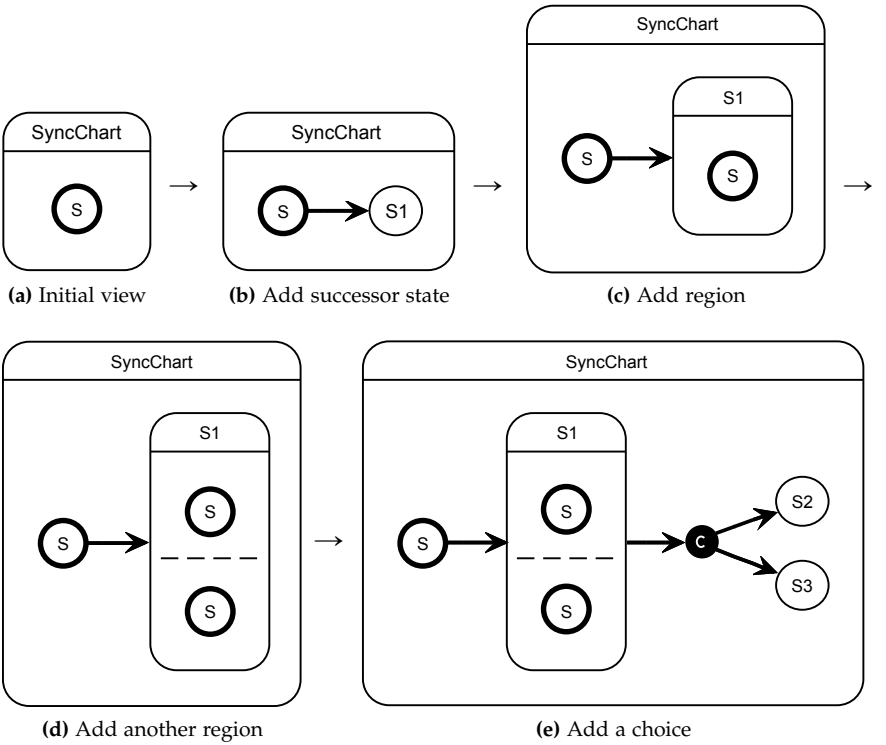
Some other "syntactic sugar" can be provided, e. g., adding a choice construct, but nevertheless the operation set is relatively small. Other changes to the model are done afterwards, e. g., changes of labels by filling out form fields.

This paradigm would also apply for other graph-based DSLs because the set of affected model elements in every step is small—up to two. For node operations one node needs to be selected, for edges there are two nodes, source and target.

The operations can be hard-coded for each language or language class. Additionally, the paradigm can be used in conjunction with model transformation frameworks. Especially *in-place transformations* change the underlying domain model by pattern matching where source and target meta-model are the same. Hence, the original model is only changed instead of transformed into another DSL. Therefore an in-place transformation framework such as from Taentzer et al. [BEK+06] can be used to specify the transformations while the view management with automatic layout adds the graphical feedback to get the full WYSIWYG experience.

Structure-Based Editing for Port-Based Models

For *actor-oriented data flow* models with *ports* (cf. Sec. 6.2.2) the editing task is a bit more complex. Especially adding new nodes requires more specification than a simple operation like "add a successor node" can



(a) Initial view **(b)** Add successor state **(c)** Add region

(d) Add another region **(e)** Add a choice

Figure 3.16. Example for structure-based editing of a Statechart.

provide. In a graph-based model this operation will generally transform one valid model to another valid model, because it can add a new state and simply connect old and new state with exactly one new edge. Port-based models have stricter connection requirements. In general there is an arbitrary set of different kinds of operator nodes; usually this node library is also extensible by the user. Each node has a certain *interface*, i. e., the set of input and output ports that specifies how the node must be connected to other nodes. Hence, a new node in the model likely requires not only one but multiple connections. These have to be specified not only between the

nodes but between specific ports. There are different possible ways for the user interface in this case.

In the first approach the goal is still to provide the diagram itself as the user interface. To support incremental editing, the operation to be performed can be divided in small incremental steps where each does not necessarily lead to a valid data flow model because it might be not sufficiently connected. After every step the view manager can update the layout and some meaningful graphical representation of the intermediate step is created. An example sequence of such operations is shown in Figure 3.17.

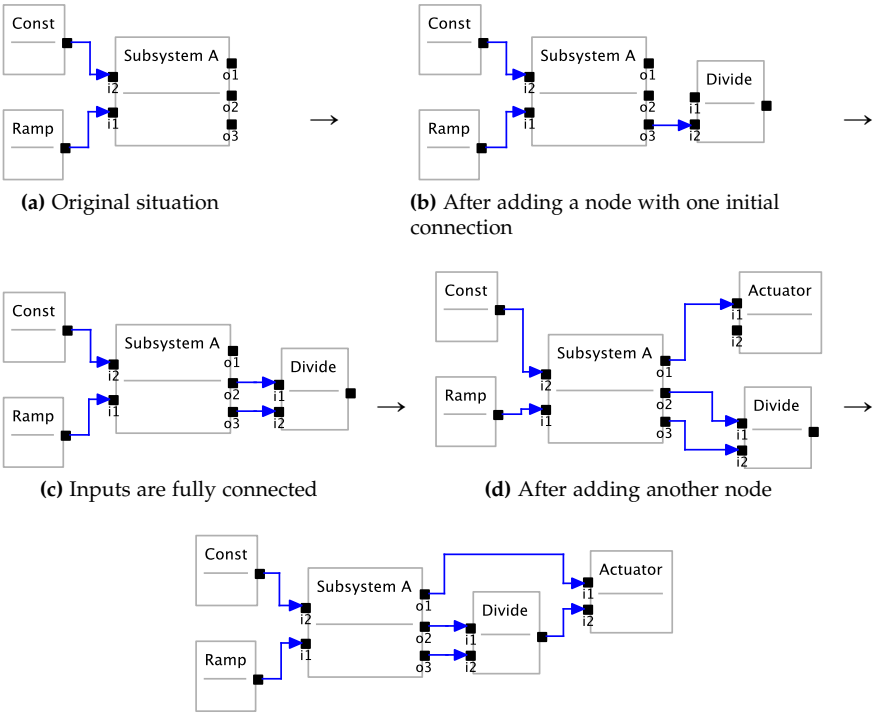
In this scenario, the set of operations to connect ports determines the efficiency of creating or editing models. Shortcut operations to connect multiple ports can help to reduce the manual steps. For example the SCADE editor provides the operations *connect by rank* and *connect by name* which will interconnect all inputs of one with the outputs of another selected node either by name of the ports or successively by their rank. In SCADE this is not post-processed with the view management, but this can give a first inspiration for the type of connection operations that are helpful.

3.3.2 Modification

For all possibilities of model changes, the set of model operations must be augmented by operations for removing nodes and connections. Additionally a set of syntactic sugar operations should be provided to manipulate the models efficiently, e. g.,

- replace a node by another node of another type,
- replace a connection by a different connection type,
- redirect a connection, or
- insert a new node into one or multiple connections if the port rank fits—i. e., break up the connection into two parts, insert the new node and connect the input and output to the connection endpoints.

This can reduce manual steps especially by keeping attributes of the objects that were manually set after the object creation.



(a) Original situation

(b) After adding a node with one initial connection

(c) Inputs are fully connected

(d) After adding another node

(e) Connections complete

Figure 3.17. Possible structure-based editing steps in a port-based language.

Copy and Paste

One common established use-case during editing is performing *copy* and *paste*. This is also often used in graphical modeling. However, the enabling steps in a usual freehand editing environment are even more severe than for primitive editing operations. The user would have to

1. select all objects to copy,
2. call the copy operation,

3. choose a target space,
4. make free space at the target location big enough for the copied objects in their current layout,
5. select the target place (however, selecting an empty location usually is not possible in most tools),
6. call the paste operation,
7. move the whole pasted set of objects to the new empty space (as placing them initially at the desired target location is usually not possible) and
8. rearrange the surroundings such that the new objects seamlessly integrate.

Especially steps 4, 7 and 8 may be arbitrary effort-prone, and step 7 may be frustrating when the pasted objects do not appear at the target space of step 3 and the tool does not state explicitly about its target space policy.

However, structure-based editing employing automatic layout can improve the situation considerably. The editing steps would boil down to

1. select all objects to copy,
2. call the copy operation,
3. select a target *object*, and
4. call the paste operation.

With automatic layout, the user should not specify any target *location*, but only a target *object* where the contents should be pasted. A generic transformation description should then specify how the elements are pasted *into* the target object and the automatic layout would do the rest. An example for SyncCharts is given in Figure 3.18 and the corresponding rules are listed in Table 3.2. Each transformation rule has to consider the *copy sources* (labeled "S" in Figure 3.18), i. e., the selected elements which get copied, and the *copy targets* ("T"). For SyncCharts these objects may be States, Regions, and Transitions, and each set may be of arbitrary size.

A good example is "copy multiple states to one transition". In a usual freehand editor, this is not possible and would do nothing. As implemented in KIELER (see Chapter 4), the transformation

1. cuts the target transition into two transitions,

2. adds a new State in-between both transitions, and

3. adds the selected nodes into a new Region of the new State.

Other similar transformations are possible, which the toolsmith would have to define according to experience in the context of the given DSL. Selecting multiple target objects is a fast way to replicate objects multiple times.

As a word of caution, these copy&paste effects go considerably beyond what designers are familiar with today. Also, some of these effects are probably needed only rarely, such as the "copy transitions to transitions". Still, extending the copy&paste paradigm in this fashion may significantly increase productivity, and is yet another example of the possibilities for harnessing automatic layout.

Error Handling

Proper *error handling* influences the efficiency of the modeling process, especially for beginners and intermediates, where small modeling errors are quite common. We should learn from best practices in textual programming IDEs and try to adopt features to graphical modeling. For example the *Quick Fix* feature of Eclipse allows beginners to learn textual programming—e. g., Java—in an interactive tutorial-like way. Errors are displayed immediately with the help of incremental continuous compiling. Additionally the UI presents a list of possible solution operations which can be triggered by the user.

Features like this can be incorporated into graphical modeling by orchestration of different building blocks. There are generic modeling frameworks that support model validation such as the EMF with its Validation Framework[3]. Hence, the user consequently can get feedback about the model

[3] `http://www.eclipse.org/modeling/emf/`

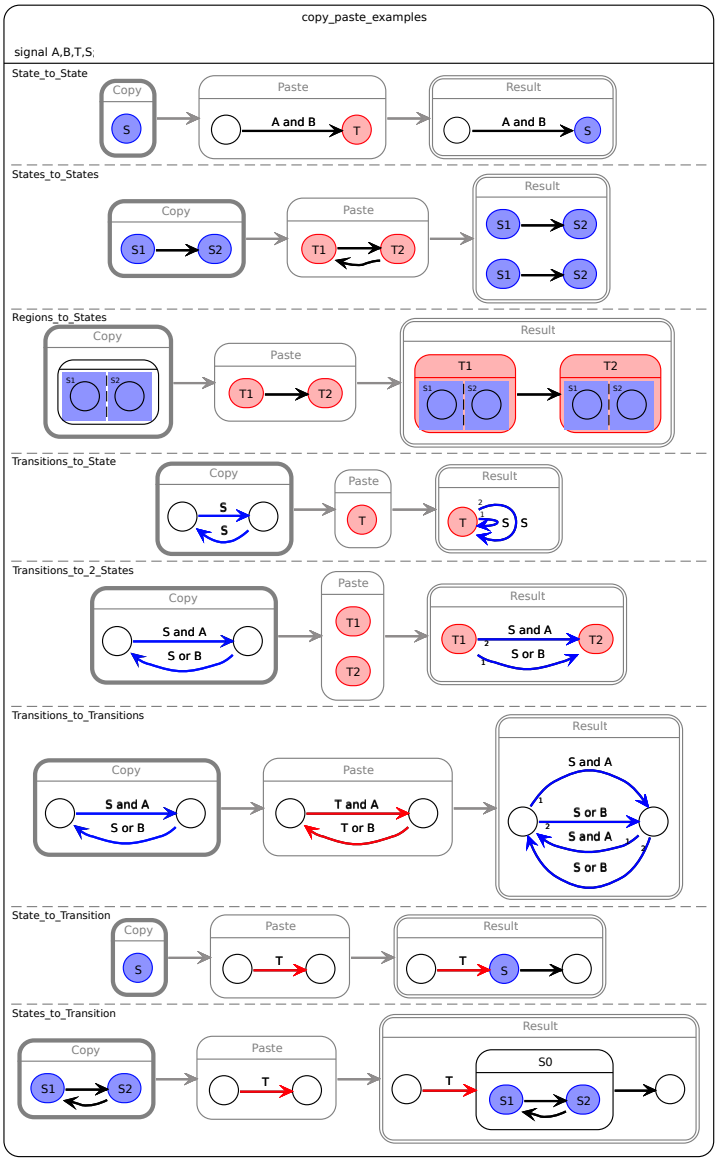


Figure 3.18. Different Copy&Paste sources and targets in a SyncCharts diagram.

Table 3.2. Copy & Paste Overview for SyncCharts

Source/Target	State	Region	Transition
State	replace, keep all transitions	add State to Region	insert, old Transition sticks with source
Region	insert	replace, if root: create state and insert	insert, old Transition sticks with source
Transition	add as selfloop	create dummy States and connect	replace
States	replace, discard transitions	add States to Region	insert, old Transition sticks with source
Regions	insert	replace, if root: create state and insert	insert, old Transition stick with source
Transitions	add as selfloops	create dummy States and connect	replace with all

Source/Target	States	Regions	Transitions
State	replace each	insert into each	insert into each
Region	insert into each	replace each, if root: do nothing	insert into each
Transition	connect 2 states	add two dummy states with transition in each	replace each
States	replace each with all	insert all into each	insert into each
Regions	insert all into each	replace each with all	insert into each
Transitions	connect 2 states with all, selection order for direction and priorities	add two dummy states with all transition in each	replace each with all

consistency. For specific DSLs there should be a set of standard error cases provided together with a set of possible solution operations, again supported by automatic layout of the created solution model.

3.3.3 Synthesis

With an automatic layout capability, it is not only possible to change models interactively with the developer. One can also synthesize completely new graphical models, including the domain model and its graphical representation. There are multiple scenarios where this *model synthesis* can be of significant benefits and lead to innovative modeling environments.

Textual Modeling

An alternative to the graphical representation of a model still is text. Having information in a textual representation can have many advantages [Gur99, BWGP01, PTvH06]:

- Handling text is well known and efficient in CS. Textual editors are far developed to the ergonomics of human interaction and are mature. Developers are used to handle text and are likely to have trained typing skills.
- Text is only one-dimensional and might be more comprehensible for some problems. Text can be well processed automatically, there are generic parser and generator frameworks available. There also exist efficient tools for all kinds of specific tasks, from versioning via comparison to beautifying.
- Textual information to work with might be already available or there might be a simple transformation into text. This way a developer does not create new information from scratch but can use some existing data to work on, e. g., from XML files, other interchange formats or databases.

There are already well-accepted approaches for *textual modeling* available. Examples are the Textual Concrete Syntax (TCS) [JBK06] or Xtext [EV06], both frameworks for Eclipse. The developer specifies the meta-model of

the DSL and the textual syntax, usually in form of a grammar, and the framework generates parsers and textual editors. The latter are equipped with convenient features like syntax highlighting, auto-completion, static analysis, structure outline view, source code navigation and folding. Textual models will be parsed into the actual domain model data structures so they can be processed like all other domain models.

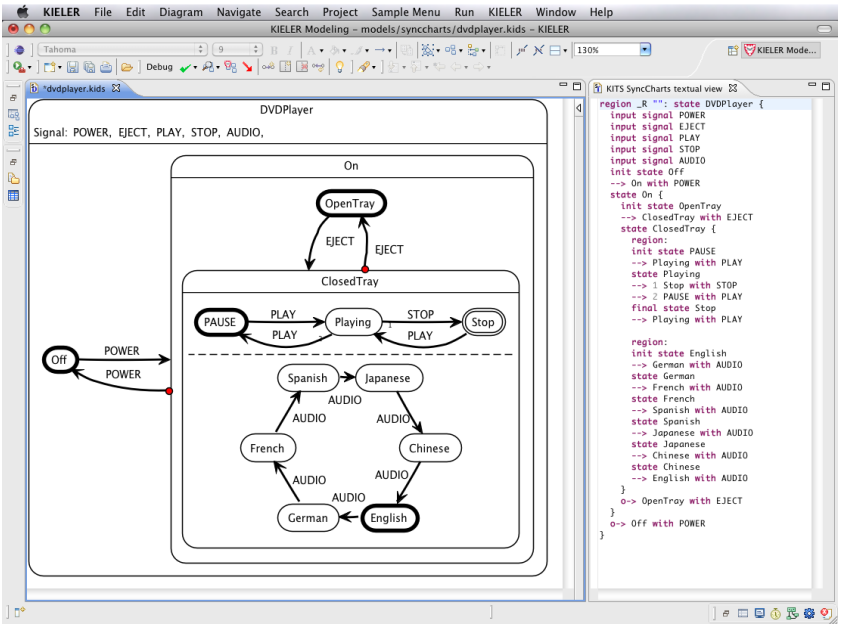
The missing link is the one to a graphical model. Here, automatic layout and view management can be used to synthesize the graphical representations from the textual ones. This can be done in different levels of integration:

1. A graphical model is only once *initialized* from the text. Afterwards the graphical model is worked on. Usually there is no way back into the textual model.

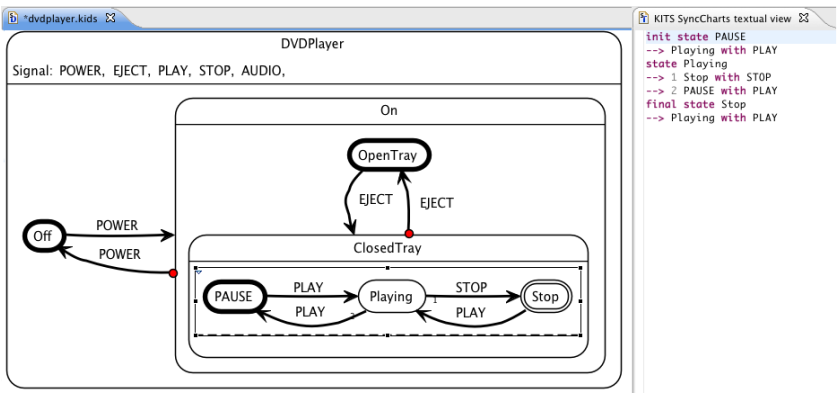
2. There is a transformation between textual model and graphical model in both directions. This is usually denoted as *round-trip engineering*. Some dedicated commercial tools support this for special DSLs, usually class diagrams, but this is still uncommon (cf. Appendix A).

3. A tight integration perfectly synchronizes textual and graphical representation. Hence, the user sees two different views and every change in either of the views automatically updates the other view. So the views are interchangeable even for small editing steps. This paradigm has been explored in KIELER for Statecharts (see Chapter 4) as shown in Figure 3.19a and is applicable for other DSLs as well. It combines well with the focus & context effect discussed above as shown in Figure 3.19b.

4. To increase the integration further, text and graphics could be mixed in one view. If there is a textual representation for single graphical objects, there could be two different views of the graphical model. One view displays all graphical entities, while the other exchanges one of the objects with a text box containing the textual representation of only this model part. This is shown in Figure 3.20, where only one Statechart region is presented as text.



(a) Synchronization of diagram and text in KIELER.



(b) Focus & context in both, diagram and text. Only the selected item—here a region—is shown in the text view and is the focus in the diagram. The other region is collapsed.

Figure 3.19. Textual and graphical synchronization with different levels of integration.

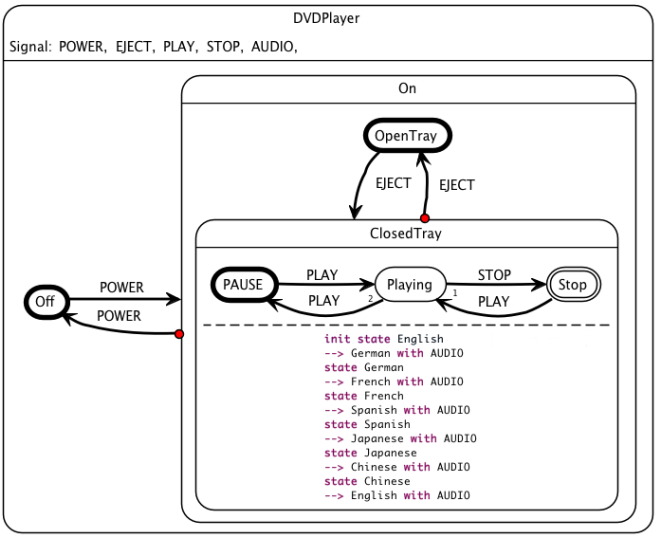


Figure 3.20. Integrated text for a single region in the diagram (montage).

Scalable Models

Model synthesis can be applied together with scripting techniques to create complex and large models according to predefined and parametrizable patterns. Scripts of different flavors could be applied just like *scripts*, *macros* or *templates* in textual languages. This leads to *scalable models*, as investigated by Feng and Lee [FL08]. In their case the scripts that configured the model creation process are in the same graphical syntax as the models themselves. More sophisticated automatic layout techniques could enhance the graphical results. This approach could be applied more generally for arbitrary DSLs and combined with an appropriate user interface.

Pattern-Based Modeling

Development patterns are a common technique in software engineering. When creating behavior diagrams such as Statecharts or data flow models,

one should model common tasks in a common way. This naturally leads to *patterns* for graphical modeling [GHJV95, Dou03]. Examples are patterns for error handling, sequencing or loops—depending on the DSLs, many more can be identified. Graphical modeling environments could support the usage of pattern-based development in various ways.

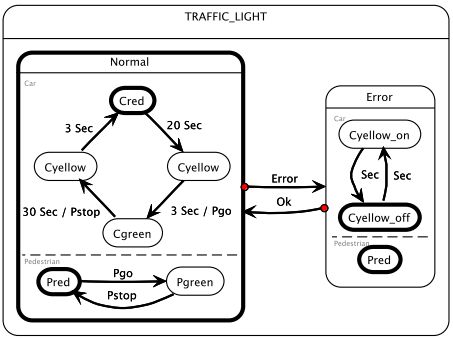
- ▻ Design patterns can be accentuated in a model [Pet08].
- ▻ A specific pattern can be chosen and parametrized by the user and added to a graphical model.
- ▻ The view management should support user-defined automatic layout schemes according to a given pattern [Pet08]. If in a state diagram a loop should be modeled, this could correspond to a pre-defined graphical positioning of the nodes, e. g., in a circle or in a sequence with one back transition.
- ▻ Analysis of the model could detect certain patterns for standard operations such as graph transformations [BEK+06, KASS03]. Additionally it should be able to layout existing patterns to given pattern layout schemes.

A simple user interface is necessary so even beginners and intermediates can quickly start to employ patterns in their development.

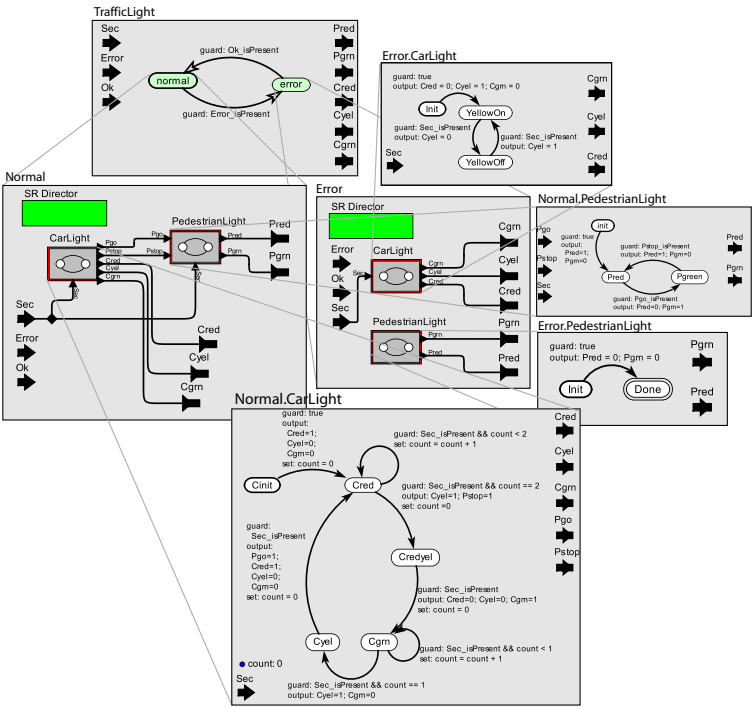
3.3.4 Multi-View Modeling

So far we were considering multiple views only within the same DSL. We change the levels of detail in certain circumstances to get the best trade-off between overview and details. One can drive the idea of multi-view modeling further by defining completely different views instead of only manipulating the focus and context configuration.

The term *multimodeling* refers to employing multiple modeling semantics in one single model [BCF+08]. For example mixing different semantics such as synchronous data flow with state machines and discrete events or others is a preeminent feature of the Ptolemy modeling framework. This still keeps only one view on the same model, although the model itself is of very



(a) SyncChart model



(b) Ptolemy model

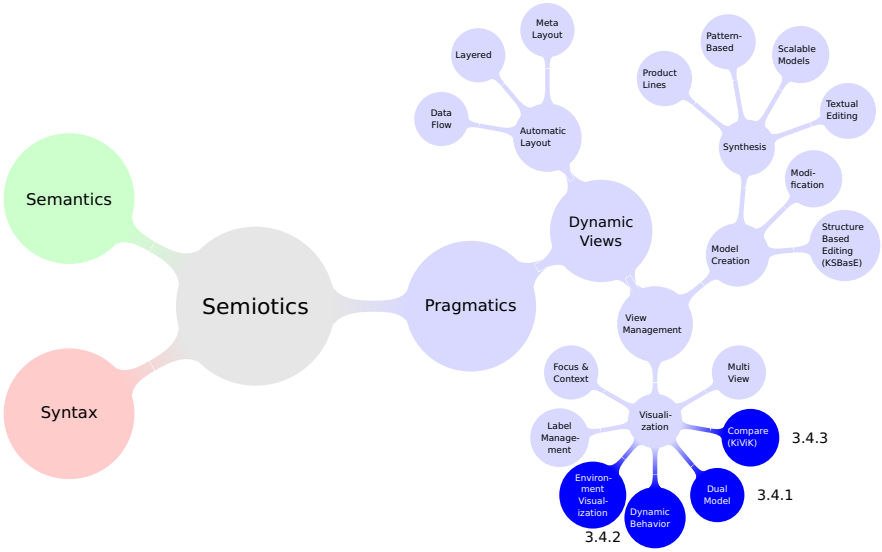
Figure 3.21. From SyncCharts to Ptolemy: both models implement the same behavior [BCF+08].

heterogeneous character. However, one can for example establish semantic equivalence between SyncCharts and mixed synchronous reactive and state machine models [BCF+08]. Hence, for the same semantics, there exists a SyncCharts and a Ptolemy model that implements that behavior. This means that for the same semantic behavior there exist multiple different graphical representations, each with their advantages and disadvantages. Considering the example in Figure 3.21, one might argue that the Statechart model is more compact, but the Ptolemy model makes further information explicit, notably the information flow (however, this might get mitigated by the *dual model* presented in the next section). We could exploit the equivalence by transforming a SyncChart into a Ptolemy model or vice versa—at least for suitable Ptolemy subsets. The disadvantage would be that we still have two completely different models including two different domain models. Both models could be transformed only as whole in a global transformation of all model parts.

An alternative is to keep only one common domain model and on top of that create two different graphical representations, one for SyncCharts and one for Ptolemy. This would be always applicable where one model part can be expressed in multiple ways. Then the model part could have multiple completely different views. The major benefit would be that the different graphical representations could be interchanged in any hierarchy level, resulting in a mixed graphical model. The different views could be handled by the view management just as the other views proposed above.

While we have now discussed *what to display* and how to *create and maintain* it, in the following section we want to see what we can *do* with a model and how analysis and interpretation can benefit from view management.

3.4 The Controller—Interpreting the Model



A model shall give a comprehensible representation of a system. Its semantics serves to collect new insights about the system. Complex analysis can be executed on models or simulations can be run. However, so far the view of a model is used only very little to feed these new results about a model's properties back to the user.

We distinguish two major categories of model analysis: *static* and *dynamic*. We will consider some examples for static analysis use cases, but the main focus of this work is on dynamic or *interactive* analysis.

Sophisticated static analyses can determine properties of a model, for example causality issues for actor-oriented data flow models [ZL08]. If such an analysis determines certain properties of a set of model elements, it can be used as a trigger for the Meta Layouter in order to get a visual feedback of the analysis. Especially a categorization of model elements in two sets

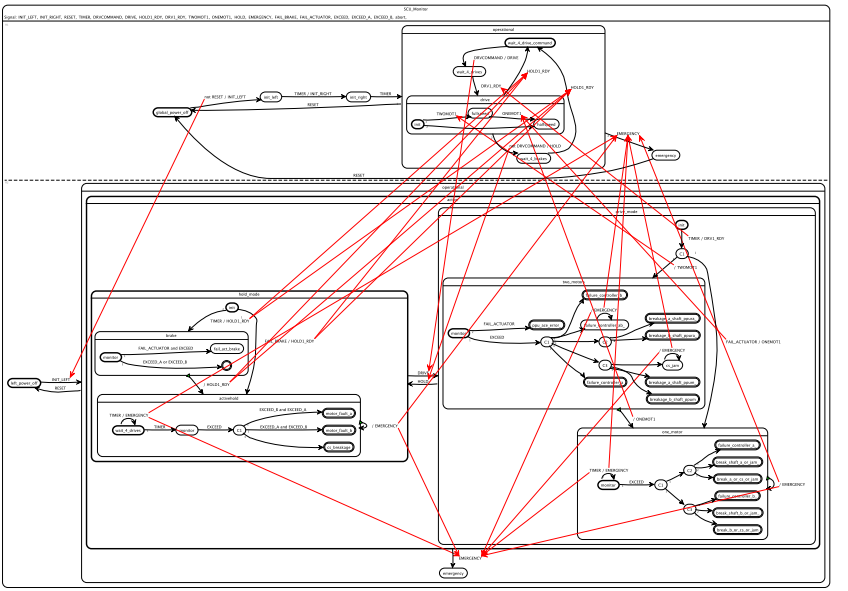
can be interpreted as a categorization into focus and context objects. For example an analysis of dynamic and static parts of a model under certain input values can visualize only "active" parts of the model or the flow of control that was taken.

3.4.1 Dual Modeling

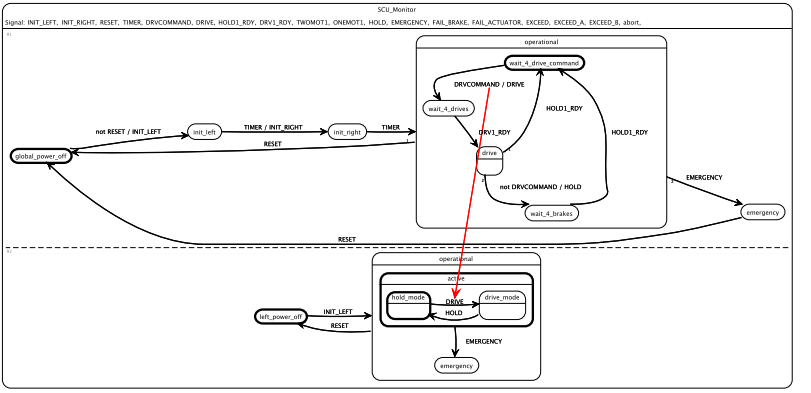
Next to focus and context views, we now examine another means to better understand the *references* in a graphical model. We see the following problem with typical DSLs: the graphical representation depicts the main model objects as nodes, where the containment relations can be reflected by hierarchy in the model and containment of graphical symbols like rectangles. Therefore, the diagram exhibits intrinsic properties, and these properties directly correspond to properties in the represented domain [Gur99]. Explicit connections display some other relations between the model objects. However, there is typically a set of model attributes that is hidden in simple property dialogs or simply represented by a label in the graphical representation. Relations between those attributes are usually not visible.

We propose a dynamic extension of the graphical representation by its *dual model*, i. e., a graphical representation of the relations between referenced objects where this reference is not yet visualized. Take again the example of Statecharts. The dual model of a Statechart is a graph where the transition labels are the nodes, and the relations between guards and actions form the connections. The graph shows which transitions produce triggers and which ones read those triggers. It makes explicit how the broadcast communication is used by showing the flow of data and signals in the model. By graphically overlaying the original graphical representation with its dual model, we reuse the same graphical view in order to keep the mind map of the user, as illustrated in Figure 3.22a.

Selecting only specific signals or transitions could reduce the dual model edges to those relevant for the selection. E. g., for a transition with a guard trigger, only the other transitions are connected that emit the corresponding signal. Figure 3.22b shows an example where the data flow only for one signal is shown, depending on the selection of the user. Irrelevant composite



(a) All communication.



(b) Only showing data flow of the `DRIVE` signal. Focus & context only shows the relevant parent states in full detail allowing to scale the diagram up.

Figure 3.22. Dual Model for Statecharts: Two parallel controllers communicate via broadcast. The data flow is displayed as an overlay of the original control flow graphical representation.

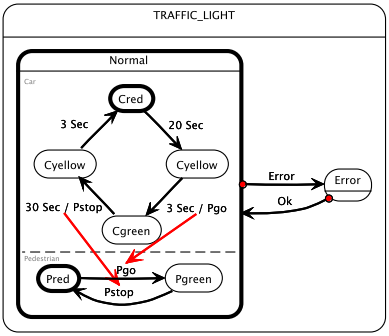


Figure 3.23. The dual model view of the traffic light example from Figure 3.21. It reveals a rather simple communication.

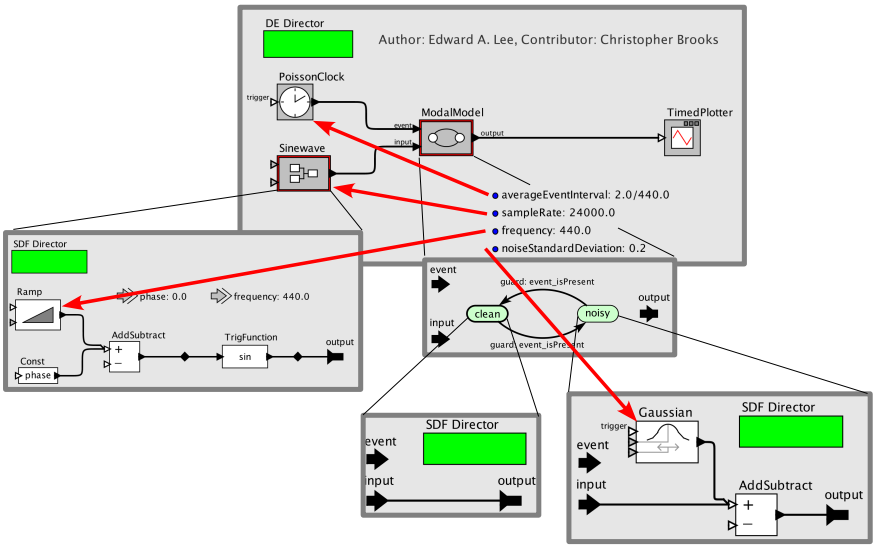


Figure 3.24. A dual model for Ptolemy could show where parameters of an actor are used.

states are collapsed to show only what is necessary. This way, the user can dynamically explore the communication and learn about the model. Using this selective aggregation, focus & context techniques as introduced before can create custom comprehensible views for complex models.

Figure 3.23 reveals the rather simple communication of the multi-view traffic light example. Signals without arrows are global in- or outputs. The collapsed `Error` state has no inter-communication at all. The original Ptolemy view also shows this communication explicitly, however, the simplicity is more obvious in the dual model; maybe also due to the visible hierarchy there.

The *dual model* methodology should not only be helpful for Statecharts, but applies to very different types of models. References to other model parts are quite common where an explicit graphical representation is omitted for the sake of clarity in the original model. Two examples are:

Class diagrams The attributes of a class are presented more or less textually including the type of the field. However, the type may also reference another class or a data type definition node in the model. The dual model of a class diagram would reveal the data type usages of the classes and their attributes.

Ptolemy II In Ptolemy one can define arbitrary parameters of actors. They are represented by an unconnected node only showing the key and the value of the parameter. Then they get referenced by arbitrary expressions in Ptolemy's expression language, which is just text. They are often used to map parameters of lower-level actors to the top-level actor. The dual model could explicitly show which objects use which parameters. An example montage is shown in Figure 3.24. Technically this would work best if the editor would use visible hierarchy, which the Ptolemy editor Vergil does not (cf. Appendix A).

3.4.2 Dynamic Behavior Analysis

We usually distinguish the *structure* and the *behavior* of a model. To validate behavior, it is common practice to employ simulations prior to physical

deployment. Therefore DSLs with known specified semantics get employed, such that the models can be executed.

Simulation Management

Employing view management during testing gives us the same benefits as for simple manual browsing: "interesting" parts can be put into the focus while the context is still visible. Additionally, a simulation run gains a new dimension: time. Hence, there might be times where nothing of relevance happens and other points in time with interesting events. The problem is to determine "interesting" parts and times during simulation.

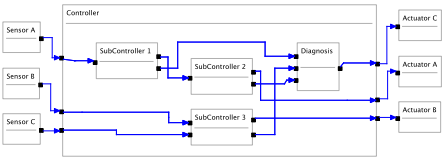
This suggests to extend the view management by *simulation management*. It defines an additional set of system events for triggering view management effects and additional effects for manipulating simulation time.

Both simulation triggers and effects are highly dependent on the language semantics. Hence, a simulation manager is usually only applicable for a small set of DSLs.

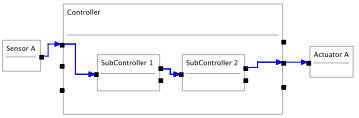
Visual Breakpoints

Simulation *triggers* are customizable conditions over internal states and variables of the simulation. Hence, both the specification and the interpretation of those triggers require access to the semantics of the model and a simulation engine. The triggers cause effects. These are on the one hand usual view changing effects, such as graphical focus change events. On the other hand they are *simulation effects* that alter the behavior of simulation time, such as simulation pause or stop. Therefore we propose to use simulation or execution as visual interactive *debugging*.

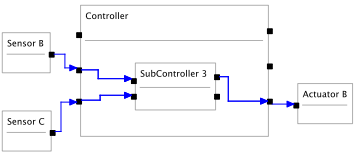
A simulation manager should allow to specify *visual breakpoints*, the combination of a specific target view with the condition under which this view will be shown. Additionally it could comprise the pausing of the simulation to give time to inspect the situation. A properly configured simulation manager knows what "interesting" items are, both in time and model objects. So during simulation a user always gets to see the right parts of interest without any user intervention; no manual navigation actions are



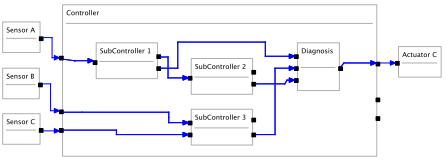
(a) Original controller



(b) All dependencies of `Actuator A`, two sub-controllers and only `Sensor A`



(c) Dependencies of `Actuator B`



(d) Dependencies of `Actuator C`: it requires all input sensors.

Figure 3.25. View Management in a data flow language for some embedded controller with three sensors and three actuators.

required.

An example for data flow diagrams is shown in Figure 3.25. Here a focus is set to one actuator and all components in the data flow towards that actuator. Other components are filtered. This results in tidy diagrams that illustrate specific aspects, e. g., for analyzing `Actuator B`. During a simulation run, the respective view could be shown whenever some specific value is received by one of the actuators. The way of actually displaying the data is another issue but could be integrated into the diagram. The dynamic focus and context technique implemented in KIELER for SyncCharts (cf. Sec. 4.4) is implemented in a straight forward fashion by adding simulation events for every state change and setting the set of focus objects to the active states.

Simulation Tracking and Control

It is common practice to show (highlight) the current *state* of a system. In some areas, it is also common to show the current *change of state* (e. g.,

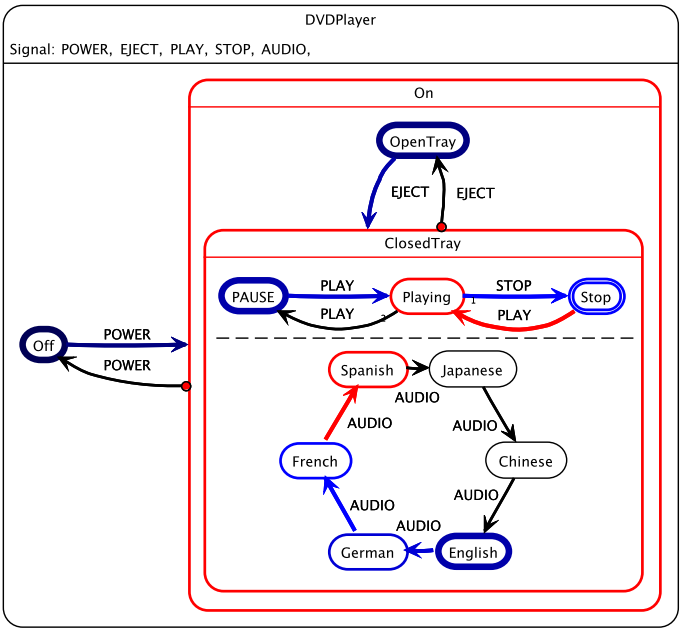


Figure 3.26. Showing active states and taken transitions (red) and the recent history (shades of blue).

a transition in a Statechart). There are natural extensions that one could consider, such as showing the *recent past* (e. g., the last n states as shown in Figure 3.26), or the *possible future* (states that might be reachable in the next n steps, this would require some kind of static/dynamic analysis).

A desirable feature is to be able to not just run a simulation and to stop it at certain points, but also to step backwards again. This *tape recorder paradigm* has already been integrated into some modeling tools, e. g., Statemate [HLN+90].

Data Visualization

During a simulation run one wants to be informed about the internal state of the simulation. That comprises states for state machines and other data. For states, the presentation is quite natural. For other data, it is not that obvious as we have seen in the introduction in Section 1.2.

Experience with the state-of-the-practice tools leads to the following basic requirements for proper interactive data visualization:

Basic Aesthetic Criteria The original diagram should not be altered in a way that breaks basic aesthetic criteria. Especially overlaps should be avoided. Data as well as the diagram must still be readable.

Map Data to Diagram It should be easily possible to map displayed data to the part of the diagram which is responsible for producing the data.

Avoid Permanent Enrichment of Diagram Adding special visualization nodes to a diagram permanently enlarges its view. Even when watching the diagram for other use cases where little or even no data visualization might be required, the view will be bloated. It is like in textual programming languages: the source code should not be cluttered with big amounts of explicit debug output messages, which make the original source code hard to read.

View management employing automatic layout offers approaches to data visualization that can fulfil these requirements:

Data Nodes In general, adding nodes for data visualization creates useful views: automatic layout can take the visualization nodes into account and therefore can avoid overlaps and adhere other aesthetic criteria.

Dynamic Insertion With view management these nodes could be added to the diagram dynamically only when they are required. View synthesis with automatic layout allows to create multiple different views, e. g., a view containing no data nodes and another view containing all of them.

Conditional Insertion This dynamic insertion could be done to show only the currently "interesting" data. Instead of adding such nodes to all

potentially interesting parts, conditions could be formulated as view management triggers. When they fire, data nodes get inserted to a specific data connection. For example when a data value exceeds some user-specified maximum allowed value, a data node could be inserted. This could help to examine why a model shows some exceptional behavior.

3.4.3 Model Comparison

Complex systems are often developed by teams, where multiple developers work on the same models. They create models together or some developer has to maintain older models that have been created by somebody else. Therefore it is important that developers can easily get information about model changes, and more generically, about the differences between two arbitrary models.

However, diagrams offer multiple dimensions—usually two—instead of only the one-dimensional texts. This is used to depict the contents in an appealing and comprehensible way. The diagrams often are some kind of graphs, where the graph itself holds its semantic. Inherently graphs can be represented by many different embeddings, and a given embedding can be drawn in different layouts. Additionally some information that may have changed is not at all reflected in the diagrams but only within some internal properties of graphical elements. Therefore it is often difficult to manually compare two graphical representations to see what items are different and even what parts are the same when only diagram layout has changed. Therefore computer assistance in finding changes is a crucial feature in collaborative and iterative development.

It is of great importance to the success of system modeling that tools offer intuitive and easy to use interfaces to create and change the model. A major concern is the depiction of changes in graphical models in a graphical way, visualizing the changes in the same manner in which they were produced. This prevents the user (of the modeling tool) from switching of different abstraction levels in trying to map the textual description of the differences to the diagram. As observed by Mehra et al., graphical comparison is an advantage, as this is the natural way to compare visual

objects [MGH05]. The still prevailing approach of converting the differences to some structured text is just a workaround due to the lack of better methods. Transforming the textually displayed changes back to the graphical world requires, according to Green, a "hard mental operation" [Gre89], which is unnecessary and should be avoided. Depicting changes in the diagram itself helps the developer working with it to understand the resulting modification immediately in its meaning. As Ohst et al. point out, it is not appropriate for two-dimensional documents like diagrams to display possible changes in the traditional way, in two linear columns with corresponding elements facing each other [OWK03].

That there is a real need for a visual comparison is also supported by commercial applications recently introduced that try to provide at least some limited form of graphical comparisons. SCADE Model Diff (Esterel Technologies) and ecDIFF (Expert Control), intended for Simulink (Mathworks) models, were introduced into the market in 2007 and 2008.

I here propose different alternatives to display differences of graphical models. It appears to be useful to combine well established means, like structural comparisons and colored side-by-side confrontation, with advanced model presentation and interaction techniques, like automatic layout, navigation and folding. The illustration here focuses on Statecharts. However, the techniques are language independent and should be applicable to other languages as well, including those of the UML.

Visual Comparison

When talking about a reasonable visual comparison, issues like layout, as well as the *mental map* [MELS95] have to be taken into account.

We see two different use cases for visual comparison which have different technical requirements:

1. The mental map of the user, that is the position of elements, their connections and sizes, should be preserved as much as possible to support the user's understanding of different diagram versions [PHG06]. This applies also to the visual comparison, when different diagram versions are presented to the user and may get changed by the framework to visualize changes. This is especially the case when the diagrams were

still created with freehand editing tools or the developers are explicitly using specialized secondary notation to put implicit information in the drawings.

2. Conserving the mental map of the diagrams to compare is of no importance. This may be the case when a third person is comparing the diagrams that had no prior knowledge about the originals. At least the editing history does not play any role, for example because there is an automatic-layout policy already provided in the company. This simplifies matters when it comes to questions about layout as discussed in the following.

There is no obvious best strategy. Whether one wants to conserve the mental map or not should still be the decision of the developer or the respective development team as already discussed before. Therefore the following approaches should cover both cases.

The following classification of possible visualization mechanisms combines related proposals (presented in the related work Chapter 2) with further alternatives. The two versions of a diagram to compare can be selected by the user. This will generally, but not necessarily, be the actual version where he or she is working on and any older one. To illustrate the alternatives, we will compare the Statechart examples shown in Figure 3.27. The classification is as follows:

Plain: The two original layouts are just shown side by side, with colors or similar markers indicating differences. This is illustrated in Figure 3.28a and Figure 3.28b.

Animation: A small animation or video is created, which shows the transition from one version to the other by morphing the Statechart, thus maintaining the mental map of the user.

Hover pop-up: Having enabled the compare mode, the user can navigate through the one version of the Statecharts, which is annotated with modifications, and hovering pop-ups will show in detail the changes that occurred in the neighborhood relative to the other version.

Free merge: A merge of the two versions is calculated. This merged model will be laid out from scratch, with colors showing alterations from one release to the next one. This can be seen in Figure 3.28d, the coloring is like in *plain*.

Incremental merge: This is similar to the free merge and shown in Figure 3.28c. The calculation of the merge remains the same. The layout is not computed from scratch, instead one of the original layouts serves as a reference for the merged layout, maintaining the mental map of the developer. This requires the usage of automatic layout algorithms with good layout stability (cf. subsection 3.2.1).

A side by side comparison, in its static case as described here, is the simplest way of comparing entities. The first thing coming into mind as an analogy for this type of comparison is the ordinary textual diff, enhanced by a graphical representation showing the versions in two columns with corresponding text blocks at the same vertical level.

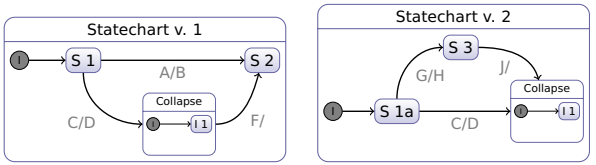
There are several advantages to this mechanism. No new layout has to be computed. Just the two existing layouts are next to each other. In this manner, different colors could help the developer to discover the changes. This is particular true for *states* that just have changed attributes, a characteristic which cannot be detected in a graphical model at first glance.

It is difficult to answer which way is the best to render the visual diff. Hence, view management should come into play to allow customization to the user's needs. Therefore view management should be augmented by triggers and effects that control the comparison:

Comparison Trigger A comparison trigger would give the technical information for the comparison. I. e., the data about the differences in the models, what objects have changed, what are new and what were removed.

New View Effect An effect would be required to create a completely new diagram. This could be fed with elements of the respective models. If a side-by-side representation would be required, then two such view effects could be used.

Target Editor Effect A diff combination would have to display the newly created views into a new editor window. For example a side-by-side



(a) Version 1 **(b)** Version 2

Figure 3.27. The two original versions of the example diagram.

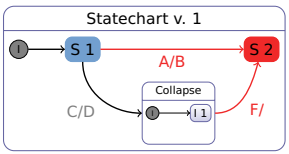
representation usually uses one widget that contains two views of the models. A target editor effect would specify the configuration of widgets where the views are shown.

This toolbox within view management allows to create different combinations that implement the diff alternatives discussed above. Additionally they would use the layout effect of meta layout to possibly re-arrange the views and the already proposed highlight effect to visualize the differences.

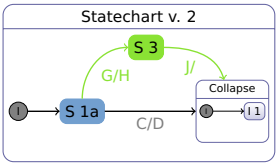
3.5 Summary

I have presented an overview of different aspects of modeling pragmatics. A guiding principle has been the model view controller paradigm, which has been quite successful in software engineering and which seems to have much to offer in the world of model based design as well.

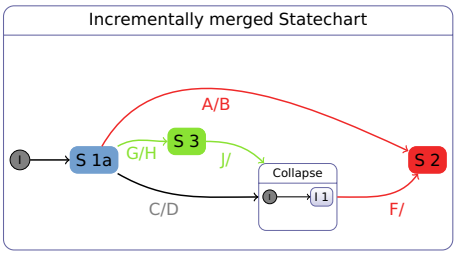
We consider automatic layout of the graphical representation to be one of the basic key enablers for good pragmatics. We build upon layouters by dynamic filters that reduce the complexity of diagrams and focus & context as a special case of such filters. A view management engine organizes different dynamic views synthesized with filters in order to assist the user in seeing the "interesting" parts of the model. We extend the view management by meta layout, which plays with different layout styles even within different parts of one graphical model in order to get optimal layout results.



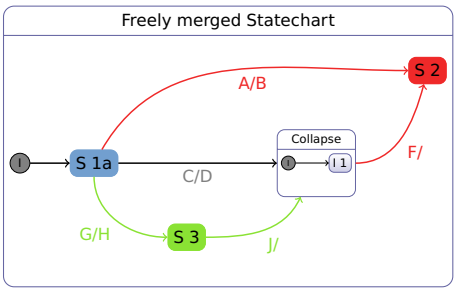
(a) Plain visual diff. Ver. 1



(b) Plain visual diff. Ver. 2



(c) Incrementally merged visual diff.



(d) Freely merged visual diff.

Figure 3.28. Different ways to compare visually. Color legend: green/additions, red/deletions, blue/changes.

These building-blocks support a set of use cases in the modeling process that help to cope with very large model instances. Structure-based editing for creation and modification frees the user of many manual effort prone tasks. Auto-layout enables graphical model synthesis and opens the door

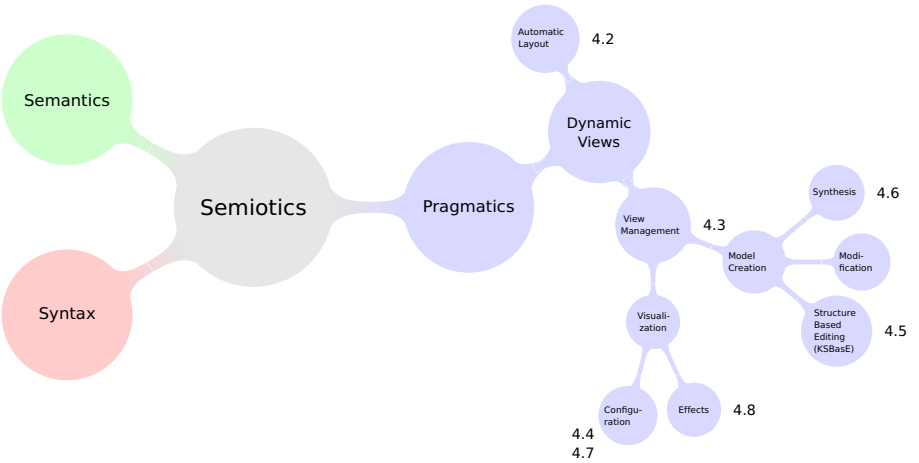
for perfectly synchronized textual and graphical representations, scalable models, pattern-based modeling and support for product lines.

The concepts presented above are the vision of tool capabilities consistently building upon automatic layout of graphical models. It is a vision of a new consistent integrated practical MDE process with enormous potentials. Implementing all of these concepts is an ambitious undertaking, effort prone and venturesome; it requires acceptance of the community to permeate. However, every stepping stone that is realized contributes enhancements to usability, productivity and tool acceptance by its own.

Let us see in the next chapter which of the stepping stones have been built so far in the implementation project Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER). It will show how far we have already gone along the presented path and what still is left to do.

Chapter 4

The Implementation: KIELER



Many approaches presented in this work are implemented and evaluated in the project KIELER, the Kiel Integrated Environment for Layout Eclipse Rich Client[1]. In the spirit of genericity, KIELER builds on the plug-in concept provided by Eclipse and especially its modeling projects[2]. KIELER provides enhancements for pragmatics, to be combined with syntax and semantics defined by other projects. A screenshot is shown in Figure 4.1.

[1] http://www.informatik.uni-kiel.de/rtsys/kieler
[2] http://www.eclipse.org/modeling/

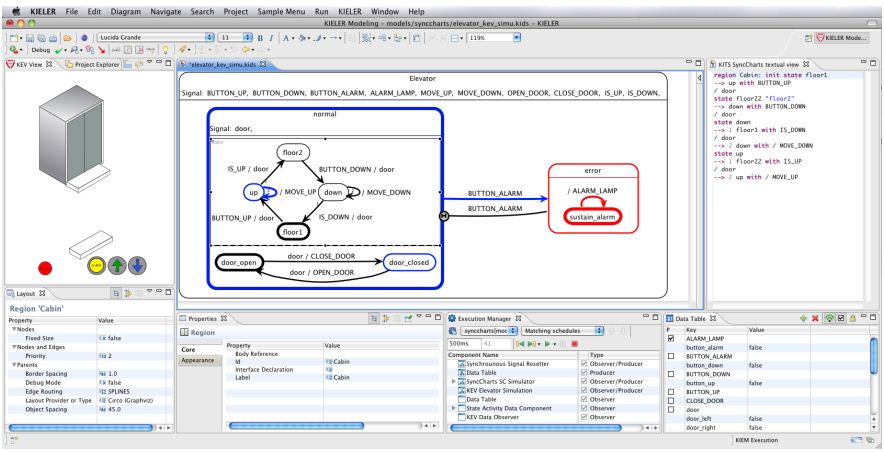


Figure 4.1. The KIELER modeling application: focusing on usability and genericity.

The Eclipse platform offers powerful software engineering concepts to support the creation of highly modular Java software. The KIELER project leverages many of the features that Eclipse provides. Therefore I will give a quick introduction into the platform and the main concepts from which KIELER benefits in the next Section 4.1. If you are familiar with the internals of Eclipse (extension points, modeling projects), you can safely skip this section.

Above, concepts have been described in order of abstraction degree from pure automatic placement to guidance on where and how to reduce the levels of detail of views on graphical models and how the MDE process benefits from that.

The implementation of these concepts is divided into two main subprojects, presented in the following: KIML in Section 4.2 and KiVi in Section 4.3.

The implementation of meta layout is the KIELER Infrastructure for Meta Layout (KIML); see a small screenshot in Figure 4.5. It comprises all topics that have to do with automatic layout, including interfaces to layout algorithms, the infrastructure to parametrize them and finally how to choose layouters themselves.

The second is the KIELER View Management (KiVi). On a higher abstraction level it aims at managing the diagram view detail levels following the current user context. On a lower level it includes all necessary implementations like certain effects such as filters, collapse-mechanisms, highlighting, and the run-time infrastructure to define and combine the user conditions with the desired effects.

4.1 Eclipse

Eclipse originally was developed by IBM since 1993 as an IDE for object oriented programming languages such as Java, C++ and Smalltalk under the name "VisualAge". The source code—Java—was opened in 2001 which emerged the new open source Eclipse IDE. Still more than half of the base developers are working for IBM. In 2004 the Eclipse Foundation was established, which is currently responsible for the further development of Eclipse.

Since version 3.0 Eclipse implements the OSGi specification[3] that defines a platform-independent component model for a better modularization of software. From that version on, Eclipse consists of a small runtime kernel, called *Equinox*, which coordinates the set of *bundles* (OSGi terminology) or *plug-ins* (Eclipse terminology), which themselves implement the original features of the IDE. Therefore the notion *platform* is used instead of IDE. The set of plug-ins in Eclipse that one actually uses is not fixed. However, there is some common terminology in the community, of which a roughly sketched example setup is shown in Figure 4.2. The kernel together with a set of base plug-ins is usually denoted as the *Eclipse platform*. Together with concrete programming language tool kits—with editors, compilers, builders, debuggers, etc.—it comprises what we usually call an IDE. The large modeling community has developed a growing set of plug-ins dedicated to MDE. These integrated in Eclipse form the *modeling environment*. Using the Eclipse platform as basis for a client for one's own application is denoted the *Rich Client Platform* (RCP). To minimize resources this can be configured, such that only the plug-ins necessary for the client are used.

[3] `http://www.osgi.org`

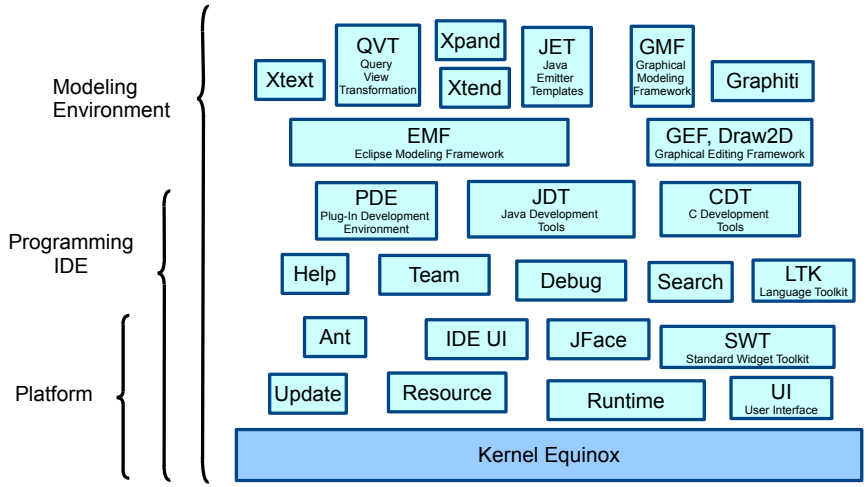


Figure 4.2. Overview of the Eclipse Platform

Extension Points

The main feature of Eclipse that simplifies collaborative development is the plug-in mechanism. OSGi defines the concept of *bundles*, which are small packages of code whose lifecycle is controlled by the runtime kernel. This allows to *install*, *start* and *stop* software modules during runtime. Eclipse extends this mechanism by specifying the way of plug-in *interaction*. In order to do this eclipse uses its *extension point* concept, outlined in Figure 4.3.

There are two basic ideas that are important:

Laziness In order to cope with the complexity of a big set of plug-ins—maybe hundreds or thousands—, Eclipse avoids to execute the code of all available ones. The laziness principle states that only plug-ins that are actually used by the user get activated. This is done when the user first requests functionality of the corresponding plug-in. It requires that each plug-in provides some *meta information* that specifies

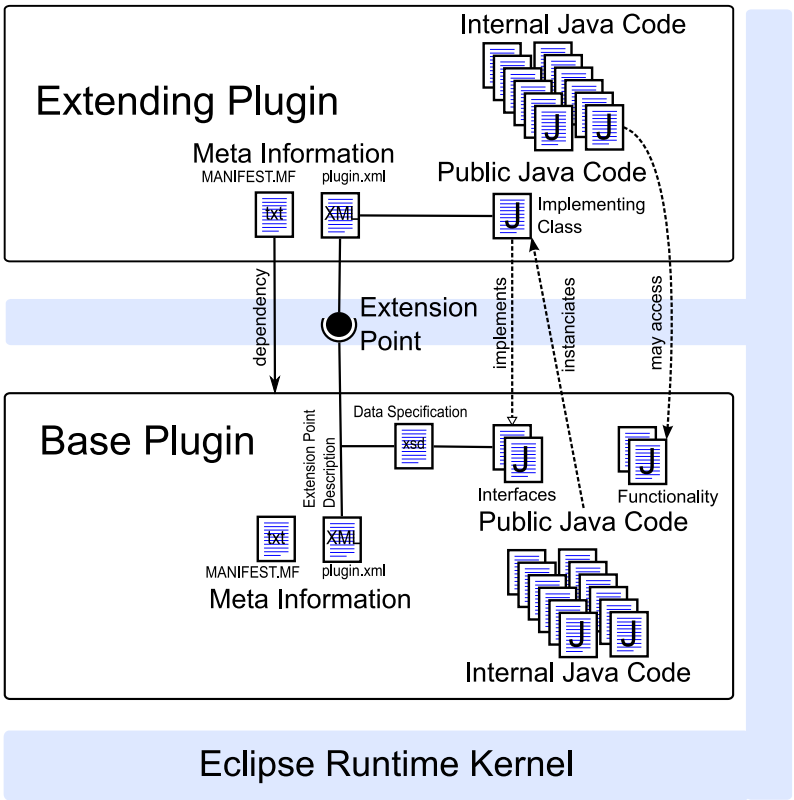


Figure 4.3. Eclipse Extension Point Mechanism

the features the plug-in contributes. Otherwise some Java code would have to be instantiated to inspect the plug-in code and learn about it. Therefore the plug-in archive carries textual specifications (`MANIFEST.MF` and `plugin.xml`) which specify the UI contributions it provides (e. g., buttons, menus), the other extension points it extends and what dependencies it has on other plug-ins.

Extensibility There are different qualities of code interaction between plug-ins. The simplest one is that one extending plug-in (`E`) uses existing code of some base plug-in (`B`). `E` simply may access any code that `B` has specified to be public. The more interesting case is the other way round. While `B` is implemented first, it can access code of `E`, although `B` had no access to this code when it was implemented. However, the developer of `B` might have anticipated that someone wants to extend the functionality of `B` later, and therefore has created an explicit *extension point*. This specifies which meta information `E` has to provide and what source code interface it has to implement. Then, during runtime, the Eclipse kernel will handle the instantiation of `E`'s code and pass it to the base plug-in `B`.

Specification of the required meta information uses an XML notation. For an extension point, the set of required meta data is specified using an XML Schema Definition (XSD). The Eclipse kernel can interpret these meta information during runtime rather efficiently without the need to load any Java classes unless they are explicitly required.

Modeling Environment

Models *represent* some software artifact or real world domain and *conform* to a *metamodel* or *grammar* (cf. Section 3.1). The metamodeling backbone in Eclipse is the well-established *Eclipse Modeling Framework* (EMF).

One modeling standard of the OMG is the Meta Object Facility (MOF) [Obj06]. It specifies object-oriented structures using a class-model with exactly 3 meta-levels, where strictly speaking M0 is no *meta*-level:

M0 holds objects of reality, that are going to be represented by models.

M1 are the models of real objects. Examples are concrete Simulink, Statecharts or UML Activity models.

M2 are metamodels that specify which model objects are allowed in model instances. It describes the *abstract syntax* of models. Usually class models are used for this purpose. So examples are UML class models or an EMF Ecore model (see below).

M3 is the meta-metamodel level. It describes the abstract syntax of metamodels and is the highest level that the MOF describes. The only entity in this level is the *Class*.

Some drawbacks of this structure are discussed in Section 5.2. It seems that the MOF is mainly applicable for object-oriented structures but has some issues with *actor*-oriented approaches that use a prototyping mechanism to dynamically extend the possible syntax in shape of an actor library.

A useful subset of the MOF is the *Essential MOF* (EMOF), which is almost exactly implemented in EMF. It uses only a small subset of UML class diagrams that are considered sufficient to model object-oriented data structures and simple enough to easily learn and to provide mature code-generation facilities for it:

> "EMF relates modeling concepts directly to their implementations, thereby bringing to Eclipse—and Java developers in general—the benefits of modeling with a low cost of entry." [SBPM09]

Inputs for EMF metamodels—so-called *Ecore* models—are class diagrams, annotated Java source code, XML Schema Definitions (XSDs) or manually created structural Ecore model trees. To each metamodel EMF generates a Java implementation that allows to instantiate that model programmatically. A large number of related projects has emerged in the context of EMF. There are different ways to edit concrete models, such as tree, text or diagram editors. There are projects that process models, such as validation, model transformation—either model to model (M2M) or model to text (M2T, aka. code generation)—or model comparison (cf. Section 4.7). Additionally, there are different means to persistently store models. They can be saved to files in XML format, put into data bases or serialized to arbitrary text.

This flexibility and the active community make EMF a good basis for many modeling projects. A special focus of this thesis lies in the ways to implement graphical diagram editors to create and view EMF model instances. This is introduced in the following.

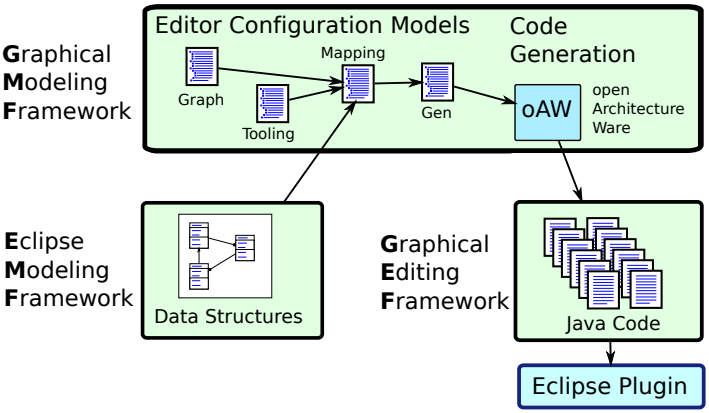


Figure 4.4. Process of creating a graphical editor with GMF

Graphical Modeling

The Graphical Editing Framework (GEF) is a runtime library for Eclipse that provides the basic infrastructure for graphical diagram editors. It offers the lightweight 2D drawing toolkit *Draw2D* and a model-view-controller concept for user-tool interaction, focusing on freehand DND editing. Using this framework requires to implement the target diagram editor manually by creating many Java classes which are using or extending the library classes. The connection of abstract syntax (EMF) with the concrete graphical syntax (Draw2D) must be established by hand. Hence, implementing such an editor is effort-prone and has a steep learning curve.

There are different approaches on how to ease the implementation of a GEF editor. The upcoming Graphiti framework is also an API-centric runtime library. It uses GEF in the background as a rendering and interaction engine, but offers a much more simplified API that also already covers the connection of concrete and abstract syntax. The toolsmith does not get in touch with GEF and therefore it is replaceable as rendering engine.

Another new ground is broken by the *Graphical Modeling Framework* (GMF). It provides a generative approach to a GEF editor, which is outlined

in Figure 4.4. GMF takes an Ecore model as base for the abstract syntax of the diagram editor. The user has to provide other abstract models that specify the editor itself: a `Graph` model lists the graphical symbols that correspond to semantic items, such as rectangles or arrows; a `Tooling` model describes the possible user interaction, e. g., creation tools for the different model elements that will be presented to the user in a palette toolbar. These three models are combined by the toolsmith to a `Mapping` model which connects semantic elements with concrete diagram symbols and palette tools. From there GMF provides fully automated Java code generation which results in a large set of Java classes that use or extend the GEF library. Bundling this code in an Eclipse plug-in results in a ready-to-use diagram editor if deployed in an Eclipse distribution that also hosts the GEF and GMF runtime libraries. By default, the code generation creates a lot of useful features for the editor, such as creation wizards, usual freehand editing mechanisms and so on.

However, the drawbacks are mainly twofold: First, the process is rather complicated when one is not familiar with the general modeling approach in Eclipse and therefore also has a steep learning curve. Second, the default code generation only generates common features. However, users of such an editor might have special requirements that raise the need for customization. However, customizing the code of a generated GMF editor is again rather complex and requires deep insights into both, the GMF code generation process and the GEF library. Therefore some of the advantages over a pure GEF editor are mitigated.

Some example editors created with GMF are introduced in more detail in Chapter 5. So far, mainly GMF has been employed, because it is mature. However, the upcoming Graphiti API-centric approach appears to be beneficial when it comes to learning the techniques and to customization. It might be a considerable alternative for the future.

4.2 KIELER Infrastructure for Meta Layout (KIML)

The purpose of automatic-layout is to synthesize views automatically, thus freeing the user to focus on the model itself. As discussed further in Section 3.3, this does not only save time formerly spent on manual drawing activities, but yields completely new possibilities for user interaction.

The presented concepts around automatic layout in Chapter 3 contained three main ideas:

1. a bridge between layout algorithm libraries and diagram editors,
2. parametrization possibilities to get the desired layout result of available algorithms and
3. means to automatically choose the right layout algorithm and parameters for a given use case and model to get an optimal diagram. This involves *view management*.

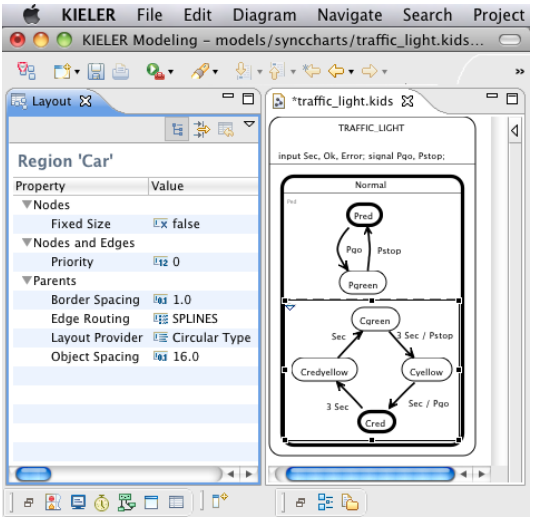


Figure 4.5. KIELER specifying layout options. They can be set for edges, ports, nodes, their role in the parent or their children.

I call the last point *meta layout*, because it works with layout algorithms on a meta-level and is no layout algorithm itself. However, the implementation of all three parts is done in the subproject *KIELER Infrastructure for Meta Layout* (KIML). Figure 4.5 shows a screenshot with the user interface to change layout options. The following introduces the first two parts while the third part will be discussed together with view management in the next Section 4.3.

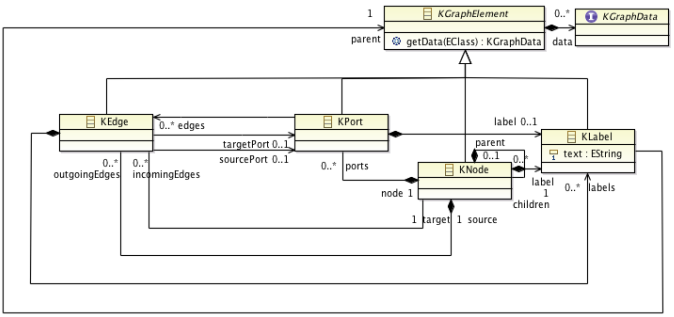
The layout bridge connects a range of layout algorithms with established graphical model diagram editors. Figure 4.6 shows example layout/editor combinations. Figure 4.6a shows the EMF Ecore tools class diagram editor with a Mixed-Upward-Planarization algorithm of the OGDF [GJK+03], which takes into account the different types of edges—inheritance vs. relations. Figure 4.6b is a UML activity diagram of the upcoming GMF-Papyrus UML editor suite using the dot algorithm of the Graphviz library [GN00], which is well suited for compound graphs without inter-level edges. Figure 4.6c shows a use case diagram of Papyrus, employing a force directed algorithm of the Graphviz library. Finally, Figure 4.6d shows a simple actor-oriented dataflow diagram, whose port constraints can be layed out with the KLoDD layouter presented in subsection 6.2.2.

As illustrated in Figure 4.7, the meta layout framework contains a basic graph data structure, the *KGraph* shown in Figure 4.6a, for exchanging data between a concrete diagram editor and a layout algorithm. To achieve genericity, this does not assume any specific format of either of the two worlds. Glue code that translates between used data structures in both domains allows to use any diagram editor with any layout algorithm. The KGraph is used as an intermediate format to (1) formulate the layout problem and to (2) store the layout result, i. e., the concrete coordinates and sizes. The KGraph follows the ideas of GraphML[4] but is simplified to the needs in this context.

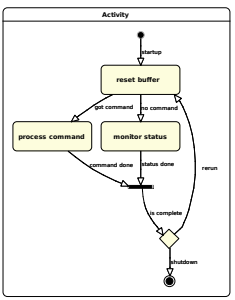
The layout process is executed as follows:

1. Use diagram glue code to read the model structure from its current view (e. g., but not necessarily, from a diagram editor). Create a KGraph from this structure.

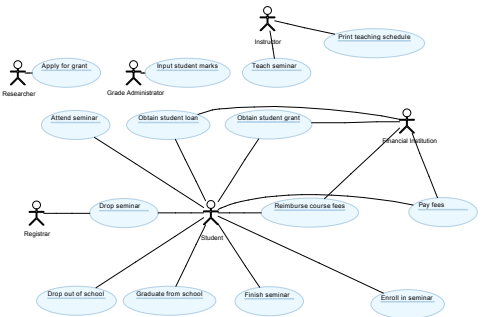
[4] http://graphml.graphdrawing.org/



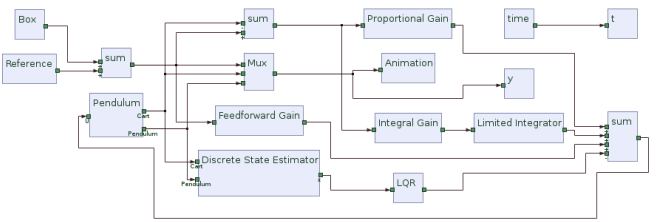
(a) The KGraph as an EMF Ecore class diagram with mixed upward planarization [GJK+03].



(b) Papyrus UML Activity diagram / Graphviz dot layout [GN00].



(c) UML Use Case diagram of Papyrus / Graphviz neato layout [FSMvH10].



(d) KLoDD layouter on a simple actor-oriented dataflow diagram (cf. subsection 6.2.2)

Figure 4.6. Automatic layout for different editors with different layout algorithms.

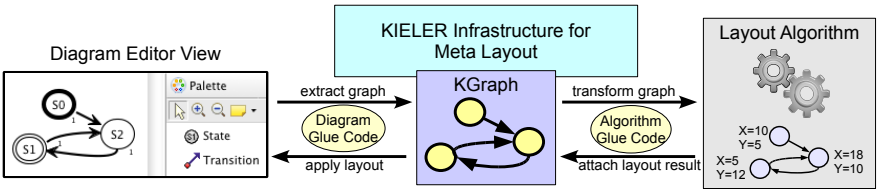


Figure 4.7. Overview of the KIML.

2. Pass the KGraph to the layout algorithm. Use algorithm library specific glue code to transform the graph into the internal data structure of the library.

3. Call the automatic layout algorithm which creates a layout result containing coordinates and sizes in its internal data structures.

4. Attach the layout result from the algorithm back to the KGraph.

5. Apply the layout result from the KGraph to the diagram.

Meta layout not only bridges between diagrams and layouters, it also tries to do this in a smart customizable way.

There are certain requirements in diagram syntaxes as well as certain limitations of layout algorithms that it tries to mitigate:

Advanced Diagram Syntax Diagrams may contain nodes, edges, multiple labels at all objects and ports [SFvHM10]. Algorithms might be limited to perform only layout for some of the elements, e. g., not support port constraints or labels.

⇒ Specify requirements and limitations explicitly for diagrams and algorithms. So for a concrete algorithm interface there is some meta information added about what features the algorithm supports, i. e., for what *kinds* of diagrams it is suited best. Vice versa, diagram editors can specify in meta information what kind of diagrams they provide. Parameters provided by algorithms can be made available, for example

Listing 4.1. Recursive Layout

```
1 algorithm recursiveLayout(G: compound graph, v: vertex in G)
2   for each child v_c of v:
3       recursiveLayout(G, v_c)
4   if v is not a leaf then
5       retrieve layout method A associated with v
6       set up A with layout options of v and its children
7       execute A on children of v with given configuration
8       set size of v to bounding box of layout of A plus insets
```

layout directions or seed values. Figure 4.5 shows a KIELER screenshot and shows the layout options for one selected diagram region.

Compound Graphs Diagrams can be compound, i. e., nodes may contain other nodes. This is typically seen for example in UML State Machines or Packages. Many algorithms do not work on compound graphs.

⇒ Apply layout recursively for compound graphs following Listing 4.1, starting with leaf nodes. This works for all layouters unless there are any hierarchy crossing edges, i. e., edges whose source and target node have different parents. Such cases require special treatment [SM91].

Complex Models One single layout algorithm might result suboptimal layouts for complex models.

⇒ Meta layout allows to use multiple different layout algorithms for different parts of one and the same view as shown in Figure 3.15. This is well suited for compound models.

In summary, meta layout bundles a set of layout algorithms together and matches them with concrete diagram syntaxes. It lets the user mix parameters and layouters to find the optimal layout result for custom model views.

KIML uses the Eclipse Modeling Framework (EMF) to specify abstract syntax and graphical editors generated with the Graphical Editing Framework (GEF) for concrete syntax as explained in Section 4.1. The generative

approach of GMF has a standard persistence handling of models and their views (the *notation model* in GMF terminology). KIML provides a generic implementation of the diagram glue code (Figure 4.7) for GEF/GMF that performs the following tasks:

- ▻ It extracts the graph structure from graphical GEF objects (so-called Edit Parts) into the KGraph.
- ▻ It provides a command to apply the layout results back to the diagram following the GEF request-command pattern. Therefore KIELER layout can be used with every GEF-compatible editor. This is non-trivial, because GEF is not designed for automatic manipulation of views but only for single-element manual user interaction. For example changing bend points of an edge in one single step together with the position of a corresponding edge label requires some low level manipulation.
- ▻ It exploits the style mechanism of the notation model of GMF to make user defined layout options persistent in a generic way for all GMF editors without introducing new files.

Hence, for most GMF editors KIELER automatic layout can be used out-of-the-box. Some of the glue-code is only GEF specific, however, the GEF and GMF runtime libraries are so deeply interlinked that some of the glue-code also depends on GMF, where GEF is too general. E. g., in GEF there is no standard way to distinguish nodes from ports.

Optionally the Eclipse extension point *layoutInfo* is used to specify default values for layout options, e. g., diagram types to setup default layout types. This has been done for example for the MDT/Papyrus UML suite [FSMvH10]. Such meta information in form of an XML extension specification (plugin.xml) is the only thing a tool smith needs to provide to give the user a smooth user experience with KIML and his or her diagram editor. For other concrete syntax frameworks based on GEF, like the Generic Eclipse Modeling System (GEMS), Marama or Autofocus, the glue code would have to be extended accordingly. Connecting Graphiti is ongoing work, which does not exploit the GEF background at all but works on a higher abstraction level.

For layout algorithm integration KIELER provides the extension point *layoutProvider*. There one specifies the layout options that the corresponding algorithm accepts and priorities for diagram types that it supports. The algorithm itself has to adhere to the following signature:

```
public abstract void doLayout(KNode, IKielerProgressMonitor)
```

The given KNode forms the layout problem in shape of a KGraph. It is already augmented with the layout information of the current view, i. e., the layout options and the current coordinates and sizes. The task of the method is to exchange these values by new ones that yield the layout result of the algorithm. If not specified otherwise, the given graph is only a flat subset of one hierarchy level of the possibly compound view. It is then applied recursively following Listing 4.1. Usually the processing will take a significant amount of time. Hence, with the IKielerProgressmonitor one can give some feedback to the user interface.

From this interface one can directly start to write a layouter in Java or add some glue code to bridge it to existing layout algorithm libraries. In order to give KIML some initial capabilities, it provides some libraries or bridge code for them:

- GraphViz [GN00], see subsection 6.2.1.
- OGDF [CG07] is a C++ project with many recent and sophisticated algorithms. It gets connected using the Java native interface with a customized C wrapper.
- Draw2D provides an "arrange all" functionality, which provides a simple layer based [STT81] approach. This is also bridged in pure Java to add the recursive functionality to this algorithm and to compare it to other libraries.
- With GEF comes the visualization toolkit Zest[5], which provides a set of rudimentary layout algorithms. A basic connection to Zest is implemented in KIML, however, the other algorithms usually give better layout results.

[5] http://www.eclipse.org/gef/zest/

▷ Also provided is the KLoDD, a customized layered layout algorithm that supports hyperedges and especially port constraints as a pure java implementation [SFvHM10].

KIML is designed to be an experimentation platform for layout algorithms. Instead of forcing the user to accept a fixed layout algorithm, it offers a flexible interface that allows to set different layout options for each diagram, or even for each part of a diagram. These options include the selection of a specific layout algorithm, which in turn can be contributed using Eclipse extension points.

Layout Procedure

Diagram editors in the Eclipse GMF are structured with the MVC paradigm (cf. Section 4.1): the *model* is built on an EMF structure that stores the semantic data, the *view* consists of *figures* that are used to draw the actual diagram, and the *controller* consists of a set of *edit parts* that are assigned to the model elements in order to control their visualization and editing. By exploiting the standardized structure of these edit parts, we are able to derive an annotated graph from any GMF diagram editor at runtime. Since diagram elements may be hierarchically structured, the resulting graph G is a *compound graph* $G = (V, H, E)$ with a set of vertices V, a set of *inclusion edges* H, and a set of *adjacency edges* E [SM91]. The *inclusion graph* (V, H) must form a tree, where each inclusion edge $(v_1, v_2) \in H$ is interpreted as v_1 *contains* v_2. The annotations that are attached to the elements of G are either layout options to select and configure layout algorithms or layout results such as the coordinates and size of each element.

We apply the following procedure to layout GMF diagrams:

1. Derive a compound graph G from the structure of edit parts of the diagram and add layout options as annotations. This involves the aforementioned diagram glue-code.

2. Recursively perform the layout on the inclusion tree of G, starting with the leaves, considering the individual layout algorithm associated with each inner vertex of the inclusion tree. The resulting object positions

are attached as annotations. Let v_0 be the root of the inclusion tree of G. Using Listing 4.1 on v_0, perform layout on each layer of the inclusion tree starting with the leaves.

3. Apply the layout results that are attached to the elements of G to the respective edit parts of the diagram.

These steps are loosely coupled such that intermediate tasks could be performed. For example after calculating the new layout in step 2, a new bigger bounding box is used to zoom out before the layout is actually applied and animated in step 3.

So far, layout has to be configured and triggered manually. The following section introduces the implementation of view management, which allows to remotely control KIML and automate the processes.

4.3 KIELER View Management (KiVi)

The *KIELER View Management* (KiVi) follows the ideas of view management presented in subsection 3.2.5. The basic concept is formed with the three different classes `Trigger`, `Effect` and `Combination`. Triggers are low-level events in the platform and effects are arbitrary graphical feedback features in a diagram such as auto-layout or applying filters.

Combinations — Abstraction and Genericity

A `Combination` is the logic that connects triggers with effects and therefore controls *what* is visible in a diagram *when* and *how*. The separation of concerns is done in order to get the highest abstraction level possible for the designer of a concrete *View Management Scheme* (VMS). By this the goal is to make view management "first-class citizen" in the modeling environment which can fully be controlled and customized by the user. We cannot assume that KIELER can provide all thinkable VMSs. Therefore KiVi wants to give the user the opportunity to program them by him- or herself. Hence, the main focus is to make the implementation of a VMS in shape of a `Combination` as abstract and simple as possible.

A `Combination` listens to the required `Triggers` employing the observer pattern [GHJV95]. Whenever a `Trigger` noticed some state change in the platform, it informs the listening `Combinations`, which then decide whether they want to execute any `Effects`. Following the observer pattern, a designer of a concrete `Combination` would have to register it as listener to all required `Triggers`. For performance reasons we do not want it to listen to *all* available `Triggers`.

The idea is that the KiVi runtime handles all this registering and bookkeeping for the user. Hence, the VMS designer does not need to follow the interface `ICombination` directly, but can use the abstract implementation `AbstractCombination`. This expects the designer to implement only one method, the `execute` method. It may implement an arbitrary number of parameters, all subclasses of ITriggerState. Here, some black magic in form of reflection comes in. The `AbstractCombination` determines the parameters of the `execute` method and by its type automatically registers

Listing 4.2. A minimal Combination

```
1 public class SelectionHighlightCombination
2                                         extends AbstractCombination {
3
4   public void execute(ButtonState button, SelectionState selection) {
5     if (button.get("myButton").isEnabled()) {
6       schedule(new HighlightEffect(selection.getEObjects()));
7     }
8   }
9 }
```

the `Combination` at the correct `Triggers`. By this, the designer can access the required platform state easily from these parameters and has already specified which `Triggers` are required to deliver the corresponding states.

An example minimal `Combination` is listed in Listing 4.2. It listens to a toolbar button and to the current selection in a diagram. When the button is pressed, the currently selected graphical elements will get highlighted in the graphical view.

With this concept, the body of the `execute` method is only aware of the trigger states and their explicitly provided parameters and effects that are available in the platform. Therefore it should be possible to even raise the abstraction level again. Ongoing work is to make it possible to implement such `Combination` in a simple scripting language such as Ruby or by other means such as a graphical model itself.

Triggers — Events vs. States

A `Trigger` implements the event notification mechanism of the observer pattern. A concrete `Trigger` usually listens to some event in the Eclipse platform or more specifically to the running diagram editor or its semantic model. When it receives such an event, it informs the listening `Combinations`. However, Eclipse usually provides *events* while it is more convenient in a `Combination` to work with *states*.

Take for example the artificial case of a button that is to be interpreted as

a toggle button. One push turns it on, the next turns it off. If the `Trigger` would simply submit the push-event, then the `Combination` would require to maintain a memory of the current state of the button. Therefore a `Trigger` is responsible to transform a system event into a `TriggerState`, which gets passed to the `Combinations`. This allows the `Combination` scheme outlined above, where the `execute` method can take multiple `TriggerStates`. Events usually do not occur at the same time, but KiVi will maintain the states internally and pass all required states together to a `Combination` when it needs to be re-evaluated. By this, the `execute` method can create a complex Boolean expression over multiple states.

Example Triggers are:

`ButtonTrigger` A user pushed a button that is dedicated to view management.

`SelectionTrigger` Elements in the model have been selected.

`ModelChangeTrigger` The model has changed and it might be necessary to update the view.

`LayoutTrigger` Automatic layout has just been performed. This trigger can be used to do other effects after layout.

`DataTrigger` Simulation trigger that informs about new data of an executing simulation. It contains values of variables.

`StateActivityTrigger` Simulation trigger that informs about the states that are currently active in a state machine model.

Especially the latter two are simulation triggers which are highly language- and simulator-dependent. Therefore new `Triggers` can be provided using an KiVi extension point.

Effects — Scheduling

An `Effect` is an arbitrary action that can be executed by a `Combination`, usually as a reaction to some `Triggers`. Most `Effects` process the view of a model. That is why the whole mechanism is called *view management*.

However, `Effects` could do anything, for example write into a log file or open a warning dialog to the user.

Some examples for possible `Effects` are:

`HighlightEffect` A diagram element is highlighted. Its border line width will be increased and drawn in a desired color. It could be used to give feedback about something that happened with the highlighted elements.

`LayoutEffect` The diagram is re-arranged with the current set of layout options that is attached to the diagram. With passing a parameter only parts of the diagram can be processed. This is useful to save resources if there are only local changes. Optionally, zoom-to-fit and an animated morphing of the new layout can be requested within the effect. This is done by default.

`SetLayoutOptionEffect` Set specific parameters for layout to a given graphical element. These can be the choice of layout algorithm itself and also specific parameters for an algorithm that are provided by KIML. By this, a `Combination` can implement a layout choosing strategy as demanded in subsection 3.2.6.

`ZoomEffect` The effect zooms the diagram to fit to the current size of the canvas. As parameter a zoom-target can be specified. It is not an absolute level like 100% or 50%, but a semantic zoom element, which should be visible after the zooming. Therefore, the effect will zoom the diagram to the target element bounds, and additionally pan to its current position.

`CollapseExpandEffect` The inner contents of a given diagram element get either collapsed or expanded. This can be used to alter the level of detail of the view of a model element. Usually a collapsed element can take smaller bounds and therefore a succeeding `LayoutEffect` can compact the view.

`FilterEffect` Graphical elements get completely hidden, resp. shown again. It can be used to specifically hide labels or complete nodes or edges.

ModelTransformationEffect A model transformation is executed on the model. The transformation itself as well as the target node can be chosen.

TextEffect Arbitrary text can be shown on the diagram. It can be used to annotate elements with text.

ArrowEffect An arrow is drawn from one diagram element to another one. It can be used to visualize additional relationships between model elements.

ScopeEffect A graphical oscilloscope can be attached to a view element. This is useful for representing data during an execution run of a functional model.

DataTokenEffect A graphical token element can be animated that represents a data token that flows along a connection.

`Effects` themselves have to be implemented in the platform in a rather low abstraction level. KiVi provides an extension point to contribute new effects. However, ideally, they get implemented once and can be re-used in a broad set of different `Combinations`. Internally, KiVi has to manage the *scheduling* of the effects to fulfill the following requirements:

Thread Access The thread from which the `Effect` gets executed must be right. Different kinds of effects demand different threads. For example most UI effects require to be executed in the one-and-only GUI-Thread of Eclipse's Standard Widget Toolkit (SWT). `Effects` on a semantic model must be executed in a transaction which shall not interfere with another transaction on the same model, e. g., a transaction where a `ModelChangeTrigger` is notified.

Order `Effects` have to be executed in correct order. A `Combination` should be able to specify in which order the `Effects` shall be executed. KiVi has to guarantee this order, even if the `Effects` themselves are executed in different threads, possibly asynchronously.

Local Redundancy Different `Combinations` might request similar `Effects` within a short period of time. For some `Effects` it is a waste of system resource to execute them twice. For example it would be redundant to execute a `LayoutEffect` on the whole view twice. Also to `undo` an `Effect` while to `execute` the same `Effect` is still on the queue will remove the `execute` of the effect. Therefore KiVi has to be aware of which `Effects` can be replaced by each other under which specific sets of effect parameters.

Global Redundancy Even when time has elapsed between `Effects`, it could be redundant to call the second `Effect`. Some `Effects` set a *state*, and it would be redundant to do that again until the state has changed again. For example the `HighlightEffect` changes attributes of a diagram element until it is undone. A similar `HighlightEffect` would just set these attributes to the same values.

Undo old Effects `Triggers` provide the new system state, which might indicate that some `Effect` is no longer valid and should be undone. Hence, the `Combination` needs to remember between which references it has drawn arrows. It needs to remove arrows which are no longer valid for the new system state.

Final Wrapup When a `Combination` is turned off, all `Effects` that have altered the view must be undone. Hence, 'undo' has to be performed locally in every execution of a `Combination` and globally when the `Combination` is no longer active.

To comply with these requirements, KiVi maintains two different worker threads that handle the scheduling of `Effects`. The `CombinationsWorker` has a queue of `TriggerStates` that recently have changed and are waiting to be processed by the listening `Combinations`. The sequence is shown in Figure 4.8 and as a simplified outline in Figure 4.9. To avoid redundancy, some objects have to be smartly merged. The new state of a `Trigger` gets merged with the global state of all `Triggers` when a `Combination` is triggered. A concrete `Combination` may schedule multiple new `Effects`, which have to be merged with undoing of old `Effects` and each other locally. Finally, the resulting list of `Effects` is passed to the `EffectsWorker`. It has

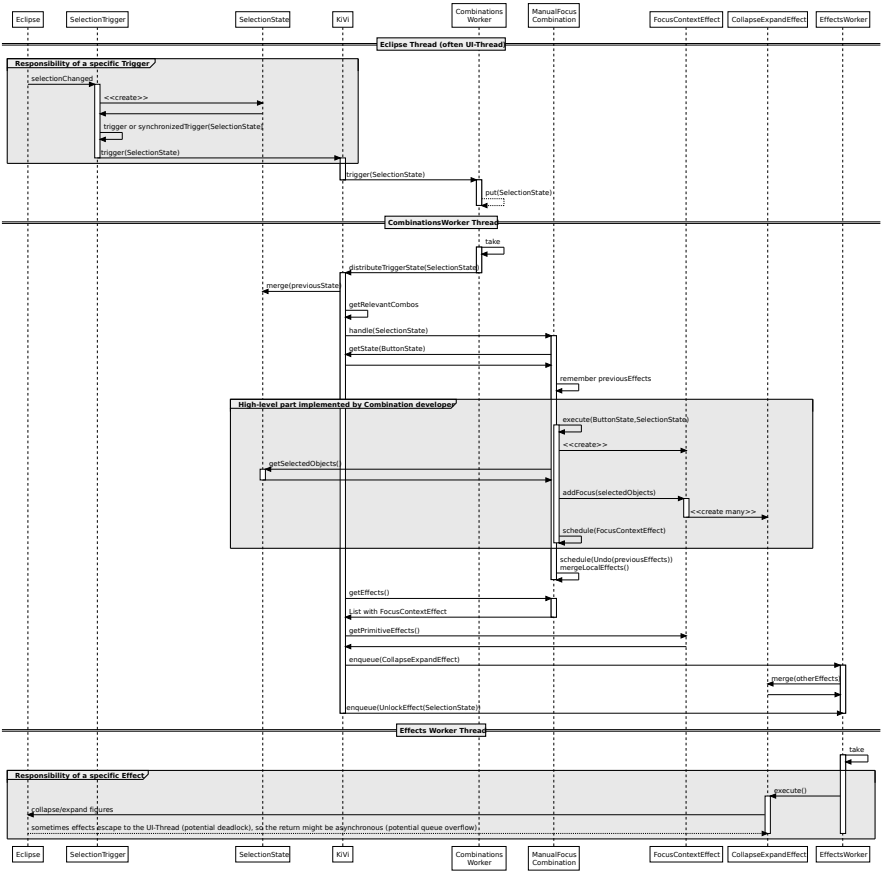


Figure 4.8. KiVi behavior showing the sequence for a `SelectionTrigger` calling a `Combination` that creates a focus & context `Effect`.

a queue of `Effects` that need to be executed. It takes care of scheduling the `Effects` in correct order and for replacing older `Effects` if they are mergeable. Therefore the effect designer can specify "mergeable" relations between specific effects through the `IEffects` interface.

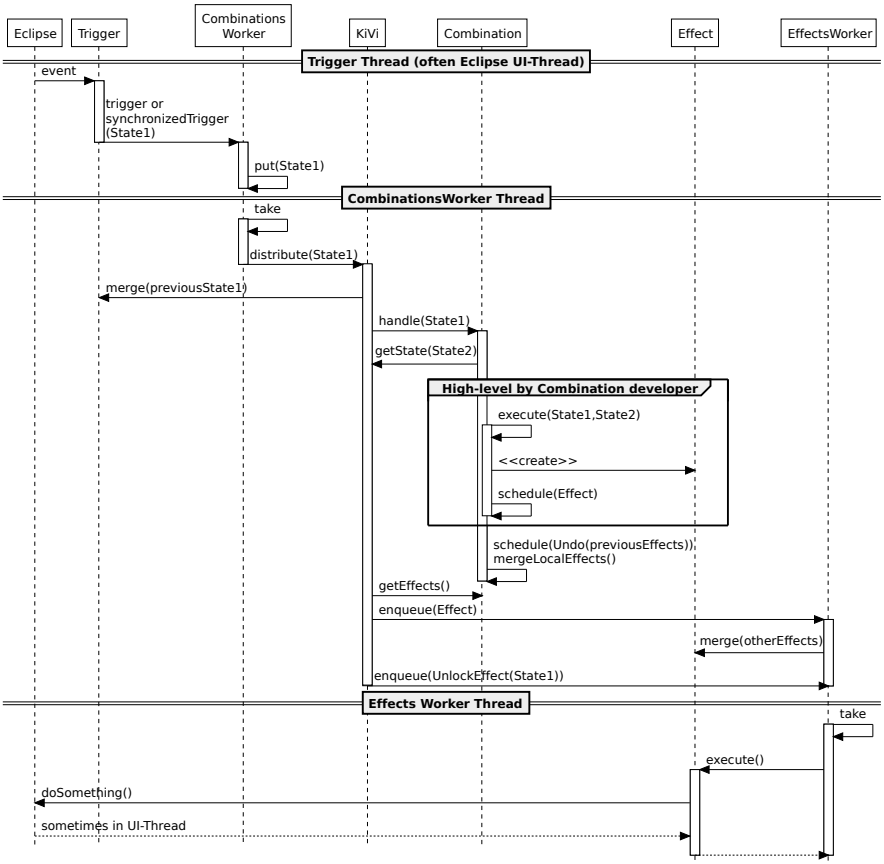


Figure 4.9. Simplified outline of the KiVi behavior.

Synchronization

It may happen that `Triggers` fire way too often and flood the system with too many events. In such case `Combinations` are also called very often and will flood the `EffectsWorker` queue with too many `Effects` and

the Effects queue can overflow, resp. Effects will not be executed in a reasonable time after the causing Trigger.

If it is known in advance that a Trigger might occur often, it can be synchronized with the Effects queue. Call AbstractTrigger.synchronizedTrigger(ITriggerState) instead of the normal trigger method in order to block the current thread until all Effects have been executed that have been caused by this triggering. This way back-pressure can be induced from Effects to the Triggers.

Technically the synchronized step calls wait() on the corresponding TriggerState. The CombinationWorker thread puts all Effects resulting from a Combination onto the EffectsWorker's queue. Afterwards it also puts an UnlockEffect onto the queue with the corresponding TriggerState as parameter (see Figure 4.8). The EffectsWorker thread eventually will take the UnlockEffect from the queue after all Effects have been executed and will notifyAll() who listen to the given TriggerState. This guarantees time-synchronization between the three involved threads, i. e., the Trigger will be blocked until the EffectsWorker has released the old TriggerState. Note that this may deadlock, e. g., if the Trigger executes in the UI thread, i. e., it blocks the UI thread and one Effect also executes in the UI thread (e. g., LayoutEffect). Then this effect will wait for the Trigger and the Trigger waits for completion of all Effects, which is a deadlock. As many Effects could arbitrarily work on the UI thread, avoid using synchronizedStep() in a Trigger that calls from the UI thread!

Example: The Dual Model

Chapter 3 presented the *dual model* on page 96. The idea is to visualize references between graphical objects that are not visible in the original syntax. This is for example the case in Statecharts, which use a broadcast mechanism of signals to communicate. The explicit data flow of signals is not visible. Transitions have references to signals, which get either emitted or received. However, it is difficult to see which sending transitions send data to which receiving transitions.

Implementing the dual model is straight-forward following the KiVi ap-

Listing 4.3. Pseudocode of the dual model VMS for SyncCharts.

```
1  class SignalFlowCombination extends AbstractCombination
2    execute(ButtonTriggerState button,
3            SelectionTriggerState selection)
4      scope = all signals
5      if selection contains signal declarations then
6        reduce scope only to selected signals
7      if selection contains transitions
8        reduce scope to referenced signals in selected transitions
9      if button is active then
10       foreach signal reference r1 in scope
11         foreach other reference r2 to same signal in whole model
12           if r1 is reading and r2 is writing reference then
13             schedule ArrowEffect r2 → r1
14           if r2 is reading and r1 is writing reference then
15             schedule ArrowEffect r1 → r2
```

proach explained above. The corresponding `SignalFlowCombination` uses a `ButtonTrigger` as well as the `SelectionTrigger` on the ThinKCharts editor (see Section 5.1). It employs the `ArrowEffect` to visualize the relations between signal references.

The combination in pseudo code is shown in Listing 4.3. The real Java code has to perform some more bookkeeping for cleaning up old arrows or reuse existing ones. This can be hidden in the concrete `Combination`, if some pre-defined bookkeeping code is used in the `AbstractCombination` that can assist with such usual cleanup tasks. Thus the implementation that has to be provided by the VMS designer can focus on the essentials.

The next sections discuss more complex examples and show how KiVi augments the simulation and editing process of SyncCharts.

4.4 Simulation with Focus & Context

One means to learn about the behavior of a SyncChart is to execute it stepwise while the simulation browser highlights active states. This paradigm is used by most state machine based tools like Matlab/Simulink/Stateflow of The Mathworks, Rhapsody of Telelogic/IBM, SCADE of Esterel Technologies or Ptolemy II of UC Berkeley. The usual means for navigation are panning, zooming and opening different parts of the model in different windows/canvases. However, for complex models it becomes difficult and effort-prone to manually navigate through a model. Debugging the application with one view gets difficult because it is hard to follow the advancements of state transitions when either (1) looking at the whole chart as an overview loosing details or (2) zooming into specific parts of the diagram loosing the context of the cutout. Figure 4.10 demonstrate this with an avionics application [FvH09a].

To alleviate this problem, KiVi synthesizes a new view on the model dynamically. Especially focus & context is helpful to present only the "interesting" parts of the model [PvH07]. For SyncCharts, a natural definition of "interesting" considers the currently *active* states, as illustrated in Figs. 4.11.

The `StateActivityTrigger` notifies the view management about changes in state activity during an interactive simulation. It connects to the KIEM (cf. subsection 6.1.1). The `HighlightEffect` then highlights active states. The `CompartmentCollapseExpandEffect` changes the level of detail at which the model objects get displayed in the view. In KiVi this is implemented by using GEF's method to fold compartments, which comprise the contents of states and parallel regions. Afterwards view management uses the `LayoutEffect` of KIML to rearrange all elements and zooms-to-fit to make best use of the given space. This unfolds the potentials of focus & context, as it presents all required details in the *focus* while still showing the direct neighbor inactive states collapsed with reduced detail level as the *context*. An animated morphing between the different views is provided to match the mental map of the user [RL08]. For an impression of this, the reader is referred to example videos on-line (or the KIELER tool itself).

The `StateActivityTrigger` not only provides the currently active states and taken transitions, but also a recent history, i. e., states that have

been active n steps ago. The history is provided by the execution manager and the amount of considered history steps can be configured through a property. The `Combination` sets all currently active state as well as all states in the history as focus. The `HighlightEffect` is used in different colors to indicate how long ago states and transitions have been active.

The best benefit of this `Combination` originates from the focus & context effects which leverages automatic layout to reduce the complexity of the diagram. Obviously a stable layout algorithm keeps the mental map better than an unstable one.

The next use case discusses the implementation of structure-based editing, which also benefits much from auto-layout.

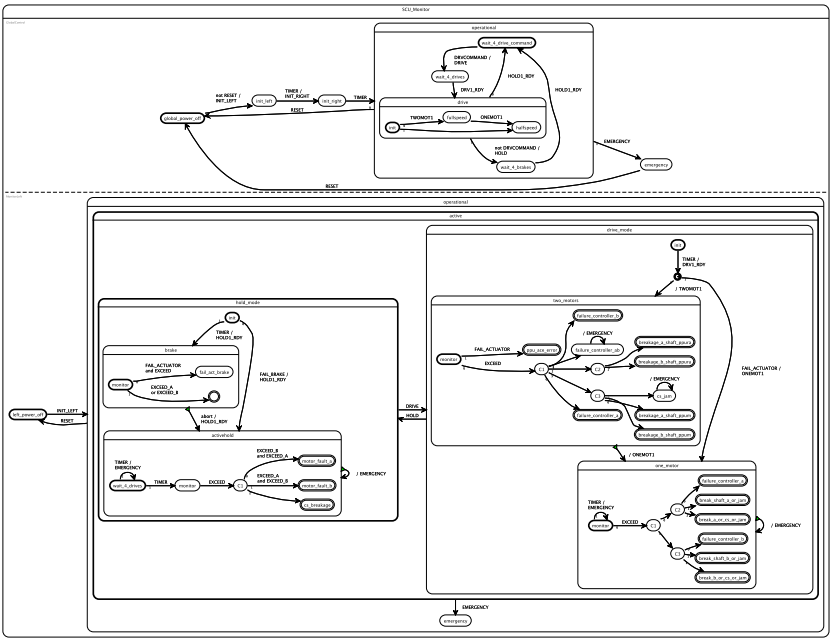
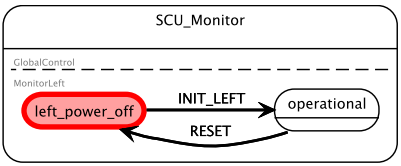
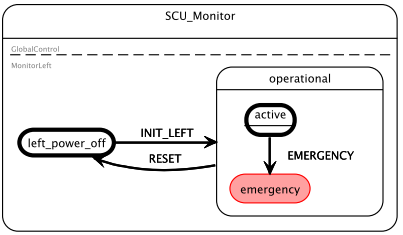


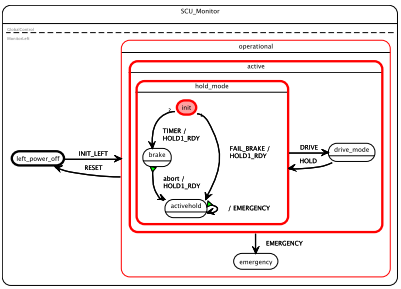
Figure 4.10. Avionics SyncCharts example: The whole model is too large to see all details without zooming.



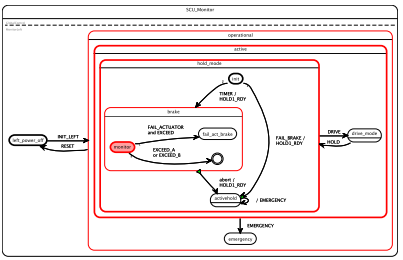
(a) Starting simulation collapses all inactive states and manually collapsed regions.



(b) Advancing simulation will always expand only active states with their full hierarchy.



(c) Active states are the *focus* and shown with full detail while inactive states are the context and their *contents* get collapsed.



(d) Even in deep hierarchy usually the full complexity of the model is hidden.

Figure 4.11. Focus & Context in a SyncChart

4.5 KIELER Structure-Based Editing (KSBasE)

The *KIELER Structure-Based Editing* (KSBasE) is the implementation of the editing proposals from subsection 3.3.1 on page 79. It uses model-to-model transformations and view synthesis with automatic layout. This results in means for creation and editing of diagrams with immediate graphical feedback but without the hassles of manual positioning.

The general implementation scope is shown in Figure 4.12. It has a core layer that connects user interaction with corresponding model-transformations and the view management. To the outside it interfaces with two entities: (1) the graphical editor and (2) the transformation engine.

Graphical Editors

KSBasE has to interface with the concrete diagram editor in Eclipse to offer a seamless user interface.

A helpful feature for many GMF-based editors is the `CanonicalEditPolicy`. It implements a standard synchronization between the model and the view (the so-called notation model). Synchronization here is limited to

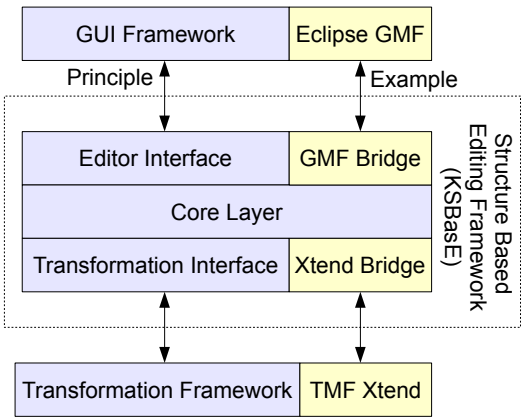


Figure 4.12. Scope of KSBasE

the following: whenever a semantic object has been created or was changed in the model, then a view object will be created or changed in the view. Freshly created view objects do not get useful initial positions, elements are just stacked on top of each other. Therefore this only technically solves the issue of initializing the view. KIML completes this feature quite naturally with automatic layout. Hence, KSBasE can be used with all graphical editors, that support some form of the `CanonicalEditPolicy`.

The set of pre-defined editing transformations is offered in the user interface in the main menu, toolbar, context menu, GMF's popup balloon menus and keyboard shortcuts as shown in Figure 4.13. The main menu holds all transformations for reference, while the others are context sensitive.

Transformations

Again, to be generic, KSBasE allows any Model-to-Model (M2M) transformation framework to be used with KSBasE. The rationale is the diverse situation in Eclipse: there are multiple transformation frameworks available such as Xpand/Xtend, Query/View/Transformations (QVT), and Atlas Transformation Language (ATL).[6] The corresponding transformation engine has to be wrapped in a `TransformationFactory`.

The current implementation provides such wrapper for Xtend, a functional language with good means to navigate models and handle lists conveniently.

Let us examine an example of an editing schema, which itself consists of another set of low-level editing steps. For a UML State Machine diagram, the schema to "add a composite state with initial contents" uses the following steps to achieve the result shown in Figure 4.14:

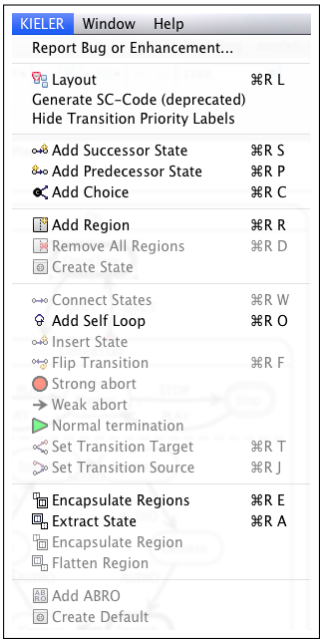
A. add a new composite state (in the Eclipse UML2Tools editor it already contains a region),

B. add a transition connector from original to the new composite state,

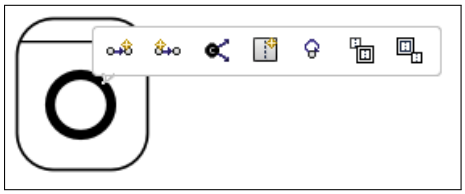
C. add an inner initial state,

D. add an inner state, and

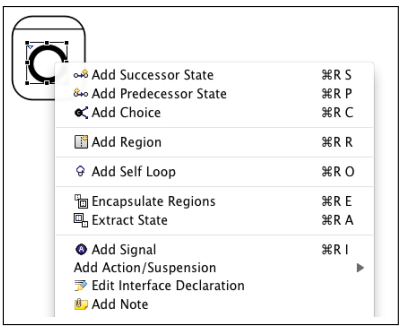
[6] All these M2M frameworks are linked at `http://www.eclipse.org/modeling/`



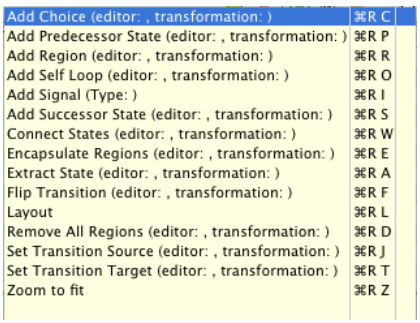
(a) Main Menu



(b) Popup Hover Menu



(c) Context Menu



(d) Keyboard Bindings

Figure 4.13. Different possible menus for KSBasE.

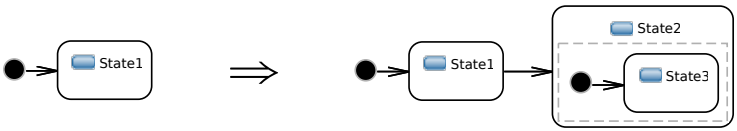


Figure 4.14. UML State Machine: Insertion of a hierarchical composite state.

Listing 4.4. Xtend transformation for insertion of a composite state

```
1  Void createComposite(State originalState):
2    let compState = new State:
3    let initState = new State:
4    let simpleState = new State:
5    let innerRegion = new Region:
6    let oTrans = new Transition:
7    let iTrans = new Transition:
8    let parentRegion = originalState.container:
9    parentRegion.transition.add(oTrans) ->
10   oTrans.setSource(originalState) ->
11   oTrans.setTarget(compState) ->
12   compState.region.add(innerRegion) ->
13   innerRegion.subvertex.add(initState) ->
14   innerRegion.subvertex.add(simpleState) ->
15   innerRegion.transition.add(iTrans) ->
16   iTrans.setSource(initState) ->
17   iTrans.setTarget(simpleState);
```

E. add a connector from initial to new state.

Listing 4.4 shows an in-place transformation in Xtend that specifies the editing scheme of Figure 4.14 discussed above. It is a pure semantic model transformation and does not involve graphical information.

Adding Xtend transformations and configuration of menu contributions for GMF editors is done through an Eclipse extension point.

KIELER provides a complete set of example transformations to edit SyncCharts. This is complete in the sense of fully covering the syntax of SyncCharts as well as giving some helpful “transformational sugar” to speedup

the editing process.

Key tasks of KSBasE are the following:

- ▻ Extract the current possibly heterogeneous selection of elements in the view and pass their semantic objects as parameters in the specified order of arguments of the chosen transformation function.
- ▻ Offer only those transformations in the user interface that are applicable for the current selection.
- ▻ Interact with view management to synthesize a new view after a transformation.

For the latter KSBasE provides a `ModelTransformationTrigger` that informs KiVi about changes in the model. Then a `KsbaseCombination` can schedule necessary `LayoutEffects`.

4.6 KIELER Textual Editing KiTE

Textual editing is well established means and has many advantages that we have discussed in subsection 3.3.3.

KIELER Textual Editing (KiTE) explores this in the context of SyncCharts. Conceptually we have seen different levels of integration between a diagram view and a textual view:

1. diagram initialization,

2. round-trip transformation,

3. full synchronization, and

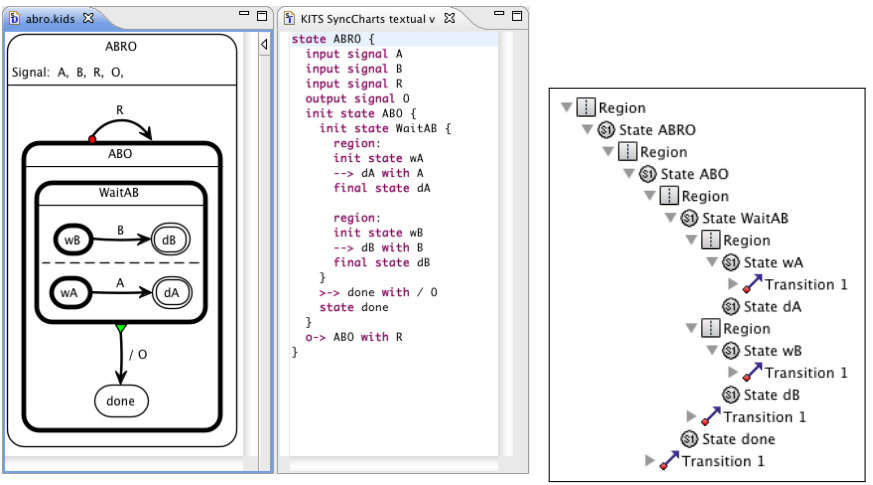
4. selective integration.

All of these require automatic layout of the diagram and therefore are not trivial to implement. However, with KIML we have the necessary toolkit at hand. Still, the pure technical synchronization has different problems that we will discuss in the following.

For the first level one uses a plain M2M transformation to create a model of the diagram, which then gets layouted. The second has to provide a binary transformation or two transformations in both directions. Still, the process is rather straight-forward. KiTE implements the third and partly the fourth.

For the diagrammatic part the GMF based ThinKCharts editor is used. For the textual representation KiTE employs Xtext. Both concrete editors are introduced in Section 5.1.

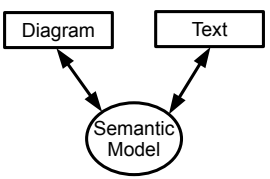
An immediate full synchronization requires to detect even small changes in either of the views, diagram or text. Then these changes have to be applied to the respective other. There are different strategies possible to fulfil this goal. The major difference is the way the semantic models are handled as shown in Figure 4.16. The 'Hello World!' program of SyncCharts—the `ABRO` model—is shown in Figure 4.15.



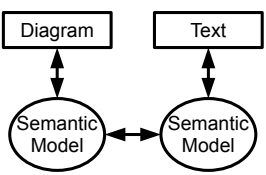
(a) KiTE showing both diagram and text.

(b) Tree of the underlying semantic model.

Figure 4.15. The ABRO model in KiTE.



(a) Views share the same semantic model.



(b) Each view uses own semantic model.

Figure 4.16. Synchronization of the views.

Shared Semantic Model In a first approach KiTE used the same semantic model instance for text and view [Bay09]. The synchronization itself works out-of-the-box due to integrated mechanisms both in GMF and in Xtext. Both frameworks are able to reflect changes in the semantic model to the

respective view. However, in the diagram the layout properties might have gotten invalid from such a *canonical editing operation* and therefore automatic layout was triggered to cleanup the diagram.

Major drawback is that these mechanisms only work on a resource base (e. g., the file) and only get synchronized when their semantic file is explicitly saved. Therefore the file has to be saved recently by the user and, hence, there is no real auto-synchronization.

Another drawback is that the respective other view is not aware of *what* has changed in the model and therefore cannot react on that. They simply get a complete new model and have to initialize it. Xtext serializes the whole model again and the current cursor and scrolling positions are lost.

Separate Models Another approach is to work on two separate semantic models. Each view has its own. The mechanism has to be extended in order to keep the both models in synch. Therefore listeners can be attached to both models to notify about any respective changes. In such case a M2M transformation is executed to transform the changed model to the other view's semantic model.

The advantage is that both models may correspond to different metamodels. Therefore the languages can be actually different. This can be used to slightly differ the metamodels to better fit to the requirements of the respective view. E. g., Xtext and GMF prefer slightly differently-structured metamodels. Alternatively this can be used for really different languages. For example SyncCharts can be transformed to the textual Esterel language and vice versa [PTvH06].

The main drawback still is that both model-to-view synchronizations are not aware of what exactly has changed in the models. A pure M2M transformation usually does not keep a trace from old to new model elements. Additionally Xtext has rigid policies about how to handle small changes in the text: it has a *partial parsing strategy*, which calculates the affected subtree of the parse model, removes the nodes in this position, and parses the subtree again. Effectively, it creates new elements from scratch with the changed attributes. On the other hand, GMF has its canonical edit policy that tries to keep semantic and notation model in perfect sync. Therefore GMF is not informed about an element change, but about invalid intermediate

steps where elements have been removed but not yet re-created. Hence, the built-in synchronization mechanisms do not fit well to interact this way.

Comparing Models Approach An extension, which is implemented in KiTE, also uses separate models. However, the models have to correspond to the same metamodel. This gets exploited to be able to *compare* the models with technical means. Therefore the EMFCompare framework is employed introduced in detail in Section 4.7. Given two model instances, it indicates which objects have changed, which are new and which have been removed. Especially it creates a trace between both models to relate corresponding objects.

This information is used to do the synchronization. The synchronization itself does not use an arbitrary M2M transformation. KiTE changes the elements explicitly with the knowledge of the compare model and limited to the corresponding scope. For example when changing a name of a state in a SyncCharts diagram, it can create a small change operation for the Xtext editor that is limited only to the scope of that state. Hence, Xtext does not have to re-parse the whole model again and therefore does not loose internal information about the editing history. The effect is a tight, seamless synchronization between an Xtext editor and the GMF editor.

Partial Models The idea behind Eclipse's distinction between *editor* and *view* is the following: An editor presents the contents of a file and a view presents some specific information about the file in the currently active editor and maybe even limited to a current selection in that editor. Handling two editors for conceptually the same model does not fit well to these Eclipse philosophies. Therefore the Xtext editor widget is put in an Eclipse view instead and will automatically try to synchronize with the currently active ThinKCharts editor.

This view and the synchronization is extended such that it listens to the current selection in the diagram and uses only this part of the model to fill its semantic model. Thus only the currently interesting model parts are presented to the user as text. It reduces complexity similarly as focus & context in the diagram.

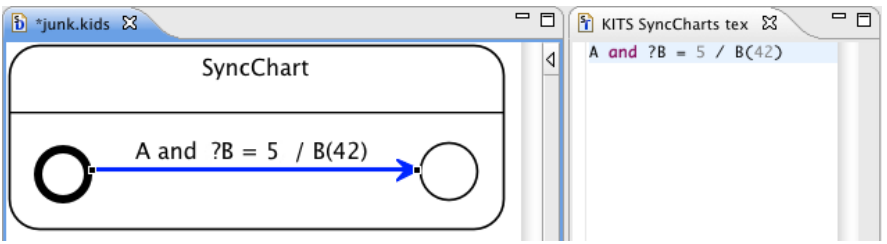


Figure 4.17. The textual view only serializes the current selection.

The main problems are missing references to elements outside the current selection and the fact that an Xtext editor only accepts elements of always the same type as root elements. For the first, a customized Xtext `ScopeProvider` passes references to unknown references in the linking step. For the second problem the Xtext editor has been modularized into multiple grammars and therefore single editors exist for the different parts. The text view then is able to use the corresponding editor for the respective model element. This is for example very useful to edit transition labels with full Xtext content-assist and auto-completion support as shown in Figure 4.17.

In summary KiTE offers full and even partial synchronization of diagram and text. Missing is only the integration of such view into the diagram itself. Eclipse has the concept of 'hover editors' that can carry arbitrary widgets and might be a first step when putting a partial text view into such hover editor. Another remark is that the SyncCharts Xtext grammar is tailored to the requirements of this synchronization and therefore the approach is only partly applicable for arbitrary DSLs.

4.7 KIELER Visual Kompare (KiViK)

As a proof of concept and to be actually able to test the visual comparison, it is implemented in the subproject *KIELER Visual Kompare* (KiViK). So far, the comparison is hard-coded into the general Eclipse infrastructure for resource comparison. It implements an *enhanced visual diff*, i. e., the *plain visual diff* with additional features added such as automatic layout, folding, zooming, and panning. This naturally takes advantage of hierarchies in a model; it detects when an internal element of a model has changed, but shows internal differences only when we are interested in them.

EMF Compare

As the main point of visual comparison is about the representation, existing supporting applications were used where applicable. Of particular benefit was the EMF Compare plug-in. This is a plug-in which extends the normal compare function of Eclipse by the support for EMF models. As depicted in Figure 4.18, first a *match engine* tries to find matches between the elements of the different versions with various metrics and computes a match model

[7] http://wiki.eclipse.org/index.php/EMF_Compare

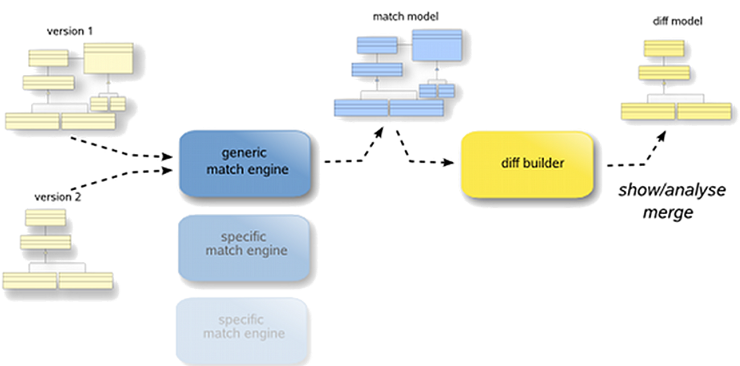


Figure 4.18. The structure of EMF Compare[7]

that is essentially a union of the compared models. Second, a *diff builder* extracts the differences into a *diff model*, which consists of *additions*, *deletions* and *changes*. The matching and differencing algorithm was inspired by work of Xing et al. [XS07]. Using EMF Compare, we can build on an established means to compute the differences, and can focus on just visualizing them.

Navigation and Visualization

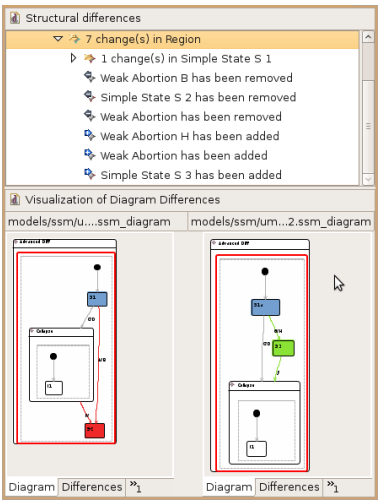
Figure 4.19 shows how the example diagrams from Figure 3.27 are compared by KiViK. We use the approach established by EMF Compare of side-by-side windows with the two versions of the diagram, with an additional third window on top that gives a structured textual description of the changes and guides the user through them.

It turned out to be very useful to provide the comparison tool with means to easily navigate through the changes. This was achieved by an adaptable click and zoom mechanism. Whenever clicking on an element in any of the windows—that includes also the top window with the textual description of the changes—, the other two windows scroll and zoom to the corresponding position in the diagram.

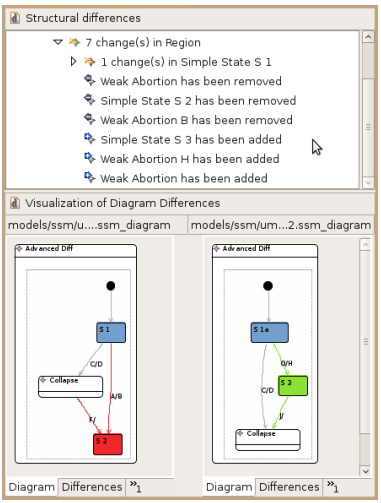
Another sensible option is to collapse the regions of the diagram which are not of interest. In the particular case of Statecharts, it is possible to collapse *composite states* in which no change occurred to gain more space during the diagram view, and to draw the users attention to the actual changes. This is shown in Figure 4.19b, in contrast to Figure 4.19a. Hence, focus & context is once again employed to cope with complex models. The automatic layout facilities of KIML are used to facilitate the space best.

The implementation is tightly integrated with the established Eclipse work flow. The visual comparison is launched when clicking on two diagram files, just as it is done with two textual files. It is also possible to compare a diagram file against its local history or the history in a version control system such as Subversion or CVS. The color scheme used to indicate additions, changes and deletions is the same as used by EMF Compare, see also Figure 3.28. This is consistent within Eclipse and can be customized by the user.

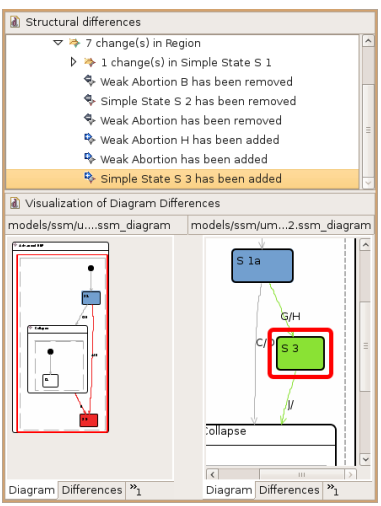
Automatic zooming and panning navigates the user to selected changes



(a) Collapsing disabled.



(b) State Collapse has been collapsed, as there are no changes inside.



(c) Auto-navigation zooms and scrolls to selected changes.

Figure 4.19. Enhanced visual comparison of two Statecharts.

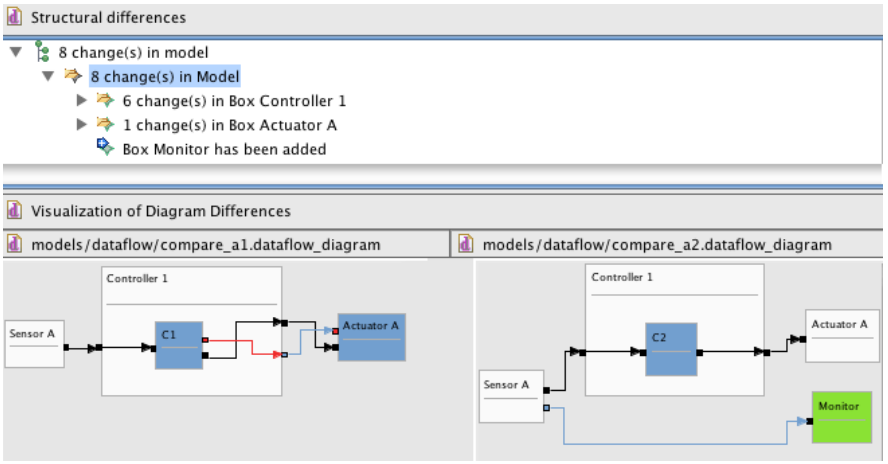


Figure 4.20. Visual comparison of data flow diagrams.

as depicted in Figure 4.19c. Changes can be either selected in the structural view or in the graphical views. The zoom level might be different in the two graphical representations to show the affected objects and their context.

A significant advantage of the generic EMF approach when working with models in the KIELER environment is that the visual comparison can be used with any EMF model, even with any editor, as long as it is generated with GMF. For example Figure 4.20 shows a comparison of two data flow diagrams as used by Matlab Simulink, Ptolemy or SCADE. They were created with a GMF editor and automatically layouted with KIELER. Moving from Statecharts to actor-oriented data flow diagrams did not require any extensions or modifications of the KiViK compare facility.

To provide facilities to compare diagrams graphically is a logical consequence of graphical modeling.

It should also be worthwhile to implement and evaluate some of the other proposals presented in Chapter 3, such as free and incremental merge. With well-designed incremental layout algorithms the incremental merge could be useful, especially if space is scarce and preserving the mental map

is crucial.

4.8 KIELER Environment Visualization (KEV)

A subproject for data visualization in a simulation run is the *KIELER Environment Visualization* (KEV). It differs form the use cases before, because it does not use the model diagram itself to present data but another view. Thus it is some kind of a multi-view model presentation that shows the system in yet another abstraction level.

The initial motivation for KEV arose from a bigger railway simulation application [HFPvH06], where the amount of data in the model is quite large. There is no feasible way to display the data properly in the model itself in a way that allows the user to grasp the internal state of the simulation. The application is a simulation of a railway installation with 48 track blocks and about 245 sensors and actuators. The main task is to implement a controller for the installation that controls multiple trains concurrently in a safe and fair manner. An environment simulation of the installation should help to validate a controller. To examine a simulation run, it is important for the developer to know the actual positions of different trains on the tracks. However, this information is hidden in a big set of arrays looped-through the data flow simulation model. This is hardly apt to procure the current train positions. However, this internal state can rather intuitively be understood by a human when the position coordinates get translated to a real track scheme of the installation.

For solving this task KEV was designed. Its main requirements were:

Graphical Data Display It should be able to display data by a graphical image that represents some real-world object. This is the case when a pure textual or scope visualization is too complex to be comprehensible.

Animation The view should be animated such that it can present the steps of a whole simulation run interactively.

Reusability It should be possible to reuse the framework. It should be used in other projects also, not only for the railway tracks.

Simplicity Creating graphics and animations should be simple. Users should not be required to learn a complex low level animation or graphics language such as Adobe Flash or Java 3D.

Open It should be possible to create such animations with free tools. Using specialized proprietary drawing environments should be avoided.

Some modeling environments offer to create advanced GUIs to present internal data. These comprise special "panels" where data can be summarized in multiple text labels, scopes, pointing scales or binary indicators. Other approaches allow to interconnect with special techniques such as Adobe flash or Java 3D.

In contrast, KEV uses an arbitrary Scalable Vector Graphics (SVG) image and maps simulation data to positions of graphical elements in the drawing. It fulfils the requirements, because it uses arbitrary drawings by definition, it updates the graphics every step to get an animation, the mapping between animation and simulation data is implemented in such generic fashion that it is easily reusable, and creating an SVG drawing can be done using any arbitrary SVG drawing tool.

SVG is an image standard for vector graphics of the World Wide Web Consortium (W3C) [182]. Next to pure graphics, it also offers a scripting language to implement simple animations. However, these animations follow this scheme: "Move object X from position A to position B during the next 2 seconds". SVG rendering engines have the task to execute such animation.

In an environment animation we cannot specify such animation durations. The simulation happens in small steps, where the step size might vary and usually is a small discretization of time. Therefore we can only map specific values of the simulation to specific positions in the graphics like this: "Move object X from position A to position B according to the simulation variable V that has a valid range from C to D".

Therefore KEV implements such mapping scheme. It uses an EMF model corresponding to the metamodel in Figure 4.21 as input that specifies a concrete mapping of simulation data to concrete animations.

Hence, next to the SVG image itself, the user needs to provide such animation mapping. It specifies the SVG file and addresses the concrete SVG element IDs for which animations are specified. An animation may be chosen from a set of pre-defined operations. This set can be extended but so far was sufficient for the considered examples:

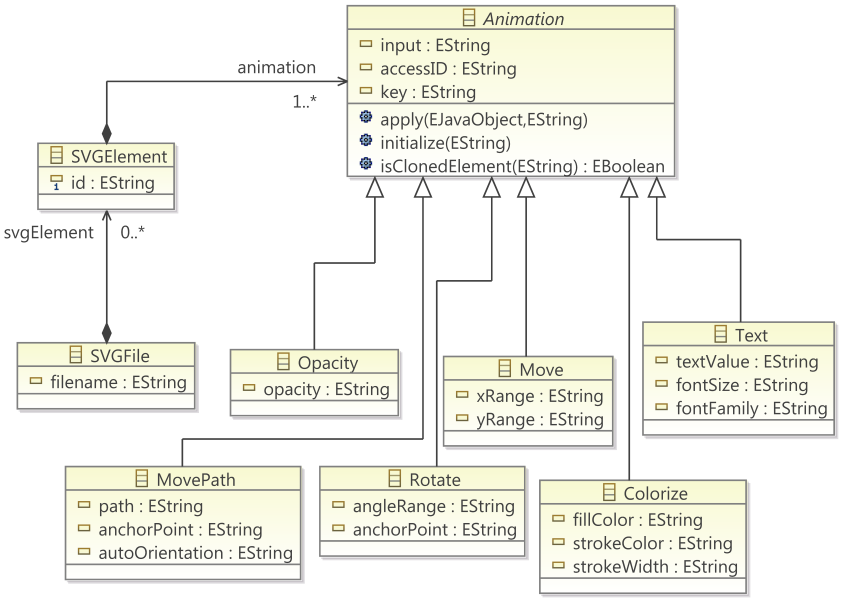


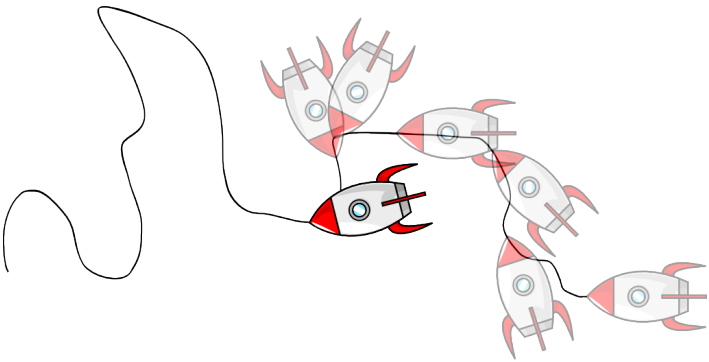
Figure 4.21. KEV Mapping Model

Colorize/Opacity The color and transparency of objects can be changed. Especially the opacity is useful to show or hide elements to represent boolean values.

Move/Rotate Elements can be moved along linear coordinates or rotated around an anchorPoint. With this pointing scales can be implemented.

MovePath Elements can be moved along an SVG path. Such a path can be drawn explicitly in a drawing tool and therefore is a convenient way to get complex move animations.

Text For convenience, text labels in the SVG drawing can be mapped directly to data values. Therefore some values can be displayed directly.



```
1   <mapping:SVGFile filename="resource:MovePathExample.svg">
2     <svgElement id="rocket">
3       <animation xsi:type="mapping:MovePath" input="1..360"
4                  path="path1" key="simulation_counter1"
5                  anchorPoint="0,5" autoOrientation="true" />
6     </svgElement>
7   </mapping:SVGFile>
```

Figure 4.22. A `MovePath` animation with its mapping: the rocket moves along the given path and automatically rotates accordingly.

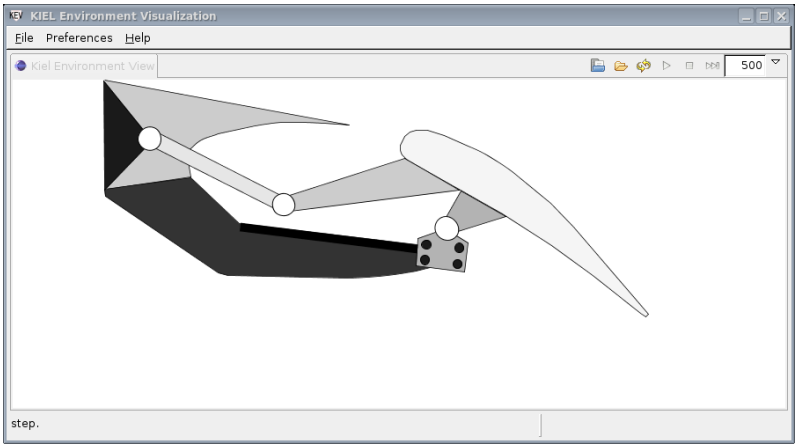
A `MovePath` example is shown in Figure 4.22 together with the corresponding mapping specification.

KEV is implemented as an Eclipse view that is connected to the KIELER Execution Manager (cf. subsection 6.1.1). It uses the Apache Batik SVG toolkit[8] to render the images. The process is rather straight-forward: Every simulation step, KEV receives new simulation data and thereupon updates the positions of the animated elements in the SVG drawing and repaints it. Thus the impression of a real animation occurs and always reflects the current state of the simulation.

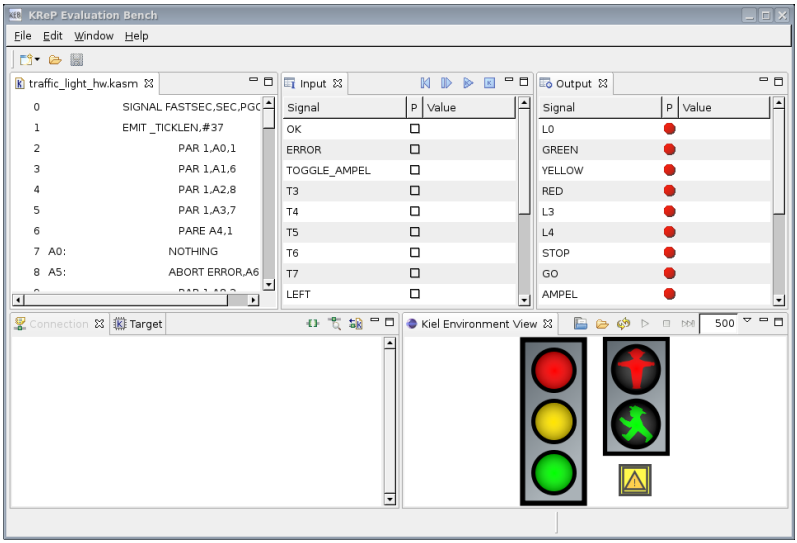
Some examples are shown in Figure 4.23.

The aforementioned railway application uses an SVG image showing the

[8] `http://xmlgraphics.apache.org/batik/`



(a) A high-lift flap panel with a rather complex animation from an aerospace application [FKRvH05]. KEV can be used in its own rich-client application.



(b) A simple traffic light visualization. The KEV view is embedded into another application for reactive processing [Tra10]

Figure 4.23. Example applications for KEV.

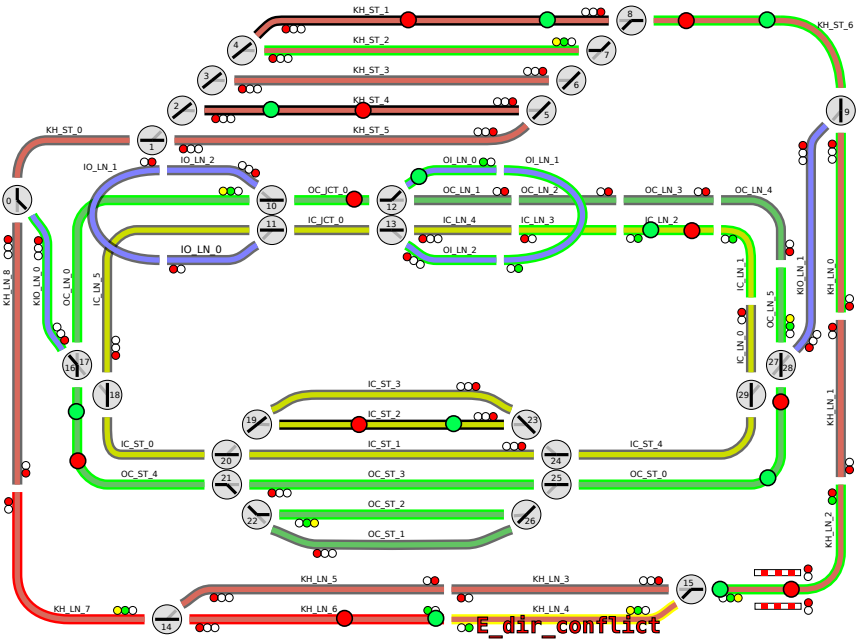


Figure 4.24. The railway application KEV animation.

track layout of the railway installation. It is shown during a simulation run in Figure 4.24. Each track block segment consists of a line and a border and itself defines an SVG path. The border's color is changed to indicate the current driving direction of trains. The paths are used to have a `MovePath` animation for every track segment which shows the current positions of the trains. Trains are simplified by a green and red circle to indicate the engine and the last wagon of the train. Switch points are small circles showing a little switch-symbol with a turn and a switch branch. The branch lines are separate SVG elements and use the `Opacity` animation to indicate the correct setting of a switch point. Positioned near to every track segment

there is a big red text label. In normal case they are hidden using the `Opacity` animation. However, if the simulation shows an error at a specific track, the text is shown and the error message is posted to that label using the `Text` animation.

In summary KEV implements a light-weight animation framework with open standard techniques that integrates with the rest of KIELER.

4.9 Evaluation

In this section we will discuss evaluation results, to assess the benefits of view management for model-based design. There was feedback collected for visual comparison and a primary study using structure-based editing. Both indicate the benefits of the approaches presented in this thesis. However, all-embracing detailed studies still have to be performed in the future.

4.9.1 Visual Comparison

Feedback of the work group and from an industrial development unit (Philips Medical Systems) that uses Statecharts gave a quite positive evaluation. The general implementation, the facility to perform a visual comparison as such, was well accepted. During the use some interesting benefits could be identified. Whereas simple structural changes of a diagram, such as removing or adding of a state, can be detected in a reasonable amount of time manually, especially for small changes the visual comparison was deemed very useful. Those small changes incorporate mainly the altering of attributes. On the one hand it were those of states, which in most cases are not shown in a diagram directly. On the other hand it were changes of the triggers and effects of a transition. Those changes manifest themselves just in different letters drawn next to a transition and are hard to recognize graphically.

The main benefit when showing these differences of the diagram was the way they were presented to the user. In former difference representations the changes were shown textually to the user, who then had to identify them in the diagram later on to interpret them and to deduce the functional differences. In this approach, the changes are instead directly mapped to the two diagrams.

Other aspects mentioned positively were the ability to collapse regions which were not of interest and the synchronous scrolling and zooming of the diagrams, i. e., the usage of focus & context.

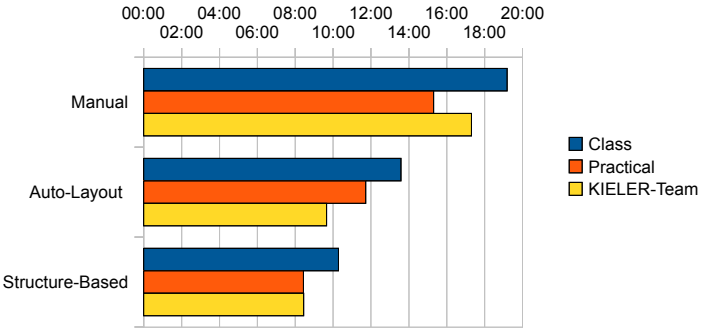


Figure 4.25. Evaluation of different editing methods, with average editing effort measured in minutes.

4.9.2 Structure-Based Editing

To assess the benefits of view management for model editing, we have conducted a study using KIELER. The hypothesis to be evaluated was that structure-based editing reduces the development times for creation and modification of graphical models significantly compared to usual WYSIWYG Drag-and-Drop (DND) editing. The 30 subjects divided into three different categories: The *class* group was familiar with the syntax of SyncCharts but not with modeling editors. The *practical* group took part in a practical course and had some experience already with Eclipse GMF editors. The last group comprised developers of the *KIELER team*, combining experiences with SyncCharts and the Eclipse SyncCharts editor.

The task was to create three different SyncCharts, using a different input method in random order for each: (1) standard Drag-and-Drop editing, (2) DND editing with manually triggered automatic layout and (3) structure-based editing as presented above. The models were provided in a comprehensible but formal textual notation. The experiment and its outcome are described in detail elsewhere [Mat10], but Figure 4.25 summarizes the results.

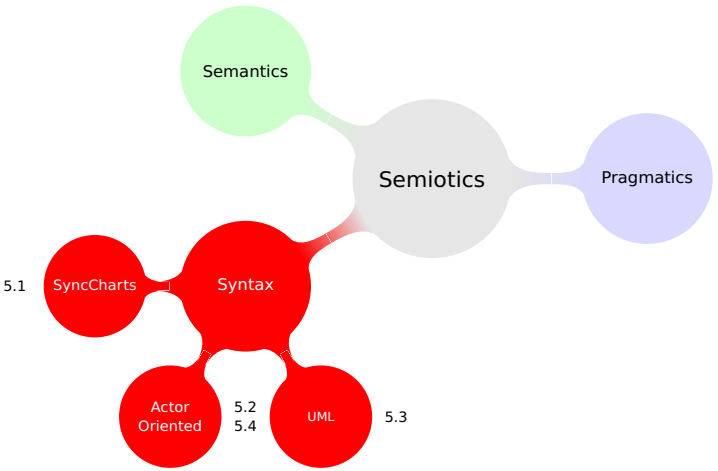
Editing with automatic layout decreased the necessary modeling times in average by nearly 33%. Full KSBasE reduced the times by another 15%

compared to DND. From auto-layout to KSBasE the difference was mainly influenced by the earlier experience, e. g., how well keyboard shortcuts could be employed.

The SyncCharts in the tasks were of rather simple structure, and only creation was required, no modifications. Hence, only rather plain transformations in KSBasE were necessary to complete the tasks. More complex transformations might result in even greater speedups.

Chapter 5

Case Studies with Concrete Editors



This chapter presents concrete diagram editors which employ techniques of KIELER introduced in Chapter 4. We will look at multiple different editors to evaluate the genericity of the approaches and their implementation.

The main demonstrator used in this thesis is a SyncCharts editor implemented with GMF (Section 5.1). It is tweaked to get basic requested features to put it to practice and not only use it as an demonstrator.

An editor for actor-oriented data flow models is used to evaluate the KIELER approaches on a different syntax that especially uses port constraints

as an advanced rendering feature (Section 5.2).

Some aspects of the UML get discussed to evaluate the applicability to languages with a complex predefined metamodel (Section 5.3).

Finally, we look into the Ptolemy II editor Vergil to validate especially the automatic layout interfaces of KIELER even beyond the scope of Eclipse (Section 5.4).

5.1 ThinKCharts

ThinKCharts is the Thin KIELER SyncCharts editor, see Figure 5.1 for a screenshot. It was developed as demonstrator for the KIELER approaches presented in this thesis. Additionally it is employed in teaching at the RTSYS Group. SyncCharts is a Statecharts dialect with synchronous semantics [And03]. Statecharts are Mealy Machines with parallel regions—to avoid state explosion—and composite states—for modularity and to avoid transition explosion. Signal broadcast allows communication between parallel regions and data adds support for complex calculations.

The concrete syntax of ThinKCharts follows André [And03] and the commercial tools Esterel-Studio and SCADE of Esterel Technologies. An overview is given in Figure 5.2.

5.1.1 Metamodels

The implementation uses GMF, already introduced in Section 4.1. The most important core in every GMF editor is the underlying metamodel that defines the *abstract* syntax of the DSL. Defining a metamodel is not trivial because it requires many design decisions about the abstraction level that should be covered by the metamodel. The technical requirements of such a metamodel are the following:

Machine Processability The metamodel should be understood by the editor generator frameworks. Different abstraction levels give different levels in which the framework can use the given data. We will see an example below.

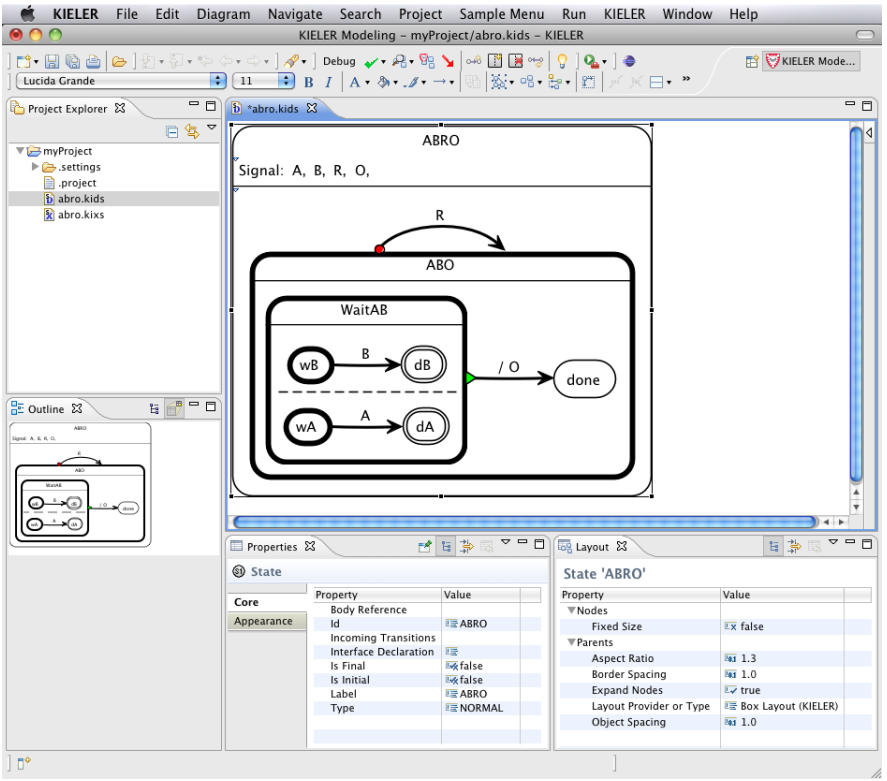


Figure 5.1. The Thin KIELER SyncCharts Editor (ThinKCharts)

Reusability Although metamodels for DSLs should be redefined for each DSL by definition, it can still save a lot of time and effort if parts of metamodels are designed to be reusable. This suggests modularization of metamodels.

Complexity The metamodel should not get too big. Complexity makes processing of model instances more difficult, effort- and error-prone.

Comprehensibility The ideas behind the metamodel should be well under-

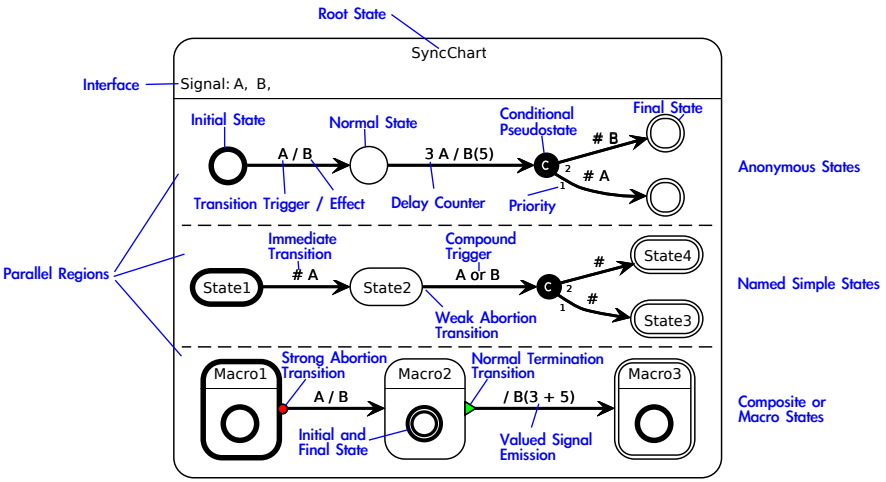
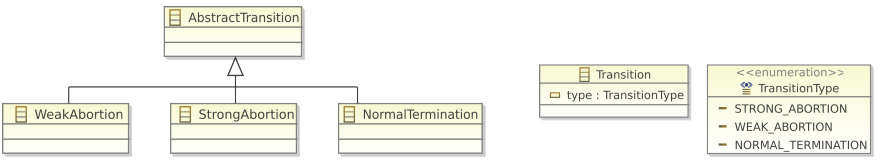


Figure 5.2. Overview of the concrete syntax of ThinKCharts.

standable by humans. Either when designing or enhancing a language or when teaching the semantics of the language, it can be very helpful to use the metamodel as a base for discussion. Therefore the metamodel should be very clear, small and well documented.

Especially the first and the last requirement are in conflict to some extent. We will look at SyncCharts transitions as an example. There are three kinds of transitions: Weak Abortion Transitions, Strong Abortion Transitions and Normal Termination Transitions. They have different semantic meaning as well as a slightly different concrete syntax. Therefore these differences must be considered in the metamodel. However, there are multiple ways to define it in the class model, as shown in Figure 5.3. One can either use a common abstract base class or interface and create multiple subclasses using inheritance. Alternatively, one can use only one class and add an attribute that holds the information about the type for example by an enumeration type.



(a) Inheritance **(b)** Attribute

Figure 5.3. Different ways to specify different types of a class.

The latter option will usually result in much smaller and clearer metamodels using less classes. Therefore the requirements Complexity and Comprehensibility are fulfilled. However, code generation frameworks as used in GMF prefer the inheritance method, because they choose code generation templates by pattern matching over classes. Still, with some experience of the evolution of the SyncCharts metamodel, its main focus is Comprehensibility while taking some additional efforts into account when developing tools around it. Hence, the metamodel uses a few classes with more attributes. This is in contrast, for example, to the UML metamodel, discussed in Section 5.3, which uses many classes with much inheritance that leads to a very big, difficult to understand metamodel.

Following the Reusability requirement, the SyncCharts metamodel is modularized into three different smaller models.

1. The the `Annotations` metamodel is shown in Figure 5.4. It is simply a means to *annotate* annotatable objects with arbitrary information. This is useful to extend models later on with information without the need to extend the metamodel. This annotation approach is reused in most other KIELER models, such as KIELER Actor-Oriented Modeling (KAOM) (introduced in the next section) and related synchronous languages.

2. The next important model is the `Expressions` metamodel, shown in Figure 5.5. It defines the core for communication in synchronous languages, i. e., data and expressions over data. Its main classes are `Signal` and `Variable` that synchronous languages distinguish. Additionally it defines arbitrary `Expressions` including complex `OperatorExpressions`

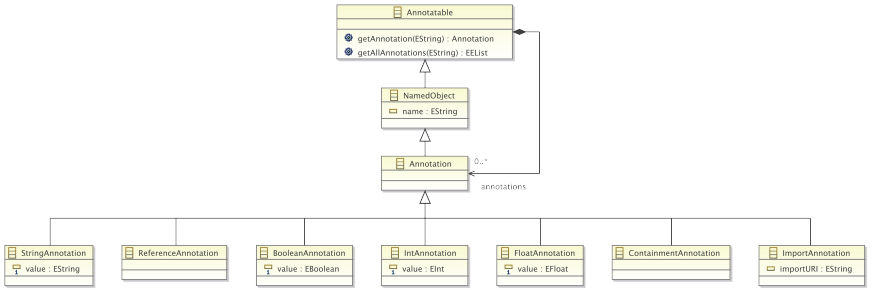


Figure 5.4. The Annotations metamodel

to combine signals in triggers with Boolean operators or to specify arithmetic expressions for valued signals or variables. This metamodel is used for SyncCharts as well as for an Esterel grammar and the *Synchronous* (S) language, an abstraction in EMF of the *Synchronous C* (SC) language [vH09].

3. The main metamodel for SyncCharts is shown in Figure 5.6. It defines ten classes itself and references some of the classes of the aforementioned models. Its main classes are `State` and `Region` with the common superclass `Scope` and `Transition` with its superclass `Action` that defines guard triggers and transition effects. A detailed documentation about all classes of all these metamodels can be found in the corresponding API on-line.

5.1.2 Graphical Editor

In an optimal scenario the GMF editor creation process should follow these steps:

1. Provide an EMF metamodel.
2. Create a `Graph` model that contains the graphical specification of the concrete syntax. It holds references about available primitive figures

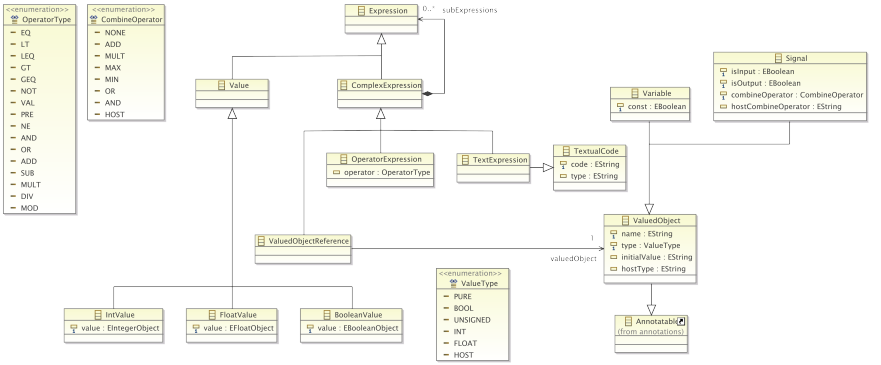


Figure 5.5. The Expressions metamodel

such as rectangles, circles and polylines and allows to customize certain properties such as colors, lines, decorators, etc.

3. Create a `Tooling` model that provides names and icons for tools in the palette toolbar with which the user is expected to interact for editing later on.

4. Combine the metamodel, the `Graph` and `Tooling` model in a so-called `Map` model, which interconnects abstract and concrete syntax.

5. Then with one button press, a `Gen` model is synthesized. It is created automatically with a GMF-internal M2M transformation from the `Map` model.

6. With another press of a button, code generation is triggered that will create a new Eclipse project and generate several Java source code files as well as all required files with meta-information to form an Eclipse plug-in.

7. Together with the GEF/GMF runtime plug-ins, this new plug-in can be deployed in an Eclipse instance, and the editor will be ready for use.

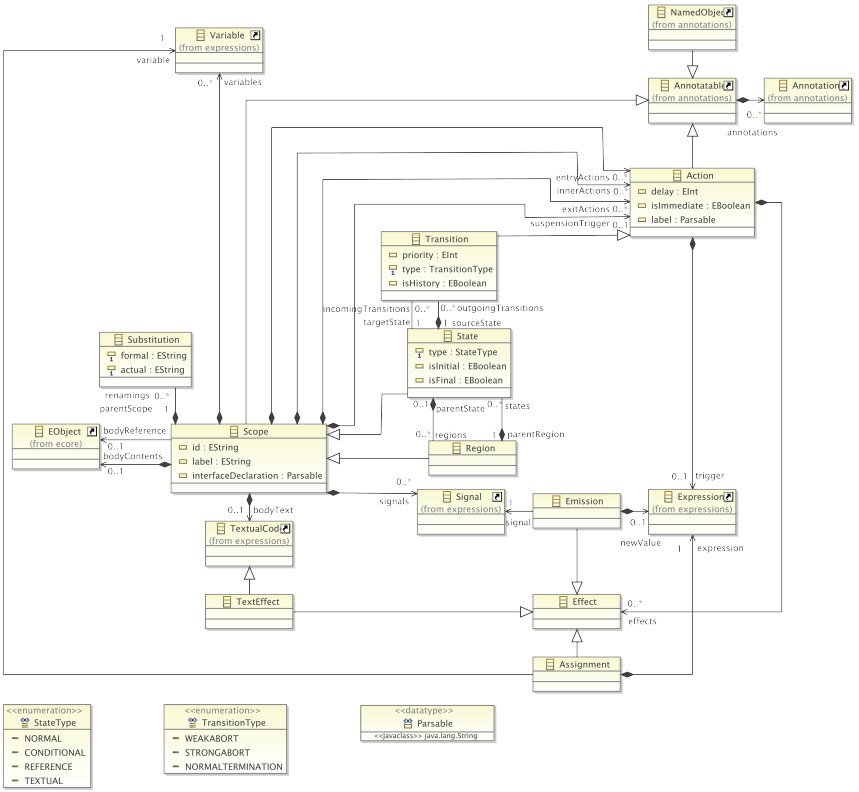


Figure 5.6. The SyncCharts metamodel

It is common practice for code generation that users work with a model on a high abstraction level while code generation templates use a model on a lower abstraction level that is optimized for code generation. This is the justification for step 5.

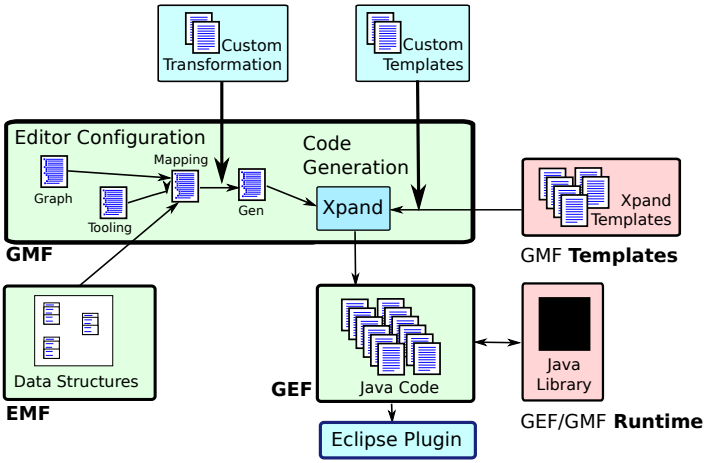


Figure 5.7. Customization in the GMF editor creation process.

With this process the final editor will be a freehand Drag-and-Drop (DND) editor with many common useful features, such as alignment tools, a properties view where colors and fonts can be configured and so on.

However, the usual case is that there are additional requirements to the editor that are not covered by this default implementation. This does not refer to the KIELER approaches presented in the chapters above—they are handled internally and will be available to all GMF editors by default, as discussed in the respective sections. There are usually smaller nuisances with the editor rendering or behavior that the toolsmith has to fix. Hence, the editor needs to be *customized*, which is a very effort-prone process in GMF. The SyncCharts editor is customized and uses different ways to do it. I extensively will discuss a representative subset of the customization options and give examples.

One way to customize a generated editor is to manually edit the generated code. This is even supported by GMF, because generated classes and even single methods are tagged with comments. If such comment tag indicates that a class or method is customized, then GMF will not overwrite

the corresponding code section. However, this feature will break when class or method names get changed and therefore is rather brittle and is discouraged. It is just the same with the `Gen` model, which is generated from the `Map` model. Its changes are not guaranteed to persist.

Other ways for customization are outlined in Figure 5.7. The generated code derives its properties from the GMF templates. They in turn use the information from the `Gen` model to make conditional code generation. The `Gen` model is transformed from the `Map` model. Therefore, following a purely generative approach, places for customization are either the `Map`-to-`Gen` transformation or the code generation templates, steps 5 and 6 from above.

GMF uses QVTo[1]—a standard for model transformations from the OMG—for the `Map`-to-`Gen` transformation. It has an explicit post-transformation hook, where a custom transformation on the `Gen` model can be registered. This is used to change rather high-level information such as additional plug-in dependencies, version number, name of plug-in and of the contributor.

For code generation GMF employs the Xpand template language.[2] It itself supports *aspect orientation*, which allows to override specific template rules by custom rules. These custom rules can generate any arbitrary different code from the originals. However, each custom rule needs to be self-contained. This means it needs to override the complete original rule and cannot use any other rules or functions from the original templates. This is a problem when the original templates contain very large rules which use a lot of functions. Then a big chunk of a template has to be copied and overridden, even if only a tiny part is to be changed.

Two examples for customization are presented in the following.

Trigger Listener

A GMF-based editor in general is a direct presentation of its semantic model instance. Editing something in the diagram will directly be edited in the semantic model if applicable. Primitive editing operations only correspond to single semantic elements by default. For example, when renaming a state

[1] `http://www.eclipse.org/m2m`

[2] `http://www.eclipse.org/modeling/m2t`

in a SynchChart, only its `label` gets updated. However, in our ThinKCharts there is a semantic relationship between the `label` attribute and the `ID` attribute. When a label is changed, then also its identifier should change to a valid identifier, unique in the region but deduced from the label. Another case are transition priorities. There are specific rules about valid SyncCharts: priorities of outgoing transitions have to be distinct and in a continuous sequence starting with 1. Thus, editing the `priority` property of one transition always creates an invalid intermediate model and the user has to fix the remaining priorities to repair it. An automated process could assist the user and re-order the remaining priorities.

There are many such use-cases and there is only little that GEF/GMF has to offer to solve them conveniently. Therefore ThinKCharts offer an observer mechanism in form of an extension point. Custom observer classes can be registered and then will be informed about any model changes that have occurred in the model. Such an observer then can react on a change by scheduling another change itself. It just has to guarantee that it avoids endless loops of changes and reactions.

The listening mechanism itself is registered at the resources of the editor input files. While such a `ResourceSetListener` or the more specialized `TriggerListener` is an existing Eclipse feature, it is not yet used by the default generated code. Therefore the code generation is customized such that it registers the listeners as `TriggerListener` every time an editor opens a model.

Attribute Aware Rendering

The above outlined the problem of metamodel design decisions. The SyncCharts metamodel contains few classes with some attributes and the latter indicate different types of states, transitions and so on. However, until version 2.2 GMF had only a simple strategy of mapping between concrete syntax and abstract syntax: each class in the metamodel corresponds to exactly one graphical figure in the concrete syntax. For SyncCharts this does not match: a `State` instance corresponds to a rounded rectangle, though it can have either a thin border (normal state), a thick border (initial state), a double circled border (final state) or a double circled thick border (initial

and final state), see also Figure 5.2. The same way transitions are rendered differently: normal termination, strong and weak abortion and the history attribute all induce different looks but are not reflected by different classes.

Therefore ThinKCharts extends the simple class-figure mapping by adding a customized figure that is able to alter its rendering. This *attribute aware* rendering uses listeners to model attribute changes. It connects conditions over model attributes with concrete figures that should be used for rendering the element. Thus a thick bordered state figure gets replaced by a double circled one when the Boolean flags `isInitial` and `isFinal` change accordingly.

This mechanism is injected into a GMF editor by changing figure constructors specified in the `Gen` model. Therefore a custom `Map`-to-`Gen` transformation is used to apply this customization automatically during the transformation process.

5.1.3 KIELER Integration

The KIELER features presented in Chapter 4 are integrated into the ThinKCharts editor straight-forward as described in the respective sections. It supports structure-based editing, whose configuration benefits from the clear metamodel. Automatic layout with KIML works well, due to the compound graph structure without inter-level transitions in SyncCharts. This is again in contrast to the UML State Machines, which make use of inter-level transitions due to the lack of a normal termination transition type. View management triggers and effects offer the focus & context technique, which is very useful during simulation, at the dual model, with textual editing and comparison.

5.2 Actor-Oriented Data Flow

Data flow models are often used to model control loops in the control engineering domain. Well known and established tools are the aforementioned Matlab/Simulink, SCADE, LabVIEW and Ptolemy II. With the latter the term *actor-oriented* was introduced in contrast to the well-known *object-oriented* term, to indicate the focus on concurrent data exchange instead of objects [Lee03a].

The *KIELER Actor-Oriented Modeling* (KAOM) subproject aims at bringing the data flow rendering to Eclipse, to harness KIELER support for automatic layout and focus & context use cases. So far, it is only a testbed for some of the KIELER approaches and not yet a full data flow modeling environment as the aforementioned tools.

A screenshot of a KAOM viewer is shown in Figure 5.8 and its metamodel in Figure 5.9. The metamodel is rather simple and small. Its requirement was to be generic and not to implement a specific syntax of one of the corresponding tools. Those have all different internal abstract syntaxes and KAOM should not be tied to a specific one. Therefore it follows the basic ideas of the Ptolemy II abstract syntax [Lee03b] to be as generic as possible.

The main object in KAOM is an `Entity`. It contains other entities and thus creates hierarchy. It contains `Link`s and `Port`s. The latter is a major difference to Statecharts, because communication is done through these ports. An `Entity` can also contain `Relations`, which are some kind of dummy-nodes to represent hyper-edges: one `Port` may be connected to multiple other `Ports`. Therefore multiple binary `Link`s can be used to connect multiple ports with the help of `Relation`s. A `Link` connects two `Linkable`s. The latter is a superclass for `Port` as well as for `Relation` and even `Entity` itself. Therefore technically the metamodel allows to connect `Entity` objects directly without the usage of `Port`s. Thus KAOM keeps its generic structure as it itself does not require to adhere to the port-pattern. `Entity` objects can also be interpreted differently to mix port-based with port-less syntaxes. This aims for example at integrating state-based control flow with actor-based data flow which is also done in Ptolemy II.

All classes inherit from `NamedObject`, which is imported from the `Annotations` metamodel introduced in the section before. It defines a

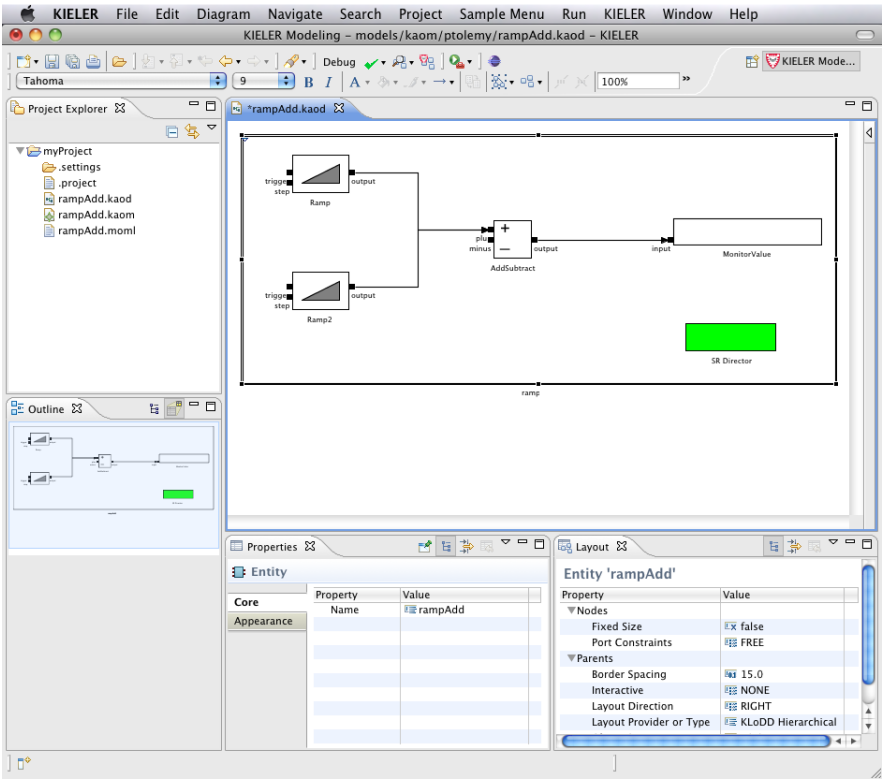


Figure 5.8. The KIELER Actor-Oriented Modeling Editor (KAOM)

name attribute for each element but, more importantly, introduces annotatability to all objects. This is the key aspect that helps to keep the metamodel generic. These few classes only define a very abstract structure of a model. They just specify how entities are nested and interconnected. So far, such model does not state anything about a more concrete semantics of the language.

Only by *annotating* `Entity` objects with corresponding information, they collect knowledge about what they really represent. For example the

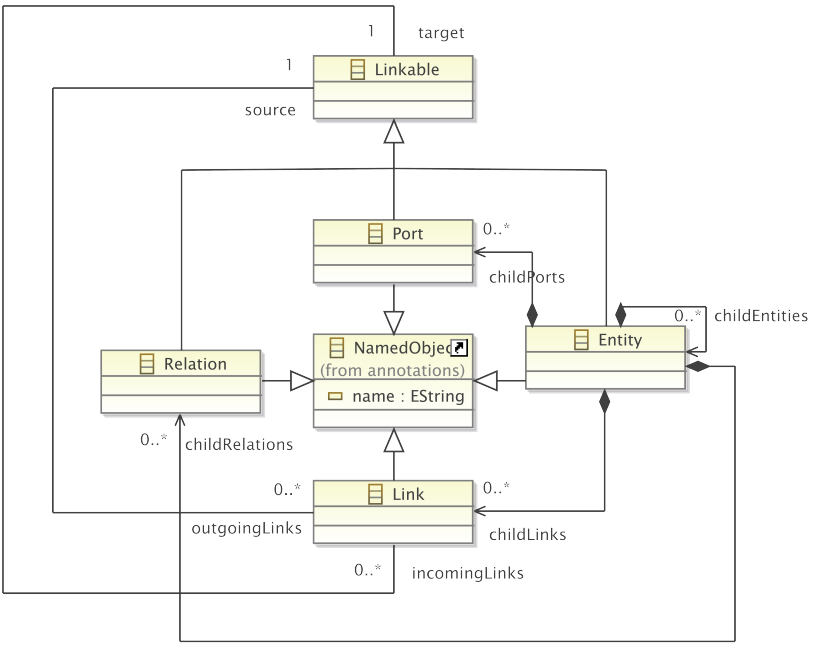


Figure 5.9. The KAOM metamodel

example model in Figure 5.8 shows a simple Ptolemy II model with some `Ramp` actors. From the Ptolemy model a M2M transformation converts a Ptolemy Ramp into a KAOM Ramp like shown in Listing 5.1.

The KAOM Ramp (starting at line 6) seems to be much more verbose than the three lines of an XML tree of the Ptolemy Ramp. One difference is that Ptolemy's elements have usually a `name` and a `class` attribute next to some possible `value`. The latter two must be expressed in single KAOM annotations and each annotation only has a `key` and a `value`. Therefore multiple annotations have to be nested to represent a Ptolemy class and a value. Additionally, Ptolemy uses much implicit information which is given

Listing 5.1. From a Ptolemy Ramp to a KAOM Ramp actor

```
<!-- Ptolemy Ramp -->
<entity name="Ramp" class="ptolemy.actor.lib.Ramp">
  <property name="_location"
            class="ptolemy.kernel.util.Location"
            value="{55,175}"/>
</entity>

<!-- KAOM Ramp -->
<childEntities name="Ramp">
  <annotations xsi:type="ann:StringAnnotation"
               name="ptolemyClass" value="ptolemy.actor.lib.Ramp"/>
  <annotations xsi:type="ann:StringAnnotation"
               name="_location" value="{55, 175}">
     <annotations xsi:type="ann:StringAnnotation" name="ptolemyClass"
                  value="ptolemy.kernel.util.Location"/>
     </annotations>
  <childPorts name="output"
              outgoingLinks="//@childEntities.0/@childLinks.0">
    <annotations name="output"/>
  </childPorts>
  <childPorts name="trigger">
    <annotations name="input"/>
  </childPorts>
  <childPorts name="step">
    <annotations name="input"/>
    <annotations xsi:type="ann:StringAnnotation"
                 name="tokenConsumptionRate">
       <annotations xsi:type="ann:StringAnnotation"
                    name="ptolemyClass"
                    value="ptolemy.data.expr.Parameter"/>
    </annotations>
  </childPorts>
</childEntities>
```

by the `class` of an Ptolemy `entity`. For example all information about ports is not explicitly serialized in its XML representation. Ptolemy finds out

about missing information by instanciating the correspondign Java class `ptolemy.actor.lib.Ramp` and asking this class about ports and further attributes. However, the KAOM syntax requires all these informations explicitly to create its rendering. Therefore the Ptolemy-to-KAOM transformation instanciates all Ptolemy classes once and makes their details explicit.

5.2.1 The Prototyping Issue

So far, KAOM is used as a renderer for existing models from other modeling tools. It imports them using a custom M2M transformation as mentioned above.

KAOM can hardly be used as an editor to create new models with a concrete meaning. The editing capabilities are limited due to the small metamodel. GMF only supports a fixed palette where the creation tool buttons correspond to elements in the metamodel. Thus a user can create new entities and then manually add new ports and interconnect them. However, real actor-oriented editing environments would require to add concrete actors from a *library* into the drawing, such as a `Ramp` actor. Otherwise one would create all basic actors from scratch and would defeat any modularity.

This seems to be an issue of object-oriented design that uses static metamodels to specify elements of a language. However, actor-oriented models are not limited. A user can combine primitive actors to a compound one and thus create a new element for the language, i. e., extend the metamodel dynamically. In other words almost all actor-oriented modeling environments use *prototyping* to build an actor library. This is not yet supported by EMF and not intended in the MOF in general.

5.2.2 KIELER Integration

In KIELER the KAOM editor is used to evaluate automatic layout algorithms that support the specific features, such as port constraints and hyperedges. A concrete algorithm is discussed in subsection 6.2.2.

Additionally, it serves as a testbed for visualization approaches for data flow models. The simulation of existing models can benefit much from ap-

plying focus & context mechanisms to an editor with visible hierarchy—note that all aforementioned modeling environments only have hidden hierarchy, but KAOM supports visible hierarchy. However, integrating simulation as well as data visualization and focus & context with view management in KAOM is ongoing work.

5.3 The Unified Modeling Language

The Unified Modeling Language (UML) defines concrete and abstract syntax for multiple diagram languages all combined into one big metamodel. Some open diagram editors for Eclipse are emerging (e. g., MDT/Papyrus) and therefore can be used as a test environment for the KIELER approaches. It can be analyzed how well it integrates with existing editors and an existing complex metamodel.

5.3.1 Automatic Layout

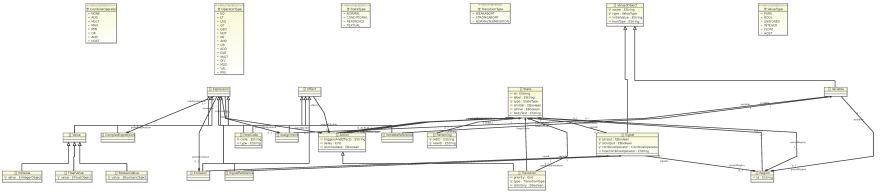
The layout procedure of KIML as presented in Section 4.2 allows to extend any graph layout algorithm to compound graphs, furthermore we gain the flexibility of applying different algorithms on the layers of the inclusion tree. However, for compound graphs where adjacency edges may connect vertices from different hierarchy levels, e. g., UML state machines, this procedure does not perform well, as it cannot take into account these cross-hierarchy edges. A compound graph layout algorithm is required for these cases [SM91, San96b].

The UML also contains diagrams that need special treatment: class diagrams contain associations, generalizations, and dependencies, with different graphical notations. Figure 5.10b shows a class diagram with manually arranged layout, which was very effort prone, took many iterations and hours to get to it, and still does not completely follow the notation guidelines. However, having spent effort, the result is quite compact. Hence, we do not claim that automatic layout always reveals better results, but it will always have a much better cost-benefit ratio than manual layout.

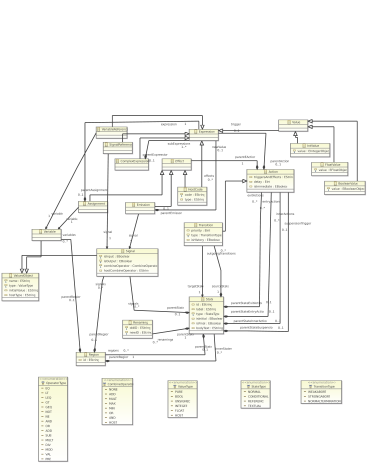
Since there already exist graph drawing libraries with specialized algorithms for class diagrams, it is one of our goals to create an interface to these tools. Figure 5.10c shows a class diagram with an adapted *topology-shape-metrics* layout [TDB88, GJK+03], which is far more readable than the layout of the same diagram shown in Figure 5.10a and compares well to Figure 5.10b with the manual layout.

5.3.2 Structure-Based Editing

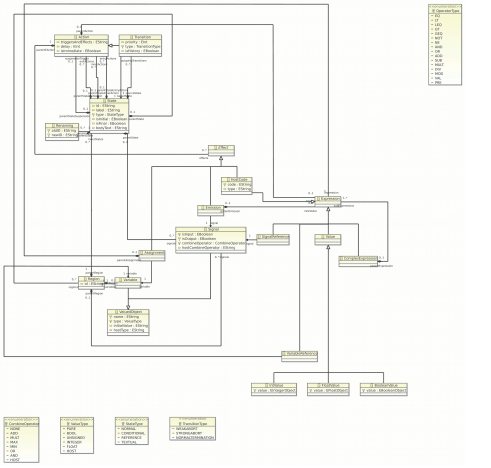
As the Eclipse modeling project contains an open UML meta model, which is used by different graphical editor implementations such as the UML2Tools or Papyrus, transformations can simply be defined for this meta model to be re-used in multiple graphical editors. Structure-Based Editing has been explored for the KIELER SyncCharts editor as introduced in Section 4.5; the editing schemes defined for SyncCharts can already be transferred to



(a) The default GMF layout



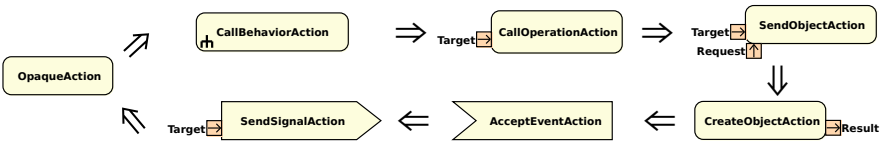
(b) Manually arranged layout, taking much time and not consistently following notation guidelines



(c) The layout provided by OGDF, with manually improved label positioning

Figure 5.10. Layout comparison for an EMF class diagram.

(a) Insertion of a successor action



(b) Toggling transformation between different action classes

Figure 5.11. Activity diagram transformations.

similar UML state machine transformations. Other diagram types can be supported with corresponding transformations, e. g., activity diagrams, for which Figure 5.11a shows a simple example. In principle, all UML diagram types are supported; in practice, however, this approach is feasible for those diagrams for which suitable automatic layout algorithms are provided.

Object Class Transformations

The transformations so far are quite straight-forward, and for Statecharts editing implemented in KIELER the number of transformations is clear and small.

For the UML meta model the situation is rather different. It extensively uses class hierarchy to express different object refinements. For example, in an Activity model there is the class *Action* which has the different subclasses *OpaqueAction*, *CallAction*, *InvocationAction*, *SendSignalAction*, *CallOperation-Action* and *CallBehaviorAction*. All metamodel classes have a graphical representative with slight differences and a creation tool in the palette (which makes palettes of UML editors usually quite crowded). There are two major drawbacks of this approach:

- If a developer inserts any of the above actions and then later realizes that a different class would be more appropriate, the object instance has to be exchanged completely. E. g., changing an *OpaqueAction* into a

CallAction requires to remove the first, insert a new instance of the latter and correct all attributes of the action that have been defined before and fix all connections to and from the new instance. Hence, changing an object's class requires a lot of manual editing steps in the standard editing process.

▻ To provide a sufficient set of transformations for Structure-Based Editing, one would require similar transformations for each of the classes. E. g., the "insertion of a successor action" in Figure 5.11a, which is shown for an *OpaqueAction*, would require a similar transformation rule for all other Action classes.

As an alternative, one could provide only a small set of transformations for one specialization of an abstract class, e. g., only for *OpaqueActions*. These rules suffice to create the basic graphical structure of the diagram employing automatic layout. Then another set of transformations would be responsible to change the concrete class of a model object. To reduce the number of transformations and user interface items, one single conditional transformation could *toggle* between all different subclasses of a given class. For the Action example, the sequence is shown in Figure 5.11b.

Such *toggling transformations* free the developer of two manual tasks next to exchanging the class itself: First, copy all common attributes from the old class instance to the new one, and second, fix all incoming and outgoing connections. In this example we see that it is not always possible to the full extent when classes have different types of connections and ports.

For example, the creation of a new *CallOperationAction* would reduce to the steps:

1. focus an existing action,
2. call the "insertion of a successor action", and
3. call the toggle transformation twice.

It is up to the toolsmith to find a good trade-off between the number of transformations provided for a language and how directly specific editing schemes should be supported. However, especially the toggle transformations are a real benefit not only for pure Structure-Based Editing, but also

for any DND editing, where in general changing object classes requires many manual steps.

The rather complex UML metamodel makes the integration of KIELER with UML editors challenging. Knowledge of UML practitioners would be required to determine the most useful transformations. However, we see here the same great benefits that also apply for the other languages considered so far.

5.4 The Ptolemy Case

In earlier sections we came across Ptolemy already several times because it is a well established modeling environment. Here we use Ptolemy as a case study on how KIML can interface with pure Java applications outside the scope of Eclipse.

Ptolemy II is a graphical modeling suite developed by the Center for Hybrid and Embedded Software Systems (CHESS) of the University of California, Berkeley, USA, under the lead of Edward A. Lee [EJL+03].

Ptolemy supports *actor oriented* system design, where the building blocks of a system are *actors*—small software components that consume data tokens and produce new data tokens like functions. Actors can be interconnected to form a whole network of data flow. A *director* is a software component that is responsible to organize the execution orders of the actors. Ptolemy provides a wide variety of director implementations that execute actor models in different ways, usually following specific formal semantics, which are also known as *models of computation* in the Ptolemy notion. Directors have to follow only a few rules and implement a certain interface and, hence, a model developer is able to create custom directors that execute actor models in any way.

Actors can be implemented directly in some host language, most likely Java, or be composed of another Ptolemy model, i. e., another actor network. Each composite actor has its own director to control the model execution of this single actor contents. So a key concept in Ptolemy is to use different composite actors, each using a different director in one model yielding heterogeneous models that comprise multiple models of computation.

One use case could be to model a software controller, with discrete events semantics, and a simulation plant model of the mechanical parts of the system or its environment, with a continuous time semantics that reflects its nonlinearity. This way a whole system can be simulated in Ptolemy prior to its physical integration in the target.

5.4.1 The Ptolemy Layout Problem

Ptolemy models can be created either

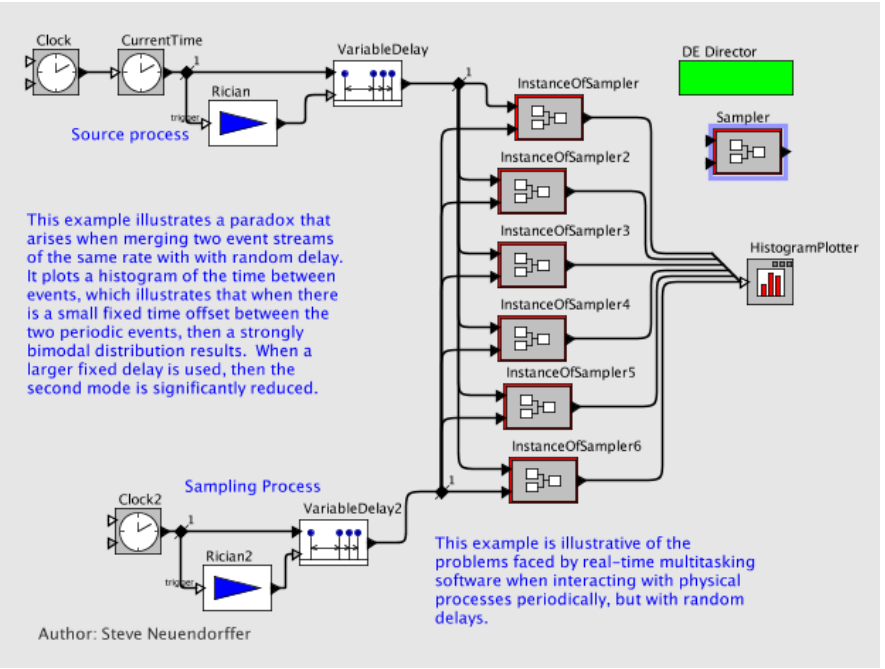


Figure 5.12. A graphical representation of a Ptolemy actor model

- programmatically by its Java API,
- manually by writing the model specification in an XML language called Model Markup Language (MoML), or
- by drawing a graphical representation of a Ptolemy model in a diagram editor called *Vergil*.

A typical graphical Ptolemy representation is shown in Figure 5.12. Actors are represented as rectangles with some actor-specific icon. Actors produce or consume data on *ports*, either inputs or outputs or both, represented as a small triangle at the border of an actor icon. Rectilinear

polylines connect ports with each other where usually a simple port may only be connected to exactly one connection, semantically called a *relation*. A connection can be branched by introducing an explicit *relation vertex*, a small black diamond icon, to which multiple connections may be attached.

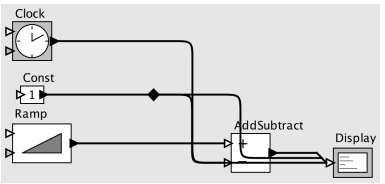
As Figure 5.12 also shows, there can be other components in a Ptolemy model that do not need to be explicitly connected to some other components. The most prominent one is the *director*, represented by an unconnected labeled box. Documentation blocks or more generally *text attributes* can be placed in a model at arbitrary positions. There are quite a few others like these available in the Ptolemy libraries.

Node Placing

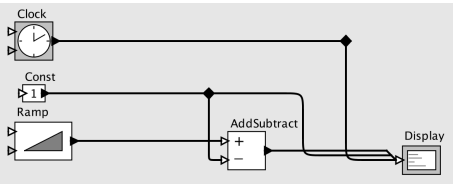
In Ptolemy all aforementioned nodes can be placed manually, e. g., actors, directors, relation vertices and text attributes. The horizontal and vertical coordinates of all nodes are persistently stored and are part of the Ptolemy model. These locations can be adapted programmatically and, hence, can be used by the layout algorithm.

Most iconed boxes such as actors, vertices, and directors have a fixed size, which can neither be changed manually nor programmatically. The size is set once for the node by its specification from its icon and settings which might be stored in the Ptolemy node library. The size of text attributes is variable by the length and wrapping of the text. The user can influence the shape of text boxes by the text's length, its wrapping (which must be set manually) and font size and style. Automatic wrapping to a given text box width is not intended. Actors also have a label, its name, which can be arbitrarily customized by the user. It can have an arbitrary size by using line breaks. An outer bounding box for nodes is given by Vergil including all elements of the node, e. g., the ports and the label. Therefore in general the size of an actor is variable but fixed for a specific instance where a label was set by the user.

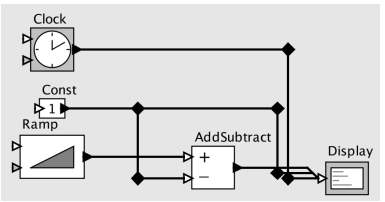
Structural hierarchy is an important concept in Ptolemy but is not expressed directly in the graphical representation like discussed in Section 1.2. Contents of composite actors like the *Sampler* in Figure 5.12 are not shown in the same diagram but can be opened in a completely new canvas. So



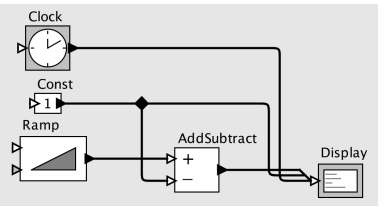
(a) Arbitrary model with node and connection overlappings by the standard Manhattan router



(b) Optimized layout by introducing an additional relation vertex and moving nodes



(c) Inserting one relation vertex per bend point results in full control over connection routing



(d) Real routing with arbitrary bendpoints by a special implementation of a router

Figure 5.13. Connection Routing in Ptolemy

layout of hierarchy is not in the scope of layout in the Ptolemy Vergil editor.

Connection Bend Point Placing

The placing of bend points is an issue in Vergil, because bend points are *not* part of the Ptolemy model and, hence, are not persistently stored. For a user it is not possible to directly influence the locations of connection bend points, neither manually nor programmatically.

Ptolemy uses an internal connection router that dynamically computes bend points for a connection between two endpoints. This router is a rectilinear style Manhattan router without obstruction avoidance. Hence, given an arbitrary node placing, it is likely to get an arbitrary number of overlappings of connections with nodes and also with other connections.

However, there are some options for influencing the routing of the

original Vergil editor shown in Figure 5.13. Depicted in Figure 5.13a is a placement of nodes for which the Ptolemy router results in two overlappings of nodes and one overlapping of connections. The latter results in an ambiguous diagram, where it cannot be decided which connection enters the *AddSubtract* actor and which enters the *Display* actor.

There are different ways to influence the routing of edges:

Node Moving Routing is usually influenced by moving nodes manually. If this is not sufficient, new nodes can be inserted, which then can be placed according to the desired layout. Relation vertices are nodes that can be added within a connection without changing the semantics of the model. Hence, insertion of a few relation vertices and optimizing the placement of all nodes is the usual way of manually creating collision-free connection routings as shown in Figure 5.13b.

This has also been used by the original author of the Ptolemy model in Figure 5.12, where the actor instances of the Sampler in the center are slightly offset in order to get an unambiguous connection routing to the HistogramPlotter. Additionally there are two routing slots created for the two inputs of all of the Sampler instances by adding a relation vertex and placing them also with a little distance at the horizontal coordinate.

Vertex Insertion To get full control over the connection routing, the Ptolemy Manhattan router can be handed trivial routing tasks by creating only straight connection pieces. This can be achieved by insertion of relation vertices for every bend point in the diagram. An example is shown in Figure 5.13c.

The advantage is that one gets full control over the bend point placement as there are no real bend points but only vertices which can be placed. In spite of that the drawbacks are obvious. The most visible is that the diagram gets less appealing by this crowded view. The other is that only for the goals of layout the underlying semantic model gets heavily changed by insertion of new semantic objects. Even while this has no semantic implications, the semantic model gets crowded.

One way to make the graphical representation more appealing would be to simply hide the helper vertices such that they do not get drawn at all.

Improved Manhattan Router Another idea is to improve the functionality of the built-in Manhattan router of Ptolemy to also avoid obstacles and minimize edge crossings itself.

However, the routing problem of a standalone router is totally different to a complete layout. The routing algorithm of KIELER presented in subsection 6.2.2 is highly interwoven with the placing problem of nodes and therefore can exploit the already prepared routing slots of the earlier layout phases. This way the routing problem can be coped with well. In contrast, a general connection router that is separated from the node placing steps is much more complex: given an arbitrary node placement, the calculation of bend points has to take into account all other nodes, while space everywhere between the nodes is limited.

Developing a general stand-alone router is out of the scope of this work and might be approached in the future in another context.

Bend Point Router The last alternative is to replace or upgrade the Ptolemy Manhattan router in order to graphically apply a given pre-computed set of bend points to the connections. Then the calculated bend points of the algorithm presented here could also be applied to the Ptolemy diagram. This would be the simplest and most appealing way to get the desired result.

Technically, however, this implies a few basic changes in the Ptolemy infrastructure. First, this requires means to persistently store bend points related to a connection. For example the MoML would have to save the bendpoints related to connection pieces, which themselves are also not really explicit parts of the Ptolemy model.

For evaluation of the approaches, a two-level Vertex Insertion and a Bend Point Router have been implemented. This is further explained in the experimental results below. First, let us see how the KIELER layout problem fits to the Ptolemy Vergil editor.

5.4.2 Mapping the KIELER Layout Problem to Ptolemy

Employing the KIELER layout algorithms to Ptolemy seems to be a straightforward mapping. Unfortunately it is not, due to some subtle differences in the two layout problems. These will be discussed in the following.

Abstraction

An issue in modeling tools like Ptolemy is that in their implementation they try to follow some abstraction rules to separate concerns and to hide implementation details of lower levels in the higher ones. In general this is a good idea as long as the intended application on higher levels does not demand any information or API of lower levels.

In graphical modeling the main user interaction mechanism still is freehand Drag-and-Drop (DND) editing, and hence, most tools are designed only for this purpose. There are high-level interfaces to manually move graphical items one by one and the feedback is given by the graphical representation to the developer's eyes directly. Usually—and this also holds for Ptolemy—the details are not available in the public APIs. For example it is not always trivial to obtain the actual structure and layout of a diagram programmatically in actual coordinates, because some location functions are not stated publicly.

In Ptolemy there is such an abstraction between the underlying graphical drawing framework called *Diva*[3] and the more specific Ptolemy II Vergil editor. Some information was originally hidden in protected APIs, such as the orientation of ports, and had to be made public for this work. Other information is not consistent and needs many special case handling in the implementation for reading and reapplying the layout to the diagram. For example locations in horizontal and vertical coordinates are handled differently in Diva and Ptolemy and even within Ptolemy differently for actors and text attributes. Bounds in Diva give the location by their top left corner (just like in KIELER) while Ptolemy uses the center point for most items, except for text attributes where it is again the top left corner.

Usually in a drag-and-drop editor the user does not need to care about

[3] `http://embedded.eecs.berkeley.edu/diva/`

such issues because movements are done relative to current coordinates. Nevertheless for automatic layout this makes the interface code more verbose and error-prone.

Nodes

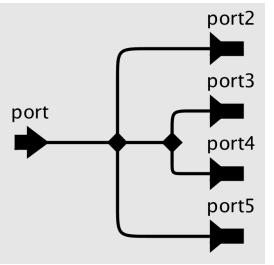
Nodes in Ptolemy can comprise multiple elements like an icon, text, and ports, where the icon usually does not fill the whole bounds of the actor. Its text, e. g., the actor name, and the ports extend the bounds, sometimes by a significant amount. Hence, to reserve enough space in the layout for placing the nodes, the overall outer bounds are used for the sizes of nodes in the KIELER KGraph. This results in a KGraph where the ports are always fully covered by the KIELER node's bounds. This is no problem for the layout algorithm as long as it knows to which side of the node—North, East, South, or West—the port belongs. This information must be read from the Ptolemy diagram.

Next to actors with ports there are other nodes which are connected but have no ports at all. These are for example internal ports of composite actors.

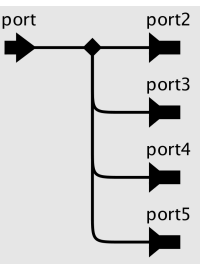
Relations

It is not obvious how to treat relation vertices. Figure 5.14 shows a layout problem specific to Ptolemy's concept of Relations. Semantically they are treated as hypernodes, connecting multiple ports, especially more than one. Hence, the intention is that graphically a connection junction is rendered as a vertex. However, due to the rectilinear routing style, no more than four ports can be interconnected using one Relation with a correct rendering. If there are more, then multiple Relations with vertices are required. A different positioning of nodes might even require more of such vertices. If the number of connections and/or the layout is ignored, then models will have connection overlaps where some connection junctions are not rendered by a relation vertex. Other frameworks such as Simulink and SCADE also draw connection junction vertices, but treat them consistently as a decorator only.

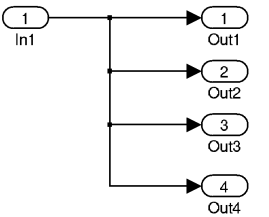
The main question is, whether changing the layout of a model should change the model structure by changing the number of Relations. As they are semantic elements—however, usually without semantic implications—this breaks the abstraction between model and view. Hence, a Relation concept that would handle connection junctions more consistently would allow a cleaner way of routing hyperedges.



(a) Number of required vertices depends on layout.



(b) If the layout is not considered, connection junctions might get rendered inconsistently.

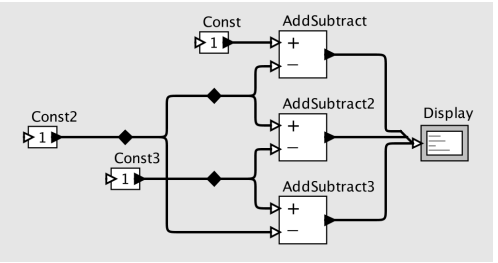


(c) Simulink and SCADE use vertices only as decorators, but consistently.

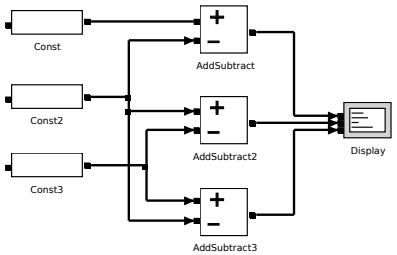
Figure 5.14. Different layouts can contain different amounts of Relations for the same model.

Following the Vertex Insertion strategy from above would regard relation vertices as connection bend points and not as nodes. However, this would change the model structure by removing and/or adding Relations.

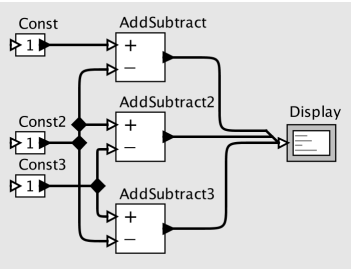
In a simple approach, no connection routing is applied at all and the internal Ptolemy Manhattan router does the job. Here, relation vertices are treated as usual nodes with a size of zero for simplicity. Still, treating them as nodes is a simplification in order to avoid changing the model structure. The cost is that they will take up a whole slot in a layer in the layout algorithm, resulting in a layout with superfluous layers as shown in Figure 5.15a. However, the algorithm itself (see subsection 6.2.2) has the capabilities to correctly route hyperedges as shown in Figure 5.15b, which is not yet used in the Ptolemy integration for the aforementioned reasons.



(a) Treating Relations as nodes in the layout algorithm to avoid structure changes leads to superfluous vertical layers.



(b) The layout algorithm itself can route hyperedges in special routing slots, here shown in the KAOM editor (see Section 5.2).



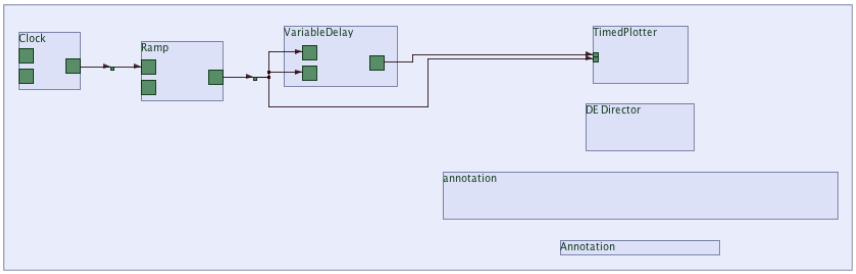
(c) The same layout manually applied in Vergil to avoid superflous layers. In this case this is possible without changing the amount of Relations.

Figure 5.15. Treating Relations as nodes

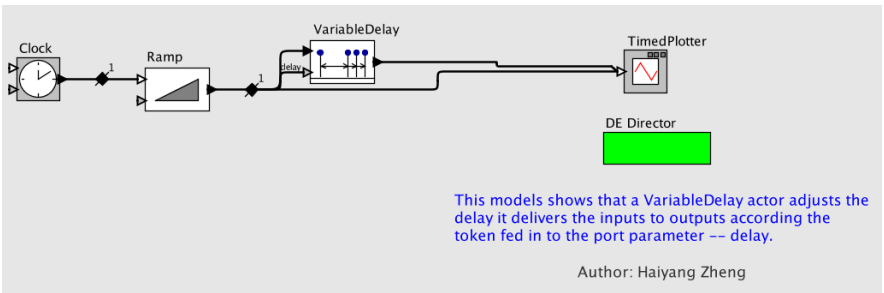
Block Layout

A Ptolemy diagram not only consists of connected nodes but also comprises unconnected decorating nodes such as text boxes for documentation, attributes like a director, and other such items.

Especially text boxes for documentation tend to get quite large as Figure 5.12 shows. Processing them in a layered layouter, like presented in subsection 6.2.2, places all unconnected items in the last layer in a large pile.



(a) Internally computed layout



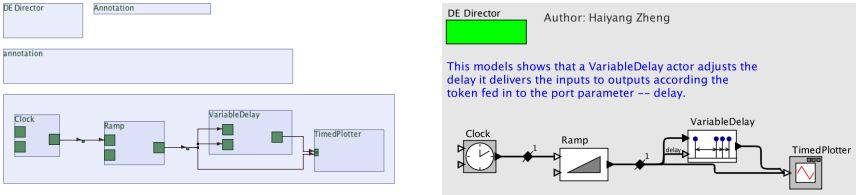
(b) Ptolemy diagram

Figure 5.16. Ptolemy layered node placing including unconnected nodes results in unappealing stacked views.

If the texts are quite wide, the whole layer is stretched to the size of the largest item. This leads to unpleasant results like shown in Figure 5.16.

The situation improved by employing the structural hierarchy (composite nodes) mechanism of KIELER, which allows for every node in the graph to contain subgraphs. The feature that each subgraph can be layouted with a different layout algorithm allows to handle connected and unconnected nodes differently.

All connected nodes are put in one composite node, which is layouted with the layered layout algorithm. All other nodes are layouted together with the composite node by a simple block layout heuristic. The result is



(a) Internally computed layout **(b)** Ptolemy diagram

Figure 5.17. Separated hierarchichal placing for connected nodes and a simple block placing for others.

shown in Figure 5.17. The block layouter assumes that the composite node containing the connected nodes is the largest node. It arranges all other nodes on top of that next to each other and wraps at the composite node sequentially with increasing size. The algorithm allows to set priorities to nodes, which is used to always position the director of the node in the top left corner. For many diagrams this simple heuristic reveals acceptable and especially much more appealing results than the layered layouter alone.

For the future, this simple block layouter could be improved to solve the block layout problem more generally, i. e., place a set of unconnected nodes with fixed sizes optimally. To some extent this relates to Harel's Blob layout [HY02]; however, Harel assumes that the sizes of the nodes are variable, which is not the case here.

A drawback arises when modelers use *secondary notation* by visual cues which are not part of the formal notation. In textual languages different ways of indentation are secondary notation. They can be used for example to show block scopes to make a program more readable while it is ignored by a compiler. In Ptolemy it is common to use text attributes to give a short documentation to specific graphical parts on the canvas. Therefore the text attribute is usually placed close to the nodes that are to be explained by the text; this placement is also a form of secondary notation. There is no semantic link that can be made between the text and the node and, hence, no automatic means can take this relationship into account.

For this case different versions of the layouter have been implemented,

such that it can either layout

- all nodes with the above described combination of layered and block layouter, or
- it can place only all nodes that are connected and all other nodes (e. g., director, text attributes, or parameters) stay untouched.

This way the developer can run the layout and manually place unconnected items for documentation in whatever way. After subsequent layout runs these will stay at the respective locations.

Graph Direction

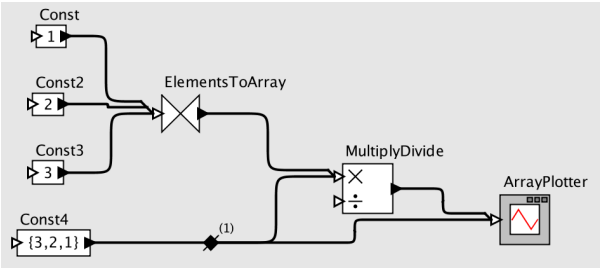
The layered layout approach is designed for directed graphs only and uses the direction information of edges to place the nodes onto the different layers. This usually results in drawings with the major direction of data flow from left to right for horizontal layout.

Although in general Ptolemy is a data flow language that transfers data tokens between ports, the direction of flow is not necessarily unambiguous or enduring.

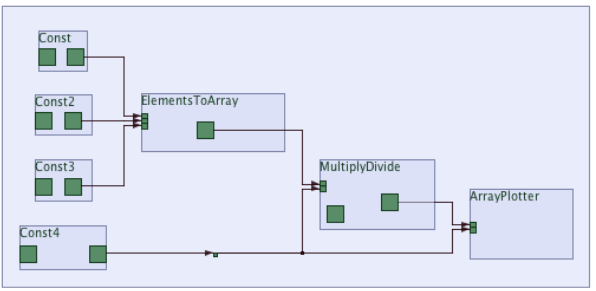
First, the low level graphical representations of single links in Diva between ports or relation vertices have directions not related at all to the Ptolemy flow of data. As connection figures themselves show no arrow heads, this is of no relevance. It implies that from the direction of a graphical edge the flow direction cannot be deduced. Hence, one needs more information about a whole connected set of relations and what kind of ports—input or output—their endpoints are. Then the direction of such a relation set resp. low-level edge set can be derived.

Second, the direction of data flow can change in a Ptolemy diagram. Flow of data is done in single steps by passing single data tokens between ports. A port of an actor can be both input and output port at the same time. While a port cannot produce and consume data at one port at the same time, it can do it interleaved. Hence, connections between ports might be bidirectional.

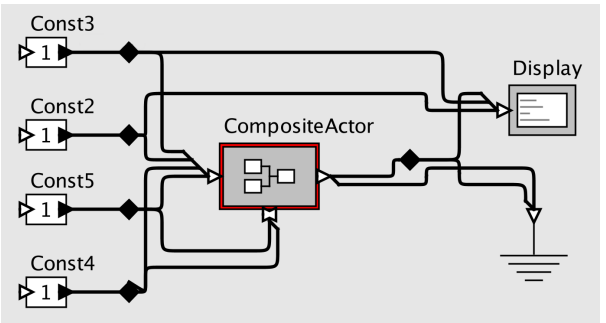
As this is not a very typical way to model in Ptolemy, the direction in a connected relation set is approximated by a simple heuristic that searches



(a) Multiports in Ptolemy take multiple incoming connections in a specific order.



(b) In the KIELER datastructure of (a) this gets mapped to a set of small ports with a small offset each.



(c) Examples of routing with hidden vertices to all directions of multiports

Figure 5.18. Multiports in Ptolemy get represented by sets of ports in KIELER to help avoiding additional connection crossings.

for the first source port in the set and uses this to determine the direction for the layout algorithm.

Multiports

Ports in Ptolemy can be of different kinds. Simple ports just take or produce up to one connection, while *multiports* (carrying a white port icon) allow multiple connections. The multiple connections get ordered and can be accessed by the owner actor by index of the so-called *channels* of the multiport. The order of the channels is determined by the temporal order the edges were connected to the port. Graphically the order is presented in Vergil as shown in Figure 5.18a. Hence, the order of the nodes in the layers might introduce additional connection crossings for nodes connected to the same multiport.

As shown in Figure 5.18b, each Ptolemy multiport gets mapped to a set of multiple small ports in the KGraph data structure. The small helper ports get shifted a bit according to the amount the Ptolemy connections are fanned out. This emulates the order of the connections, and the nature of the layered layout approach will try to avoid additional edge crossings in the crossing reduction phase described in subsection 6.2.2.

5.4.3 Experimental Results

The implementation of the KIELER and Ptolemy interface evaluates different approaches. An example is shown in Figure 5.20.

Two-Level Vertex Insertion

This scenario follows a two-level approach. The first simple and non-invasive mode only places nodes and does not touch connections. Hence, the internal Ptolemy router is used to route the edges, which is likely to produce overlappings. The second mode routes edges by the VERTEX INSERTION method to showcase the routing capabilities. This involves a lot of hard-coded model transformations that introduce new relations and remove others while trying to keep the semantics as before.

Figure 5.19. New buttons in Ptolemy to control the layout for the VERTEX INSERTION method

To provide the choice for the user, a new toolbar with five buttons (plus one to call the original Ptolemy layout) was introduced to the Vergil GUI, shown in Figure 5.19 with the following functionality:

Layout all nodes This option places all nodes following the Block Layout approach introduced above. It places both connected and unconnected nodes such as text annotation nodes for documentation. The intention was to provide the possibility for an initial non-overlapping layout for all elements.

Layout connected nodes This option places only the connected nodes. It especially leaves the attribute nodes at their current location. This option can be used to place textual annotations manually.

Layout and route connected nodes This option places only connected nodes as the option before. Additionally it routes connections using the VERTEX INSERTION method presented above. Hence, it creates new relation vertices. An example is shown in Figure 5.20b.

Hide unnecessary vertices This option hides all relation vertices that are connected with 2 or less links and therefore have no semantic implications. In most cases they have only been introduced to route edges. Hiding vertices in Vergil means that a corresponding property is added to the vertex object which causes Vergil to prevent any rendering of it. Hence, the usual diamond symbol is not visible anymore. Incoming and outgoing connections are drawn to the center point of the vertex and thus it looks simply like a bend point with orthogonal routing. Figure 5.20c shows an example.

Remove unnecessary vertices This option removes all unnecessary relation

vertices from the model by a model transformation. It can be used to simplify models to which many routing vertices have been added.

Further results of the different options can be seen in the examples starting with Figure 5.21. One drawback is the slightly different rendering of bend points. The original Manhattan router draws an appealing curve at bend points. With vertices replacing bend points, the connection is a pure polyline with hard bends.

The acceptance of this approach was rather cautious. Next to the rendering, the major problem is that the user interface is overburdened with too many options. A normal user is not aware of the implications of routing—that it inserts new vertices—and does not really understand why the different buttons are necessary. A lot of explanation and maybe training is required and this overhead prevents permeation of regular usage of all layout features among the Ptolemy users.

To mitigate these effects, the user interface was reduced by introduction of a real Bend Point Router as explained above.

Bend Point Router

The bend point router takes pre-computed locations for bend points for a link and bends the drawing of the connection at exactly these locations, as shown in Figure 5.20d. So far, Vergil only provided the default Manhattan router. Therefore this specialized router was implemented extending the original one by means to place bend points according to corresponding coordinate properties stored for each link. However, links are no semantic objects in the Ptolemy abstract syntax and therefore the bend point data are attached to relations, where multiple links may be associated to a relation. Therefore one relation can store multiple bend point lists corresponding to multiple links.

After a layout run of the auto layouter, the automatically generated bend points are used. There is a fallback to the old Manhattan router when there are no bend point data available.

A design question is how to allow manual editing of connections that have bend point data attached. Currently, bend points can only be added programmatically, but there exists no graphical user interface in the diagram

to manually move, create or delete bend points. Therefore, whenever the user manually starts to drag single items around, any such bend point information is automatically removed from a relation that was associated to a connected link before. This enables a simple interface to allow automatic layout with routing by simplifying the fallback to manual layout, in case the user really wants to move things manually afterwards. However, with manual layout, only the default Manhattan router is available in any case.

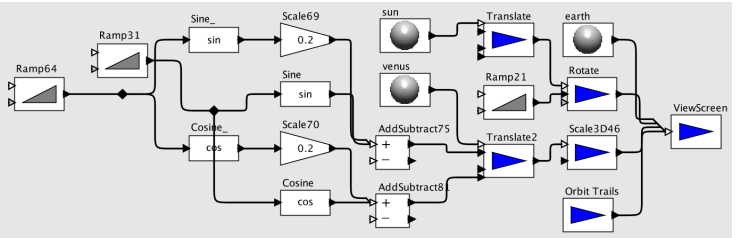
An extension is to allow translations of connected elements to move whole regions. If relative positions of start and end of a connection are kept, the bend points can also be translated and therefore manual movement would keep the routing in such cases.

Examples

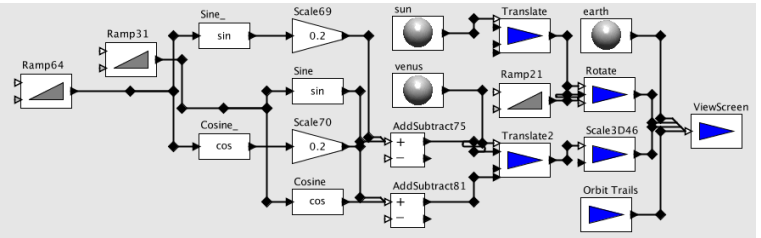
In the following we see some example layouts. The models are mainly taken from the official list of Ptolemy demos accompanying the Ptolemy tool. We show the original layout and different results of the layout algorithm. This is the placing of all nodes (connected and unconnected but without routing), which usually looks much worse than the placed and routed images, because the internal Ptolemy Manhattan Router does not avoid any overlappings of connections with other connections or nodes. Furthermore, a version of the connected part where routing was done by the Vertex Insertion method is shown, introducing helper vertices which may get hidden afterwards. Additionally, the version with the Bend Point Router is presented that usually gives more appealing results. Sometimes we also see the internal KGraph data structure which is the direct output of the layout algorithm.

In general the hand-made original output is expected to be the best, because the official Ptolemy demos are showcases caringly designed to be presented to the public especially by expert Ptolemy users. So one can expect that enough time was spent to make the layout sound. This might be the biggest drawback, that some developer has spent some considerable effort to create that layout.

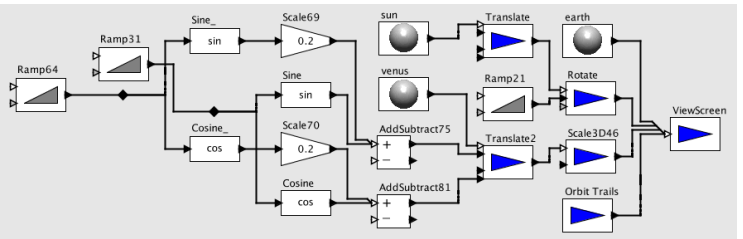
Although the Ptolemy models have very different overall sizes, they use composite nodes to reduce the amount of nodes on every single canvas.



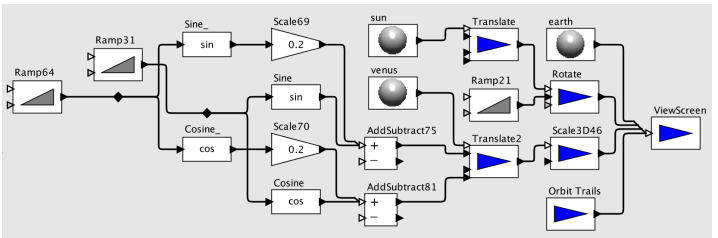
(a) Original Manhattan router produces overlaps



(b) Routing by Vertex Insertion



(c) Hiding unnecessary routing vertices

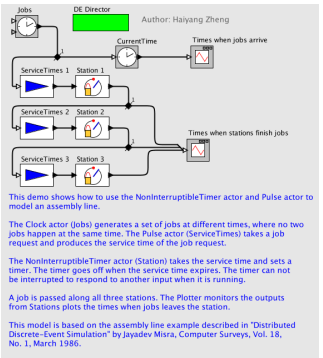


(d) Routing with the Bend Point Router

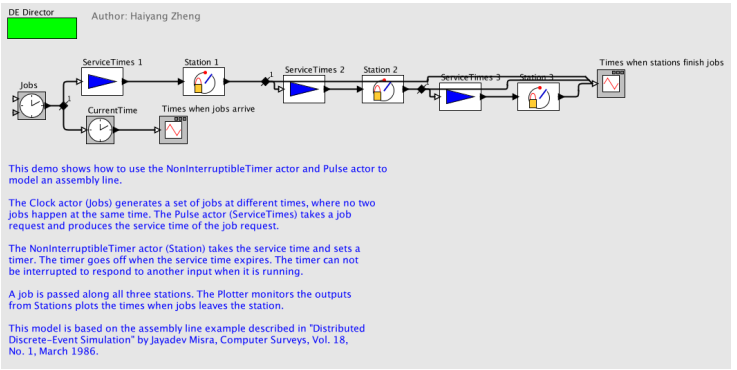
Figure 5.20. Different routing approaches for the Ptolemy Vergil Editor

Hence, almost all models in the Ptolemy demos have about the same size on one hierarchy level. As Ptolemy does not yet support to directly visualize nested models, the compound graph feature of our algorithm cannot be used. Therefore very big examples (e. g., > 30 nodes) would be quite artificial.

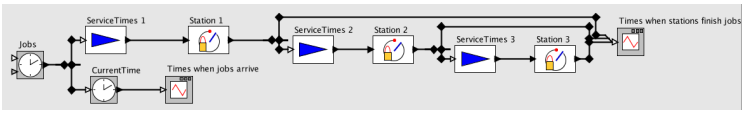
The examples that present routing results omit the unconnected nodes like the director and text annotations.



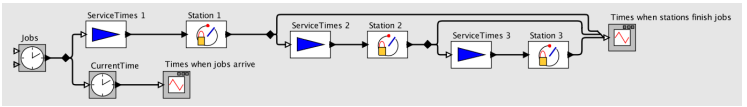
(a) Original



(b) Only placed all nodes

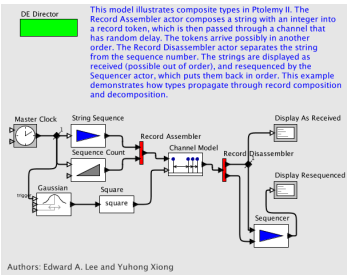


(c) Placed and routed connected nodes with helper vertices

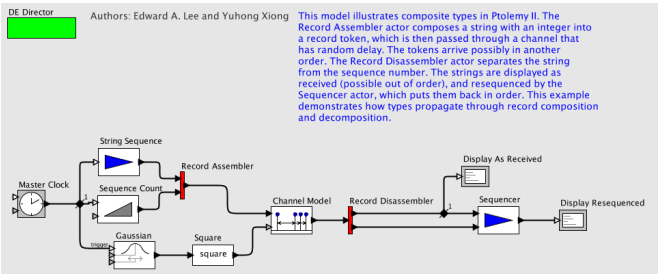


(d) Placed and routed with the bend point router

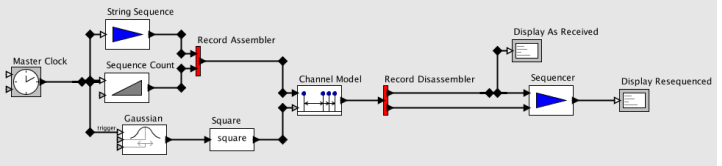
Figure 5.21. AssemblyLine: This is an acyclic, fairly sequential model that results in a wide horizontal span.



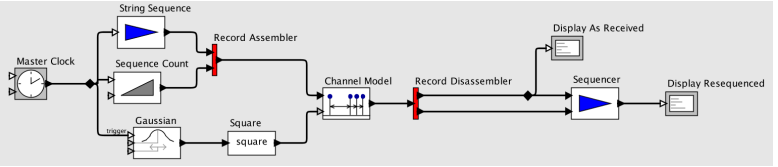
(a) Original



(b) Only placed all nodes

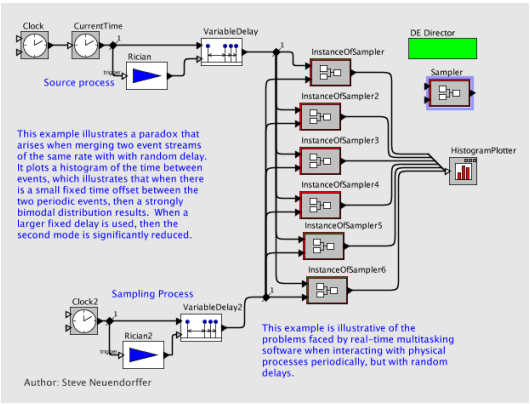


(c) Placed and routed connected nodes with helper vertices

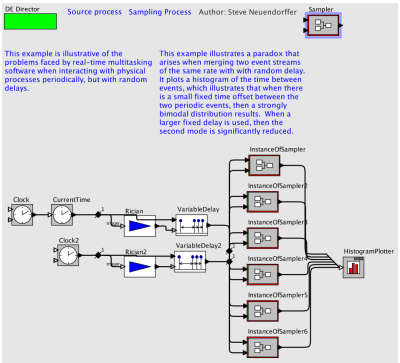


(d) Placed and routed with bend point router

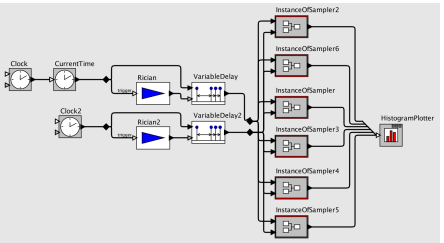
Figure 5.22. Router: A model where the manual layout is quite optimized and packed. In the generated layout some rather small nodes have long labels and, hence use much space in their layer, e. g., the Record Assembler/Disassembler.



(a) Original

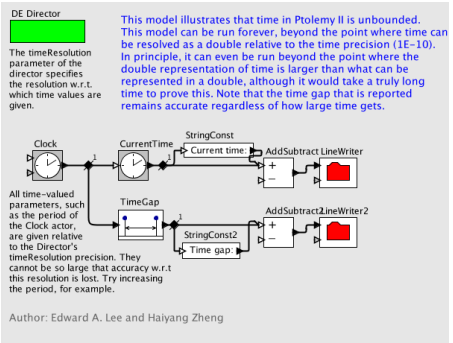


(b) Only placed all nodes, no routing

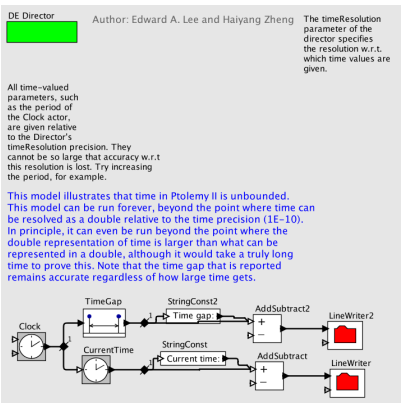


(c) Placed and routed with the bend point router

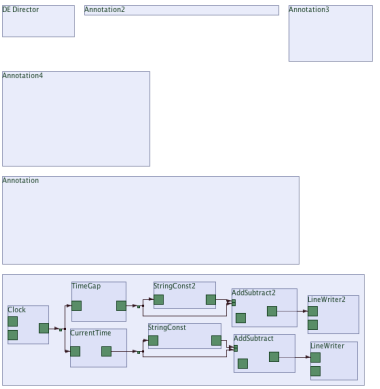
Figure 5.23. TimingParadox: In the original layout (a) the author slightly moved all nodes to reveal a clean connection layout. Without setting bend points explicitly in the auto-layouted version, the result shows many connection overlaps (b). Setting bend points by relation vertices gives a clear routing (c). However, considering relation vertices as regular nodes results in a suboptimal vertex placement, because junction points of hyperedges are not each represented by a relation vertex.



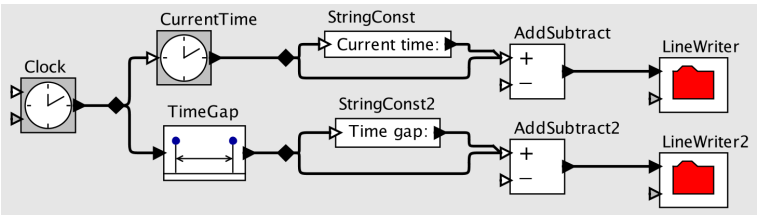
(a) Original



(b) Only placed all nodes, no routing

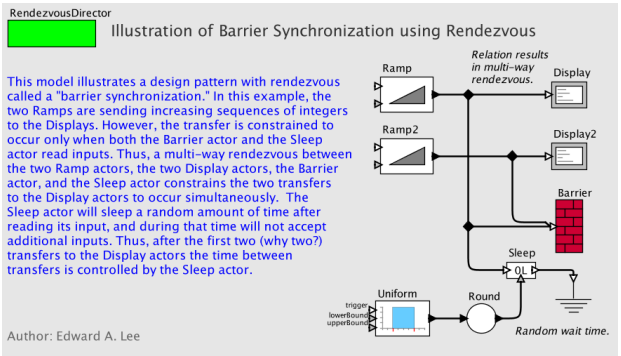


(c) Corresponding KGraph

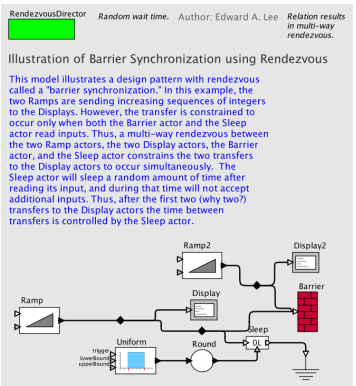


(d) Placed and routed with the bend point router

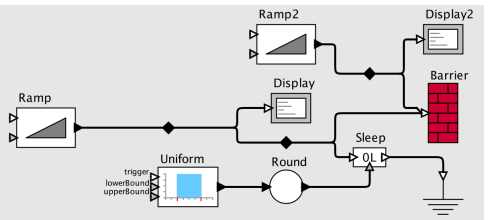
Figure 5.24. LongRuns: The box layout of the text annotations is suboptimal. In the original the lower left text box is wrapped to get a specific shape of the text to get a compact (overlapped) layout. Again, the explicit routing helps to avoid overlaps.



(a) Original

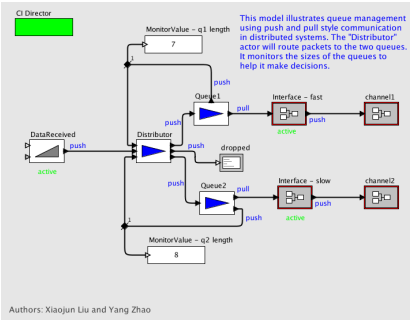


(b) Placed all nodes, no routing

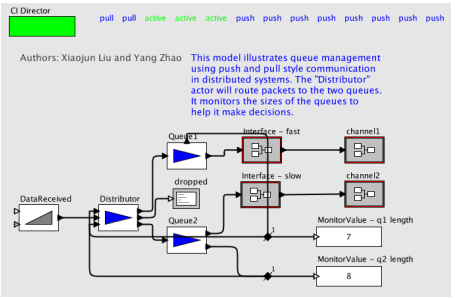


(c) Placed and routed with the bend point router

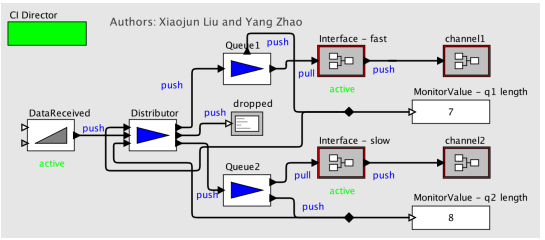
Figure 5.25. Barrier: The connected components are clearly laid out without overlappings. Considering relation vertices as normal nodes in the layout creates a rather wide layering and usually does not position them at connection junctions. Additionally, while the text box layout looks alright, the author of the original has placed text nodes next to connections, which give an information about the corresponding relation. In the structure of the diagram this implicit connection between the objects is not visible. Hence, the layout algorithm cannot take that information into account while laying out items. Therefore, in the auto-layouted version the context sensitive text attributes loose their context.



(a) Original

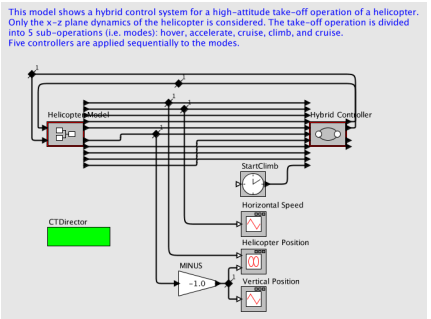


(b) Placed nodes, no routing

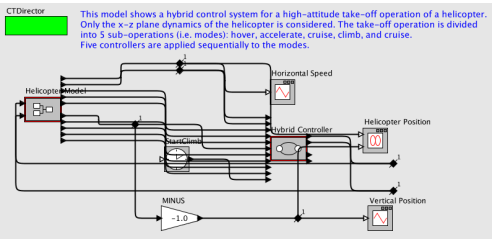


(c) Placed and routed with the bend point router with manually placed text annotations

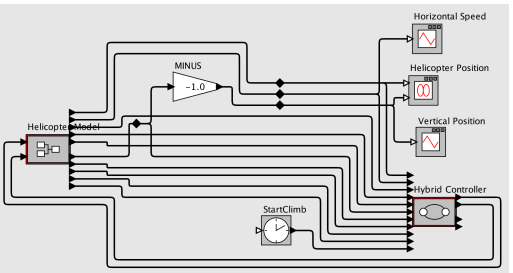
Figure 5.26. CI-Router: While the node placing is good, the Ptolemy Manhattan Router produces bad connection-node overlappings. Again the text attributes placed next to graph items totally loose their context. Here it helps to only place (and route) connected nodes while unconnected such as text attributes are left untouched. This way the user can manually place text attributes to document special parts of the model.



(a) Original

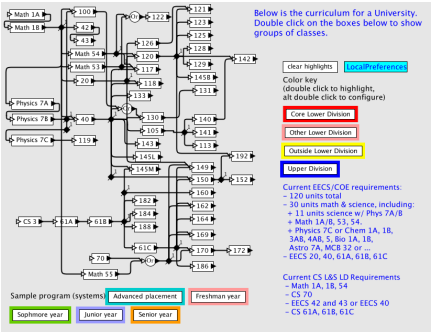


(b) Only placed all nodes, not removing vertices

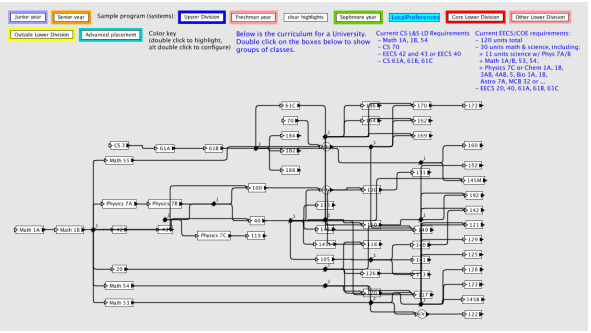


(c) Placed and routed with the bend point router

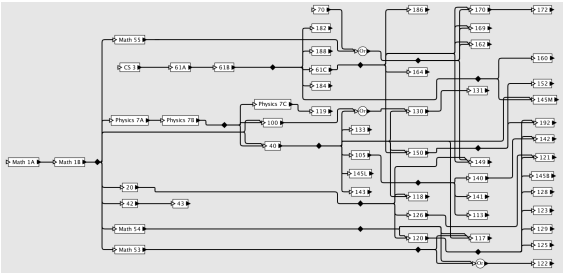
Figure 5.27. HelicopterControl: This is a model with many parallel connections. They get routed around some nodes, which the Ptolemy Manhattan router ignores and produces a tangled mess of wiring. Additionally, the original author used relation vertices to enhance the original layout. A simple placing without removing unnecessary relations keeps the vertices—although it is clear that they loose their original intention—and regards them as usual nodes. Removing such vertices completely before layout and routing results in a clear drawing.



(a) Original



(b) Only placed all nodes

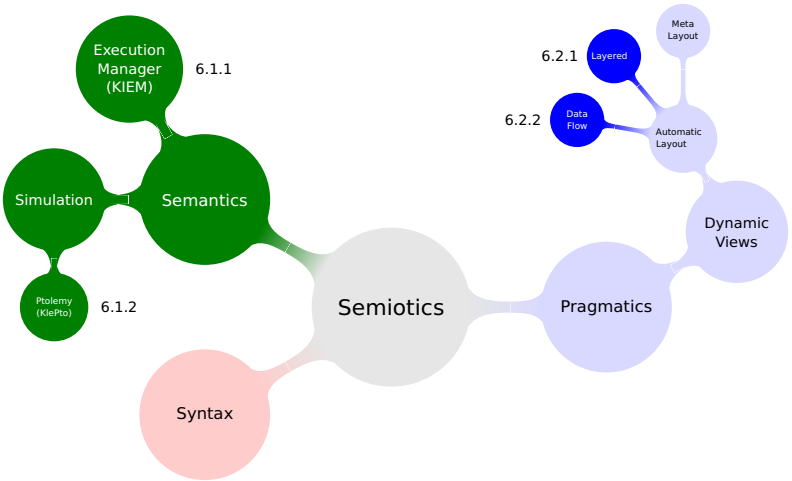


(c) Placed and routed with the bend point router

Figure 5.28. Curriculum: This is a model with many small nodes. The placing looks quite good, although it is difficult to judge whether the original is done following any semantic secondary notation. The diagram becomes wider, but better reflects the order of the nodes.

Chapter 6

Support Projects



During the course of this thesis, some stepping stones were required and missing in the target platform Eclipse. Some of them were not directly the major aim of KIELER but it was necessary to get solutions in order to get back onto the KIELER train.

To evaluate visualization effects also for behavior model *execution*, some infrastructure to execute models was required. It gets addressed in the next Section 6.1.

With a growing set of diagram notations, more sophisticated or specialized layout algorithms are required. As automatic layout is a key enabler for most of the KIELER approaches, concrete layout algorithms became neces-

sary. In general, the development of new layout algorithms is not the scope of KIELER, and the graph drawing community is encouraged to provide new layout approaches or interface existing ones with KIML to make them available to the Eclipse community. Hence, for some initial capabilities, some generic layout libraries have been integrated and some specialized algorithms, tailored to requirements of specific DSLs. This is covered in Section 6.2.

6.1 Model Execution

Computer simulations are an established means to analyze the behavior of a system. On the one hand one wants to be able to predict and better understand physical systems and train humans to better interact with them, for example weather forecasts or flight simulators. On the other hand one aspires to emulate computer systems—often embedded ones—themselves prior to their physical integration in order to increase safety and cost effectiveness.

The basis for such a simulation is usually a model, an abstraction of the real world, carrying sufficient information to specify the relevant system parameters necessary for the semantic analysis and execution. The notation of a model instance is a concrete textual or graphical syntax.

In the past all model editing, parsing, and processing facilities were manually implemented with little generic abstractions that inhibit interchangeability. Standardized languages, e. g., the UML, try to alleviate this, but they are sometimes too general and complex to be widely accepted.

As a recent development, DSLs target only a specific range of application, offering tailored abstractions and complying to the exact needs of developers within such domains. On the one hand, there are already well established toolkits like the *Eclipse Modeling Framework* (EMF) to define an abstract syntax of a DSL in a model-based way. They provide much infrastructure, such as a metamodel backbone, synthesis of textual and graphical editors, and post-processing capabilities like model transformations, validation, persistence, and versioning. We have learned about that especially in Chapter 4 and Chapter 5.

On the other hand there is the semantics of such a DSL. This additionally has to be defined in order to let a computer execute such models. For the specification of the latter no common way exists yet. But as such a semantics often exists at least implicitly in the mind of the constructor of a new DSL, there is a need to provide a way for making it explicit.

The contribution here is a proposal on how DSL semantics can be defined by using existing *semantic domains* and existing model transformation mechanisms without introducing any new kind of language or notation focusing on an Eclipse integration.

Figure 6.1 shows an example setup of the architecture. An implementation overview about the general approach of integrating simulations in the Eclipse platform is given in subsection 6.1.1—the Execution Manager Runtime in Figure 6.1. In subsection 6.1.2 I present how to define semantics for an example DSL with the Ptolemy II suite (cf. Figure 6.1). In this context a case study about simulating SyncCharts by leveraging Ptolemy is presented. Additionally I show that this solution is extensible and has open support for, e. g., model analysis and validation or co-simulations.

Figure 6.2 shows a simulation run with KIEM in KIELER which provides a user interface, shown in the bottom view in Figure 6.2, and visual feedback about simulation details, both in the graphical model view itself, and in a separate data table view.

Model transformations play a key role in generative software development. These describe the transformation of models (i. e., metamodel instances) that conform to one metamodel into models which then conform to another or even the same metamodel. The implementation uses the

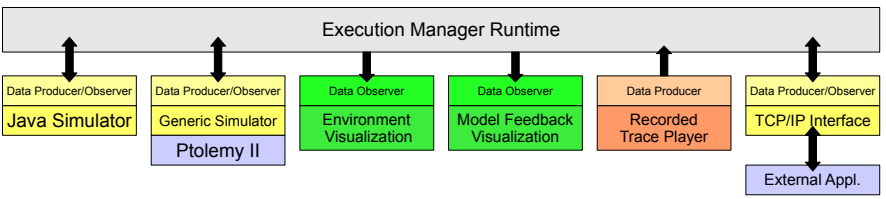


Figure 6.1. Schematic overview of the Execution Manager infrastructure

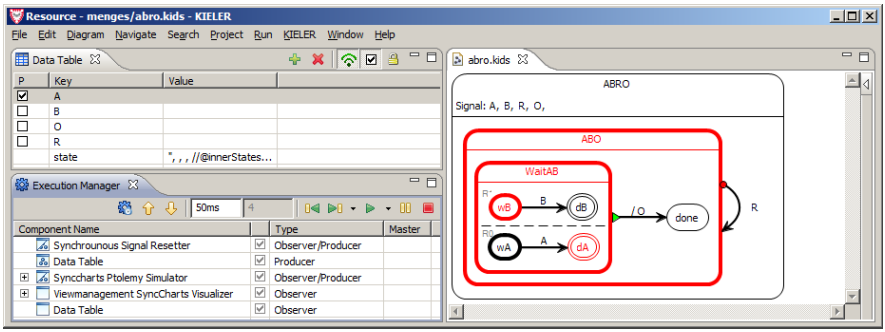


Figure 6.2. GUI of KIELER and the KIEM Eclipse plug-in during a simulation run

Xtend language, as it is widely used and was refactored for a seamless integration into the Eclipse IDE. Additionally, its extensibility features allow to escape to Java for sequential or complex transformation code fragments. Nevertheless the approach is conceptually open to use any transformation language which supports EMF meta models.

6.1.1 KIELER Execution Manager (KIEM)

As a subproject of KIELER the Execution Manager (KIEM) implements an infrastructure for the simulation and execution of domain specific models and possibly graphical visualizations. It does not do any simulation computation by itself but bridges simulation components, visualization components and a user interface to control execution within the KIELER application, as indicated in Figure 6.1. These components can simply be constructed using the Java language implementing some commonly defined interfaces.

An approach on how to implement such simulation engines themselves using model transformations and Ptolemy as a simulation backend will be presented in subsection 6.1.2.

DataComponents

`DataComponents` are the building blocks of executions in the KIEM framework. These components are pure Java code that is restricted to meet a special interface so that the Execution Manager is able to address all components and interact with them in the same way. They use data in order to interact with each other. Hence, they may produce data addressed for other `DataComponents` or observe data from other components or even both at once. See again Figure 6.1 for an example setup. It shows several example data components like a basic Java simulator or a more abstract Ptolemy simulator and also components that only visualize data either in the model itself or in a separate view of the model's environment.

`DataComponents` can be classified according to their type of interaction into multiple categories:

- *Pure observer* `DataComponents` do not produce any data which for example is the case for simulation visualizations.
- *Pure producer* `DataComponents`, like user input facilities, do not observe any data. Hence, they are data independent of others.
- Often there are *observing and producing* `DataComponents` like simulation engines that react to input with some output.

Figure 6.3 shows the simple and self-explanatory interface for `DataComponents`. A component needs to declare whether it is an observer or a producer of data. It should declare some initialization and wrapup code. The `step` method is most significant in this interface. It should implement the execution behavior of the `DataComponent`. The parameter value holds all input data in case of an observer component. The return value should hold all output data in case of a producer component.

User Interface

Figure 6.2 shows the GUI of the Execution Manager. Listed are all `DataComponents` that take part in the execution. The order of the `DataComponents` in the list is the one in which they are scheduled. Together with

```
1   public interface IDataComponent {
2
3     void initialize() throws KiemInitializationException;
4     void wrapup() throws KiemInitializationException;
5
6     boolean isProducer();
7     boolean isObserver();
8
9     JSONObject step(JSONObject jSONObject)
10                              throws KiemExecutionException;
11  }
```

Figure 6.3. DataComponent Interface

(optional) property settings the list of `DataComponents` forms a savable *execution setting*. The execution can be triggered by the user by pressing one of the active control buttons (e. g., step, play, or pause). The step button allows a stepwise, incremental execution while in each step all `DataComponents` are executed at most once (see below). The lower bound on a step duration can be set in the UI, while the upper bound depends on the set of all producer `DataComponents`.

Data Pool and Scheduling

Data are exchanged by `DataComponents` in order to communicate with each other. The Execution Manager collects and distributes sets of data from and to each registered (w.r.t. the Eclipse plug-in concept) `DataComponent`. Therefore it needs some kind of memory for intermediate storage to reduce the overhead of a broadcast, and to restrict and decouple the communication providing a better and more specific service to each single `DataComponent`.

This storage is organized in a data pool where all data are collected for later usage. The Execution Manager only collects data from components that are producers of data. Whenever it needs to serve an observer `DataComponent`, it extracts the needed information from its data pool, transparent to the component itself.

All components are called by the Execution Manager in a linear order that can be defined by the user in an *execution setting*. Because the execution is an iterative process—so far only iteratable simulations are supported—all components (e. g., a simulation engine or a visualizer) should also preserve this iterative characteristic. During an execution, KIEM will stepwise activate all components that take part in the current execution run and trigger them to produce new data or to react to current data. As KIEM is meant also to be an interactive debugging facility, the user may choose to synchronize the iteration step times to real-time. However, this might cause difficulties for slow `DataComponents` as discussed below.

All components are executed concurrently. This means that they are executed in their own threads. For this reason, `DataComponents` should communicate (e. g., synchronize) with each other via the data exchange mechanism provided by the Execution Manager only to ensure thread safety. There are also additional scheduling differences between the types of `DataComponents` listed above. These concern two facts: First, `DataComponents` that only produce data do not have to wait for any other `DataComponent` and can start their computation immediately. Second, `DataComponents` that only observe data often do not need to be called in a synchronous blocking scheme since no other `DataComponents` depend on their (nonexistent) output.

Further Concepts

Besides the described basic concepts of the Execution Manager there are some facilities and improvements that are summarized in the following.

Analysis and Valdiation: For analysis and validation purposes it is easy to include *validation DataComponents* that observe special conditions related to a set of data values within the Data Pool. These components may record events in which such conditions hold or may even be able to pause the execution to notify the user.

Extensibility: The data format chosen in the implementation relies on the Java Script Object Notation (JSON). This is often referred to as a simplified and light-weight XML. It is commonly used whenever a more efficient

Figure 6.4. A simple Java ME application to connect a mobile device to KIEM [Mot09].

data exchange format is needed. Due to its wide acceptance many implementations for various languages exist, thus aiding the extensibility of the Execution Manager.

Although `DataComponents` need to be specified in Java, the data may originally stem from almost any kind of software component, e. g., an online-debugging component of an embedded target. With this approach the Java `DataComponents` do not need to reformat the data and can simply act as gateways between the Execution Manager and the embedded target.

As an example there is a mobile phone Java ME[1] application that can fully interact with the Execution Manager, see Figure 6.4.

Co-Simulation: Co-operative simulation allows the execution of interacting components run by different simulation tools. For each different simulation tool a specific interface `DataComponent` just needs to be defined. This way Matlab/Simulink for example could co-simulate with a SyncCharts model and an online-target debugging interface to get a model- and hardware-in-the-loop setup, which is useful for designing

[1] Java Micro Edition Framework: `http://java.sun.com/javame`

embedded/cyberphysical systems.

History: Together with the Data Pool the built-in history feature comes for free. This enables the user to make steps backwards into the past. `DataComponents` need to explicitly support this feature: For example one may not want a recording component to observe/record any data again when the user clicks backwards. This feature may help analyzing situations better. For example, when a validation observer `DataComponent` pauses the execution because a special condition holds, one may want to analyze how the model evolved just before. This assists during interactive debugging sessions.

6.1.2 KIELER leveraging Ptolemy Semantics (KlePto)

KIEM does not do any simulation computation by itself but bridges simulation components that do actual visualization or simulation computations.

A concrete implementation of a simulation `DataComponent` and thus a generic way to specify execution semantics for a DSL is presented in the following.

As discussed in the related work Chapter 2, there are two possible ways to specify semantics of a DSL, where the second approach gets used here in leveraging Ptolemy II as a flexible and extensible simulation backend. Due to the flexibility of Ptolemy we have already encountered it in several sections. Especially in Section 5.4 we have learned about it as the target of automatic layout. Now we will use its simulation capabilities and therefore see it from a different perspective. Hence, in the following we will give a short introduction into Ptolemy from this perspective, which we use as an example semantic domain, and afterwards give some brief overview of a concrete case study.

Ptolemy

The Ptolemy II project studies heterogeneous modeling, simulation, and design of concurrent systems with a focus on systems that mix computational domains [EJL+03].

The behavior of reactive systems, i. e., systems that respond to some input and a given configuration with an output in a real-time scenario, is modeled in Java with executable models. The latter consists of interacting components called *actors*, hence, this approach is referred to as *actor-oriented* design. These actors can be interconnected at their ports. Ptolemy actors can be encapsulated into composed actors introducing a notion of hierarchy. Ptolemy models strictly try to separate the syntax and the semantics on one modeling layer. The first is given by the structural interconnection of all used actors. The second is encapsulated in a special and mandatory *director* actor that specifies the way of actor interaction and scheduling. Ptolemy allows models to mix different models of computations (MoCs) on different hierarchy layers. Actors consist of

- pure Java code that may produce output for some input during execution, or
- other Ptolemy actors composed together under a separate Model of Computation (MoC) that defines the overall in- and output behavior.

There exist several built-in directors that come along with Ptolemy II, such as Continuous Time (CT), Discrete Events (DE), Process Networks (PN), Synchronous Data Flow (SDF), Synchronous Reactive (SR) and Finite-State-Machines (FSM). Whenever this seems to limit the developer, one may adapt or define new Ptolemy II directors in Java that implement their own more specialized semantic rules of component interaction. The combination of these various, extendable domains allows to model complex systems with a conceptually high abstraction leading to coherent and comprehensible models. An example Ptolemy II model is presented in Figure 6.6.

We will not discuss the technical details here and refer to the Ptolemy documentation, in particular about the *charts (pronounced starcharts) principle [GLL99]. It illustrates how hierarchical Finite-State-Machiness can be composed using various concurrency models leading to arbitrarily nested and heterogeneous model semantics. This technique is employed here to emulate a Statechart dialect, thus specifying the semantics and producing executable model representations.

Figure 6.5 shows the underlying concept where the description of the transformation is defined in the file `dsl2pto.xtend` using the *Xtend* language.

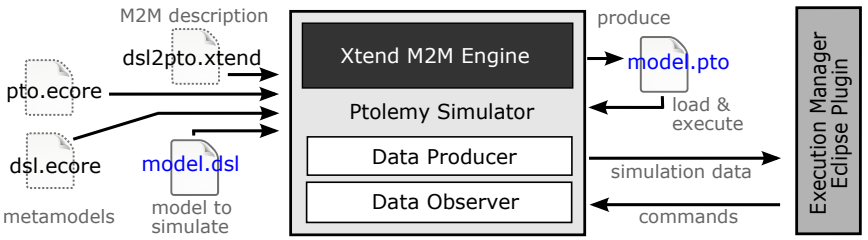


Figure 6.5. Abstract transformation and execution scheme

The latter operates on EMF metamodel instances, hence the transformation itself must be defined referencing two metamodels. The *source metamodel* dsl.ecore stems from the EMF tool chain that already exists after defining the DSL's abstract syntax. The *target metamodel* pto.ecore describes the language of all possible Ptolemy models and is common for all DSLs.

The actual transformation of the source model model.dsl into the target Ptolemy model model.pto is done by the Xtend transformation framework. Instrumentation code is injected during the Ptolemy model composition that allows to easily map Ptolemy actors back to their corresponding original model elements. The Ptolemy Simulator itself is part of the execution runtime interface and can load and run Ptolemy models and interact with the Execution Manager in form of a `DataComponent` as introduced in the section before.

SyncCharts

The *Statecharts* formalism of David Harel [Har87], which extends *Mealy machines* with hierarchy, parallelism, and signal broadcast, is a well known approach for modeling control-intensive tasks. *SyncCharts* were introduced almost ten years later [And96], evolving from ARGOS [Mar91] as an adoption to the synchronous world. They serve as a graphical representation of the Esterel language [BC84] following the same execution semantics.

SyncCharts simplify the modeling of complex reactive systems because they allow to model deterministic concurrency and preemption. However, they are more difficult to execute compared to other state machine models.

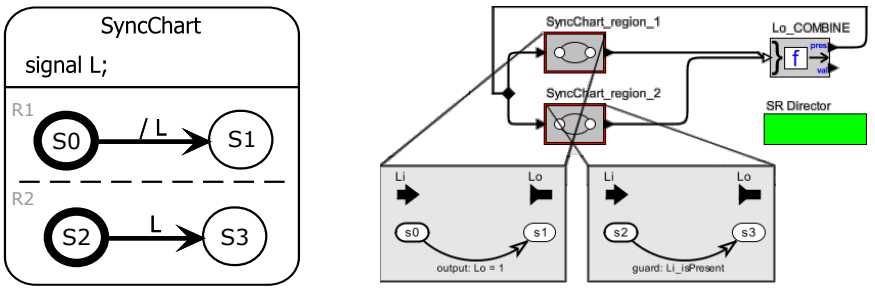


Figure 6.6. A SyncChart model (left) and the generated Ptolemy model (right)

As a challenging example the semantics of SyncChart EMF models are defined in an M2M transformation, mapping each element to Ptolemy actors utilizing the combination of the Synchronous Reactive and Finite-State-Machines domains.

Transformation The main idea is to represent the hierarchical layers by Ptolemy composite actors. These are connected by links incorporating the signal broadcast mechanism.

The SR fixed-point semantics guarantees finding a fixed point for the signal assignment w.r.t. the signal coherence rule. The latter means that each SyncCharts signal can either be present or absent in a synchronous tick instant but not both at once. For each tick, the fixed point computation in SR starts with unknown signal states on all data links.

The SyncCharts example of Figure 6.6 shows a broadcast communication between two parallel regions `R1` and `R2`. `R1` emits the signal `L` by taking an enabled transition from initial state `S0` to state `S1` guarded by an implicit *true* trigger. `R2` waits in its initial state `S2` for the signal `L` to be present in order to take the transition to state `S3`.

The structural transformation ensures that for each parallel region, every signal is represented as an input and output port (e. g., `Li` and `Lo`) because conceptually each region can emit a signal or may react to a present signal, or even both. The latter implies the requirement of a feedback structure for each signal using a special combine actor. In the Ptolemy model, the

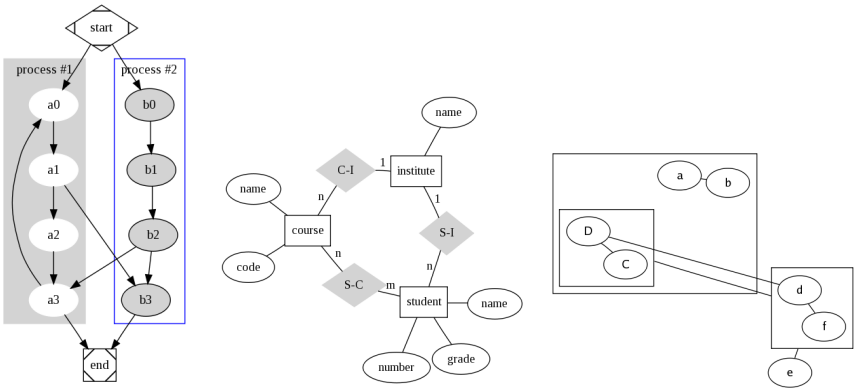
presence of a signal is represented by a data token traveling across the dedicated link. The absence of a signal is equivalent to a special clear operation on a channel. If the combine actor receives a token of any parallel region, it immediately outputs a token to the feedback loop. If the combine actor on the other hand notices a clear on each connected incoming channel, it also clears its output. This reflects the fact that a signal is present iff it is emitted in a tick instance and a signal is absent iff it is not emitted anywhere.

In the generated Ptolemy model of Figure 6.6, the concurrent actor SynChart_region_1 will produce an output token that will be received and forwarded by the combine actor. Finally, a duplicate of this token reaches the actor SyncChart_region_2 triggering a state transition.

Further concepts of the transformation description consider the aspect of hierarchy: Concurrent Ptolemy actors that represent parallel regions contain FSM nodes that are either refined in case of original SyncCharts *macro states* or not refined in case of original SyncCharts *simple states*. The refinements can again contain concurrent actors representing regions within such a *macro state*. Because signals within a SyncCharts state can be emitted in any inner state, their dedicated ports are replicated in the transformation process for all lower hierarchy layers.

The Ptolemy expression language allows the evaluation of complex triggers that for example are a Boolean combination of signal presence values. This makes it straight forward to support the last special SyncChart concept of compound events in Ptolemy models.

The mapping takes also place during the model transformation process as it links the EMF model elements with attributes of the generated Ptolemy model elements. This simulation engine is interfaced with the Execution Manager presented in subsection 6.1.2, as depicted in Figure 6.5. It will process input and output signals and also collect additional output information such as the current active states. The information and the signal data are used to visualize the simulation and feed a Data Table, as shown earlier in Figure 6.2.



(a) Layer-based Dot **(b)** Energy calculating Neato **(c)** Force-directed FDP

Figure 6.7. Different layout algorithms in GraphViz.

6.2 Automatic Layout Algorithms

In this section we discuss two different approaches to automatic layout algorithms. The first one interfaces an existing C library to KIML, while the second one implements a specialized algorithm from scratch.

6.2.1 GraphViz

GraphViz[2] is a layout library containing different layout algorithms implemented in C [Gan04]. Some direct tool results are shown in Figure 6.7.

Available algorithms are:

Dot A layer-based algorithm following the original ideas of Sugiyama [STT81, GKNV93].

Neato An energy-based algorithm according to Kamada and Kawai [KK89].

FDP A force-minimizing algorithm similar to Neato but implementing the approach of Fruchterman and Reingold [FR91].

[2] `http://www.graphviz.org`

Twopi A radial layouter after Graham Wills [Wil97]. The nodes are placed on concentric circles depending on their distance from a given root node.

Circo A circular layouter after Six and Tollis [ST99] and Kauffmann and Wiese [KW02]. It is suitable for certain diagrams of multiple cyclic structures.

The tool is implemented in C and available open-source as well as in binary packages containing executable command-line tools. These tools take a textual language, the DOT notation, as input. It specifies the general graph structure with a special notation where arbitrary attributes can be added to the graph elements to add style information such as colors, shapes, etc. The output is the same textual file, which gets augmented by the concrete layout information, i. e., position and sizes of nodes and bendpoints of edges. Additionally it renders image files such that the tools are commonly used to create diagrams from textual notations, close to the proposal in Section 4.6.

Interfacing Java (KIELER) and C (GraphViz) is not a trivial task. Java provides the Java Native Interface (JNI) to interface with C code. This works well for small C libraries that do not interfere with the operating system. However, exceptions and especially memory violations in the connected C-code will likely lead to a shutdown of the current process. In the case of the JNI it is the same process as the Java Virtual Machine (JVM) and therefore the Java application gets terminated, too, and the exception cannot be recovered. Additionally, the C program can only access the environment that its Java host provides. This is especially problematic for environment variables of the operating system. Since version 1.4 of Java, the environment can only be read by Java code but not set. Therefore, using JNI is rather brittle and especially with GraphViz these problems emerged. GraphViz requires environment variables for configuration and thus using the JNI was discouraged by the GraphViz developers themselves.

Therefore GraphViz is started in another process to interface it with KIML. A GraphViz process reads textual graph specifications from `stdin` and writes its results to `stdout`. Therefore KIML has to create this textual representation of the KGraph, pass it to the GraphViz process and parse the resulting text and apply the enriched information back to the KGraph.

Serializing and parsing DOT notation is done using a grammar of DOT implemented in the Eclipse textual DSL framework Xtext. From this grammar, Xtext generates parser and serializer which can conveniently be used to handle the DOT language. The only drawback is the rather big runtime overhead of Xtext. Especially Eclipse's lazy loading mechanism of plug-ins leads to a significant caching effect when using GraphViz layout for the first time during an Eclipse session. At the second layout operation, the layout is fast and ready for interactive use.

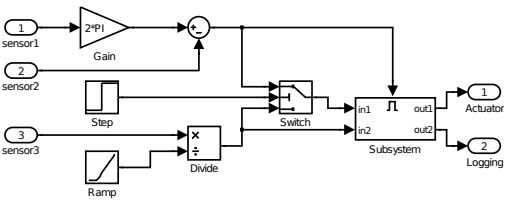
In the following we will discuss a specialized layout implementation from scratch.

6.2.2 Actor-Oriented Data Flow Models with Port Constraints

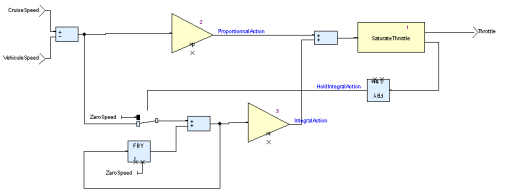
An important class of modeling diagrams are *data flow diagrams*, which are graphical representations of *data flow models* for the design of complex systems. Applications of data flow diagrams can be found in modern software and hardware development tools. Some of these are Simulink (The MathWorks, Inc.), labVIEW (National Instruments Corporation), ASCET (ETAS Inc.), SCADE (Esterel Technologies, Inc.) and the Ptolemy project [EJL+03]. Example diagrams are presented in Figure 6.8.

A data flow model is described by a directed graph where the vertices represent *operators* or *actors* that compute data and the edges represent data paths [DK82, EJL+03]. Such a data path has a specified *source port* where data is created and a *target port* where data is consumed. A source port may be connected with multiple target ports, thus forming a *hyperedge*. Furthermore, the data flow paths are required to be drawn orthogonally. These properties of data flow diagrams can be defined as a set of constraints for the drawing of the corresponding directed graph.

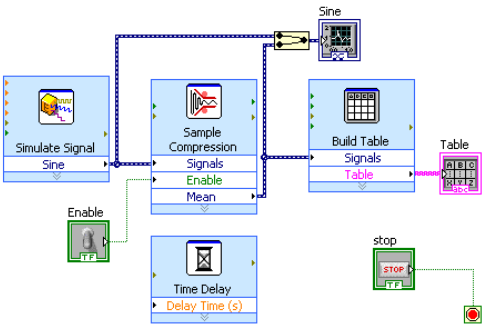
The implementation follows the *layered* approach for graph drawing, and is extended for the special requirements of data flow diagrams.



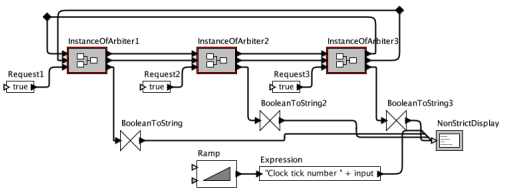
(a) Simulink



(b) SCADE



(c) LabVIEW



(d) Ptolemy

Figure 6.8. Data flow diagrams from graphical modeling tools

The Layered Approach for Graph Layout

The layered approach, which is also called *hierarchical* layout method, works only for directed graphs and aims at emphasizing the direction of flow, thus expressing the hierarchy of vertices in the graph. It was proposed by Sugiyama, Tagawa and Toda [STT81], and is often called *Sugiyama* layout.

This section provides basic definitions for graphs and drawings of graphs, an overview of the hierarchical layout algorithm, some details on the implementation in KIELER, and experimental results.

Graph Drawing

A *directed graph* $G = (V, E)$ consists of a finite set V and a multiset $E \subseteq V \times V$. The elements of V are called *vertices* or *nodes*, and the elements of E are called *edges* or *connections*. An edge $e \in E$ with $e = (v, v)$ is called a *self-loop*. An edge whose multiplicity in E is greater than one is called a *multiple edge*. The vertices u, v of an edge $e = (u, v)$ are called its *endpoints*. If there exists an edge $e = (u, v) \in E$, we call u and v *adjacent* to each other and e *incident* to u and v. The *neighbors* of a vertex v are its adjacent vertices. The *degree* of v is the number of edges which are incident to v. An edge $e = (u, v) \in E$ is an *outgoing edge* of u and an *incoming edge* of v. $v_s(e) := u$ is called the *source* of e, and $v_t(e) := v$ is called the *target* of e. The *indegree* of a vertex v is the number $|E_i(v)|$ of its incoming edges $E_i(v)$, and its *outdegree* is the number $|E_o(v)|$ of outgoing edges $E_o(v)$. A vertex with no outgoing edges is called a *sink* of the graph, and a vertex with no incoming edges is called a *source* of the graph. A *subgraph* of $G = (V, E)$ is a graph $G' = (V', E')$ for which $V' \subseteq V$ and $E' \subseteq \{(u, v) \in E : u, v \in V'\}$.

A *path* of a graph is a sequence $(v_1, \ldots, v_k)$ of vertices such that $(v_i, v_{i+1}) \in E$ for $i \in \{1, \ldots, k-1\}$. A path $p = (v_1, \ldots, v_k)$ is called *simple* if $v_i \neq v_j$ for all $i \neq j$. p is a *cycle* if $v_1 = v_k$. A cycle $(v_1, \ldots, v_k)$ is called *simple* if $(v_1, \ldots, v_{k-1})$ is a simple path. A graph G is *acyclic* if it contains no cycles. It is *connected* if for each pair (u, v) of vertices there is a path between u and v in G. The *connected components* of G are the maximal connected subgraphs of G.

A *drawing* of a graph G is a mapping Γ of the vertices and edges of G to subsets of the plane $\mathbb{R}^2$. A drawing is called *polyline* if the drawing $\Gamma(e)$

of each edge e can be decomposed into a sequence of straight lines, and it is *orthogonal*, or *rectilinear*, if all line segments are aligned horizontally or vertically.

Algorithms for automatic layout are programs that compute drawings of the related graphs. These drawings are represented by abstract values such as the position and size of each vertex, and the list of bend points of each edge. Aside from general restrictions and drawing conventions, algorithms for automatic layout are subject to the goal of optimizing a set of *aesthetics criteria* [Pur02, DETT99]. The most important to mention are the following:

Crossings Minimize the total number of crossings between edges.

Direction Maximize the number of edges pointing to a specific direction, e. g., to the right.

Bends Minimize the total number of bends along the edges.

Area Minimize the total area of the drawing while preserving a minimal distance between all objects.

AspectRatio Keep the aspect ratio low, that is the width of the drawing divided by its height for landscape format drawings, and the inverse for portrait format.

Port Constraints

A *port based graph* is a directed graph $G = (V, E)$ together with a finite set P of *ports*. For each $v \in V$ we write $P(v)$ for the subset of ports that belong to v, and we require $P(u) \cap P(v) = \varnothing$ for $u \neq v$. Each edge $e = (u, v) \in E$ has a specified *source port* $p_s(e) \in P(u)$ and a *target port* $p_t(e) \in P(v)$. We write $v(p)$ for the vertex u for which $p \in P(u)$.

In general graph drawing it is sufficient that the drawing of each edge $e = (u, v)$ touches the drawings of u and v anywhere on their border. For port based graphs the drawing of each port $p \in P(v)$ has a specific position on the border of $\Gamma(v)$, and the edges that have p as source or target port may touch $\Gamma(v)$ only at that position.

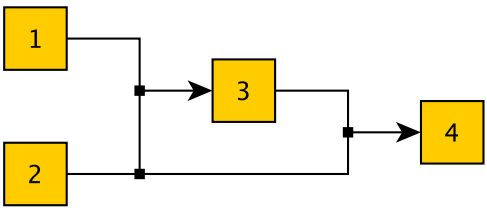


Figure 6.9. A hyperedge that connects four vertices

Some new aspects must be considered when extending a graph layout algorithm to handle ports. Firstly, edges that are incident at the same port are considered as *hyperedges*, i. e., edges that may connect more than two vertices. This aspect is handled by merging some line segments of edges that share the same port, as seen in Figure 6.9, and is mainly a matter of proper edge routing. A second aspect concerns port positions, for which we consider four different scenarios:

FreePorts All ports may be drawn at arbitrary positions on the border of their corresponding vertex.

FixedSides The side of the vertex is prescribed for each port, i. e., the top, bottom, left, or right border, but the order of ports is free on each side.

FixedPortOrder The side is fixed for each port, and the order of ports is fixed for each side.

FixedPorts The exact position is fixed for each port.

When structural hierarchy is applied we use *compound graphs*, where a vertex v is allowed to contain a nested graph G_v. In this case, the ports of v are treated as *external ports* of G_v, and may be connected to the vertices of G_v (see Figure 6.10). As opposed to the ordinary vertices of G_v, the external ports cannot be assigned arbitrary positions, but must stay on the border of v. Additional edge routing mechanisms must be applied to properly connect the external ports with inner vertices.

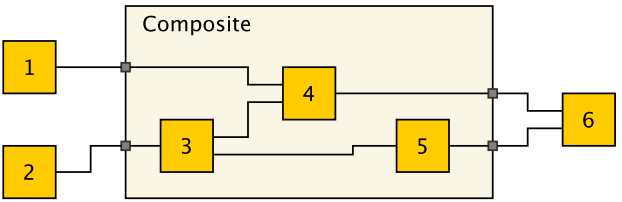


Figure 6.10. The diagram contained in Composite has connections to external ports.

The Algorithm

The main phases of the hierarchical layout algorithm with port constraints are the following.

1. **Cycle removal:** Break directed cycles by reversing some edges, while keeping the number of reversed edges as low as possible. In the final drawing the reversed edges are restored again, so that they point against the predominant direction of flow.

2. **Layer assignment:** Create a minimal set of *layers* $L_1, \ldots, L_k$ and assign a layer to each vertex such that for all edges (u, v) the assigned layers L_i of u and L_j of v satisfy $i < j$. This is possible because after the first phase the graph is acyclic.

3. **Crossing reduction:** Find an ordering of the vertices of each layer that minimizes the number of edge crossings. If the order of ports is not fixed for any vertex, it must also be properly chosen.

4. **Edge routing A:** Depending on port positions, some edges need to be routed around vertices (see Figure 6.11). The number and order of edges that need to be routed on each side of a vertex is determined in this phase.

5. **Node placement:** Determine exact positions of all vertices inside their corresponding layers. The vertices must not overlap each other, the ordering from phase 3 must be respected and the position of each vertex must be well-balanced with respect to its neighbors. We will call this *crosswise* placement.

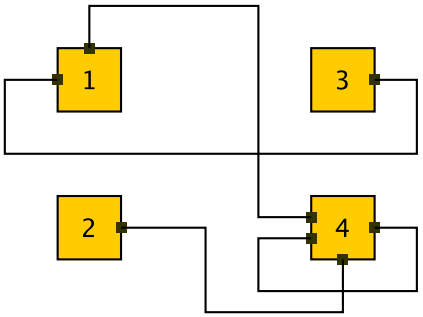


Figure 6.11. Routing of edges around vertices due to prescribed port positions

6. **Edge routing B:** Determine bend points for each edge and the exact distance between subsequent layers, which we will call *lengthwise* placement. Routing to external ports is also handled in this phase.

There are numerous alternative algorithms that can be used for each phase [Spö09, DETT99, KW01], but this report focuses on the current implementation of each phase and on the realization of port constraints in this KIELER *Layout of Data Flow Diagrams* (KLoDD).

Implementation The main class of the layout algorithm is a subclass of `AbstractLayoutProvider` to match the interface for automatic layout described in Section 4.2. Therefore, the input of the algorithm is an instance of `KNode` (see Figure 4.6a), whose direct children represent the graph for which layout is performed.

The phases of the algorithm are modularized using the *Strategy* design pattern [GHJV95]: an interface that describes the functionality of the module is created for each phase, and at least one implementation is given for each phase. If there is more than one implementation for an interface, the user may choose from different alternatives for the corresponding phase of the algorithm, possibly leading to differing layout results. With this design pattern it is also possible to experiment with new implementations while keeping the original implementation, and to directly compare their outputs.

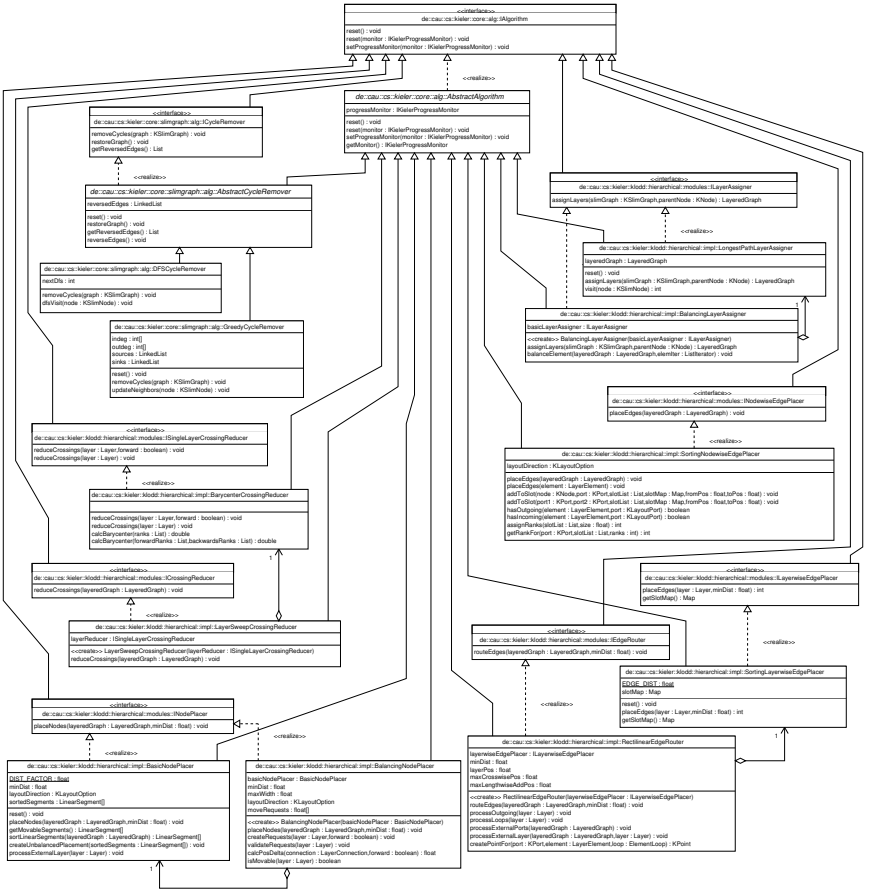


Figure 6.12. Modules for the hierarchical layout algorithm

Figure 6.12 shows a class diagram with all modules of the layout algorithm and their currently available implementations.

Each implementation of a module is a subclass of AbstractAlgorithm, which handles the usage of progress monitors (see Section 4.2). These progress monitors cannot only be used to give feedback of the progress of

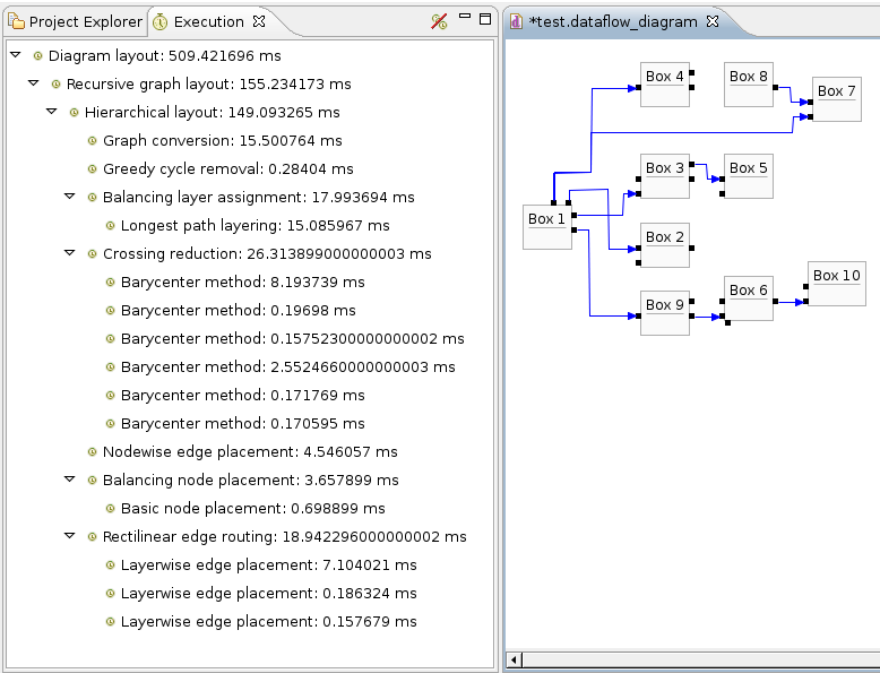


Figure 6.13. Execution time of each module of the algorithm, shown in an Eclipse view after layout was applied to a data flow diagram

the algorithm during its execution, but can also measure execution times. As seen in Figure 6.13, execution times are tracked for the whole algorithm as well as for each module.

Since for the layered approach the vertices of the input graph are organized in *layers* (see Section 6.2.2), a graph structure that directly expresses this layering is used internally by the layout algorithm.

Cycle removal The goal of this phase is to find a minimal set of edges of a given graph G for which the graph obtained by reversing these edges is acyclic. This problem is equivalent to the *feedback arc set problem*, which is NP-complete [GJ79]. A good heuristic is the algorithm *Greedy-Cycle-Removal*

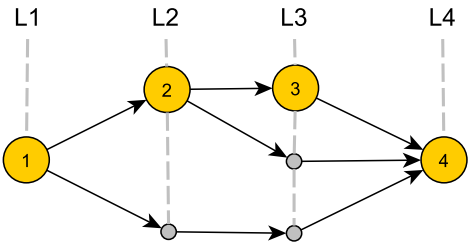


Figure 6.14. A layered graph with two dummy vertices for the long edge (1,4) and one for the edge (2,4)

from Di Battista et al. [DETT99], which determines an ordering $v_1, \ldots, v_n$ of the vertices in G. By reversing all edges (v_i, v_j) for which $i > j$, all cycles are eliminated.

Since we want to avoid changing the KGraph structure given as input, the graph is first transformed to a much simpler graph structure called SlimGraph, which is not implemented in EMF, but as a set of plain Java classes. Edges are only reversed in the SlimGraph instance for cycle removal.

Layer assignment In this step we want to find layers $L_1, \ldots, L_k$ for the vertices of the acyclic graph G. A layering is called *proper* if all edges e connect only vertices from subsequent layers. A proper layering is constructed from a general layering by splitting *long edges*: given an edge $e = (v_i, v_j)$, $v_i \in L_i$, $v_j \in L_j$, for which $j - i > 1$, we add new dummy vertices $v_{i+1}, \ldots, v_{j-1}$ to the layers $L_{i+1}, \ldots, L_{j-1}$ and split e into a series of edges $e_i, \ldots, e_{j-1}$ such that $e_h = (v_h, v_{h+1})$ for all $h \in \{i, \ldots, j-1\}$ (see Figure 6.14). The *rank* of a layer L_i is $r(L_i) := i$, and its *height* is $h(L_i) := k - i + 1$.

A simple and linear running time heuristic consists in determining the longest path to a sink: all sinks s are put into the last layer, and all other vertices are assigned a layer of height $h(L_i)$ equal to the number of edges on a longest path to a sink plus one. If there are many sinks in the graph, the last layer can become very wide with this layering. Therefore we improve the longest path layering using Algorithm 6.1, which decides locally for each vertex whether moving it to a preceding layer could improve the

Listing 6.1. balanceLayering

```
1  procedure balanceLayering(G: directed graph)
2      determine layers L_1,...,L_k for G using longest path layering
3      foreach layer L_j, j ⩾ 3, do
4          foreach v ∈ L_j, indegree of v ⩾ outdegree of v, do
5              r := max{i : (u,v) ∈ E, u ∈ L_i} + 1
6              foreach layer L_i, r ⩽ i < j with increasing i, until a
                   fitting layer is found, do
7                  if |L_i| ⩽ |L_j| then
8                      move v to the fitting layer L_i
9  end
```

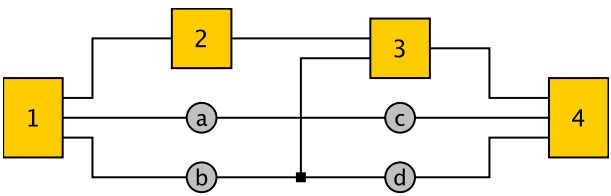


Figure 6.15. The long edges $(1,3)$ and $(1,4)$ share the dummy vertex b in layer 2.

layering, thus greedily computing a local optimum.

The input of this phase is the SlimGraph instance created for cycle removal together with the original KNode instance. The layering algorithm creates and returns a layered graph, which then requires some post-processing through the method createConnections. This method traverses the layered graph and creates layer connections for all edges that are found in the original graph. This is done by Algorithm 6.2, which also splits connections that span over multiple layers using dummy vertices, for which new layer elements are created. If two long edges share a common port, thus forming a hyperedge, they must be assigned common dummy vertices, as seen in Figure 6.15.

If the original diagram contains external ports, they are also added as layer elements: input ports, which have only outgoing connections, are

Listing 6.2. createLayerConnection

```
1  procedure createLayerConnection(L_1,...,L_k: layers, e: edge)
2      let L_s be the layer for which v_s(e) ∈ L_s
3      let L_t be the layer for which v_t(e) ∈ L_t
4      if t − s = 1 then
5          directly connect v_s(e) and v_t(e)
6      else
7          // Associations between ports and existing linear segments
               are created in line 27
8          get the linear segment S associated with the source port
               p_s(e)
9          if S =⊥ then
10             get the linear segment S associated with the target
                   port p_t(e)
11         if S =⊥ then
12             create a new dummy node d in L_i, i := s + 1
13             create a linear segment S for d
14             connect v_s(e) and d
15         else
16             // Another edge with the same source or target port
                   exists
17             connect v_s(e) and the dummy node in S whose layer is L_{s+1}
18             find the dummy node d in S whose layer L_i has maximal
                   i < t
19
20         while i < t − 1 do
21             create a new dummy node d′ in L_{i+1}
22             add d′ to S
23             connect d and d′
24             d := d′, i := i + 1
25
26         connect d and v_t(e)
27         associate S with p_s(e) and p_t(e)
28 end
```

assigned the new layer with rank 0, while output ports, which have only incoming connections, are assigned the layer with height 0. By this we

achieve that external ports can be treated as normal vertices in the following phases of the algorithm, and they are assigned dedicated layers.

Crossing reduction The problem of crossing reduction for layered graphs, which consists in setting an order of vertices that minimizes the number of crossings for each layer, is NP-complete, even if there are only two layers [GJ83]. Nevertheless it is easier to find heuristics to set the order of vertices for two layers than to optimize the whole graph at once. For this reason this phase is usually solved with a *layer-by-layer sweep*: choose an arbitrary order for layer L_1, then for each $i \in \{1, \ldots, k-1\}$ optimize the order for layer L_{i+1} while keeping the vertices of layer L_i fixed. Afterwards the same procedure is applied backwards, and it can then be repeated for a specified number of iterations. We will only cover the forward sweep here, because the backwards case is symmetric.

When ports are used to determine the source and target point of each edge, the number of crossings does not depend only on the order of vertices, but also on the order of ports for each vertex. This order is implied by the port ranks which are assigned to the ports using layout options (see Section 4.2) and are based on clockwise order. The port ranks must be translated depending on the side of the node, the overall layout direction, and whether a forward or backwards layer-by-layer sweep is performed. For example, if horizontal layout is performed, we must consider clockwise port ranks for a forward sweep, but counter-clockwise port ranks for a backwards sweep.

Based on the translated port ranks we define extended vertex ranks so that for each $v \in L_i$ and $p \in P(v)$ the sum of the rank of v and the rank of p is unique. The *rank width* of a layer element $v \in L_i$ is $w(v) := |P(v)|$ if v originates from a vertex, and $w(v) := 1$ if v was created for a dummy vertex of a long edge or for an external port. The extended vertex ranks of the ordered vertices $v_1, \ldots, v_h$ in the layer L_i are defined as

$$r(v_j) := \sum_{g<j} w(v_g)$$

for all $j \leqslant h$.

We implemented the *Barycenter* method for the two-layer crossing problem: first calculate values $a(v) \in \mathbb{R}$ for each $v \in L_{i+1}$, then sort the vertices in L_{i+1} according to these values. The $a(v)$ values are determined as the average of the combined vertex and port ranks for all source ports of incoming edges of v:

$$a(v) := \frac{1}{|E_i(v)|} \sum_{(u,v) \in E_i(v)} (r(u) + r(p_s(u,v)))$$

Vertices v_j that have no incoming edges should be assigned values $a(v)$ that respect the previous order of vertices, thus we define $a(v_j) := \frac{1}{2}(a(v_{j-1}) + a(v_{j+1}))$ if $E_i(v_{j+1}) \neq \varnothing$ and $a(v_j) := a(v_{j-1})$ otherwise. By setting $a(v_0) := 0$ and calculating the missing $a(v_j)$ values with increasing j we can assure that $a(v_{j-1})$ is always defined.

For vertices with FixedSides or FreePorts port constraints we have the additional task of finding an order of ports for each vertex that minimizes the number of crossings. The extension of the method described above is quite straightforward: instead of calculating values $a(v)$ to order the vertices, calculate values $a(p)$ to order the ports first, then calculate

$$a(v) := \frac{1}{|P(v)|} \sum_{p \in P(v)} a(p).$$

For each port p let $E_i(p)$ be the set of edges which are incoming at that port. Then we define

$$a(p) := \frac{1}{|E_i(p)|} \sum_{(u,v) \in E_i(p)} (r(u) + r(p_s(u,v))) .$$

If there are long hyperedges that share common dummy vertices, as described in Section 6.2.2, crossing reduction must be adapted to avoid inconsistencies in the following phases. If, for example, backwards crossing reduction is performed for the second layer of the graph in Figure 6.15 while keeping the vertices of the third layer fixed as $(3, \mathsf{c}, \mathsf{d})$, it can happen that the dummy vertex b is placed above a because of its outgoing connection to vertex 3. This would lead to a crossing of the edges (a, c) and (b, d), which

is not allowed for proper vertex placement.

To resolve this problem, two new rules must be added for each long edge that is split into dummy vertices $v_1, \ldots, v_k$:

1. For each dummy vertex v_i, $i \in \{2, \ldots, k\}$, only one incoming connection may be considered for crossing reduction, namely (v_{i-1}, v_i).

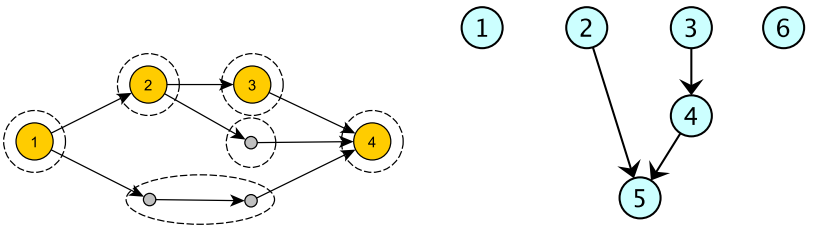
2. For each dummy vertex v_i, $i \in \{1, \ldots, k-1\}$, only one outgoing connection may be considered for crossing reduction, namely (v_i, v_{i+1}).

Node placement

From this phase on we will cover only horizontal layout direction, but the concepts for vertical layout are symmetric.

For crosswise vertex placement in horizontal layout the vertices of each layer are arranged vertically. Sander proposes a two-phase method [San96a]: determine a correct initial placement, then balance vertex positions. For this purpose the concept of *linear segments* is introduced; here a linear segment is a set which contains either a single regular vertex or all dummy vertices introduced to split a single long edge (see Figure 6.16). It is important to put multiple dummy vertices of a linear segment at the same vertical position, so that the associated long edge does not receive too many bend points. For each vertex v we write $S(v)$ for the linear segment for which $v \in S(v)$.

The *segment ordering graph* describes the required order of linear segments. It contains an edge (S_1, S_2) if and only if the linear segments S_1 and S_2 contain vertices $v_1 \in S_1$ and $v_2 \in S_2$ which are located in the same layer L_i and are ordered subsequently, thus their ranks satisfy $r(v_2) = r(v_1) + 1$. Sander's algorithm sets the vertical position of all vertices by performing a topological sort on the segment ordering graph G_S, which is possible because G_S is acyclic, and then finding the topmost position of each linear segment. Afterwards a *pendulum* method is applied to balance the drawing by moving vertices according to the positions of their neighbors [San96a, Spö09]. The exact port positions must be taken into account here to achieve proper vertex placement.



(a) A layered graph with its linear segments

(b) The segment ordering graph with a possible numbering and topmost alignment

Figure 6.16. Linear segments and their ordering graph

Edge routing

In order to achieve rectilinear edge routing, each edge that cannot be represented by a single horizontal line needs a vertical line segment (see Figure 6.17). A proper order of vertical line segments is important to avoid additional edge crossings. To accomplish this, each edge e connecting vertices from layers L_i and L_{i+1} is assigned a *routing slot* of rank $r(e)$, which is then drawn at the horizontal position $x := x(L_i) + b(L_i) + r(e) \cdot d$, where $x(L_i)$ is the horizontal position at which layer L_i is drawn, $b(L_i)$ is the amount of horizontal space needed by layer L_i, and d is the minimal distance to be left blank between any two line segments. Two bend points are inserted to create the vertical line segment: $(x, y_s(e))$ and $(x, y_t(e))$, where $y_s(e)$ and $y_t(e)$ are the fixed vertical positions of the source and target port of e, respectively. The amount of horizontal space needed for routing slots depends on the maximal assigned rank $r_{i,\max}$, and the position of L_{i+1} can be determined as $x(L_{i+1}) = x(L_i) + b(L_i) + (r_{i,\max} + 1) \cdot d$. The set of vertical positions occupied by an edge e is $Y(e) := [\min\{y_s(e), y_t(e)\}, \max\{y_s(e), y_t(e)\}]$; the basic rule for rank assignment is $r(e) \neq r(e')$ for edges e, e' with $Y(e) \cap Y(e') \neq \varnothing$.

An additional difficulty comes up when the source port of an edge is not on the right side of the source vertex, or the target port is not on the left side of the target vertex. In these cases additional bend points are needed

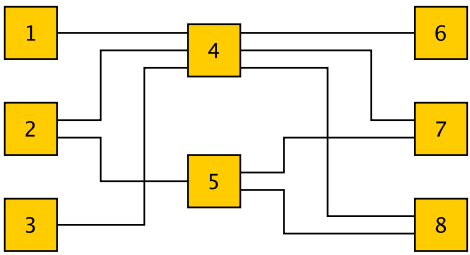


Figure 6.17. Rectilinear edge routing between layers using vertical line segments

to route the edge around the vertex, as seen in Figure 6.11. For this purpose routing slots of different ranks must be assigned on each side of a vertex, similarly to layer-to-layer edge routing. This is done in an additional phase after crossing reduction; all edges which need additional bend points are processed here, as well as self-loops.

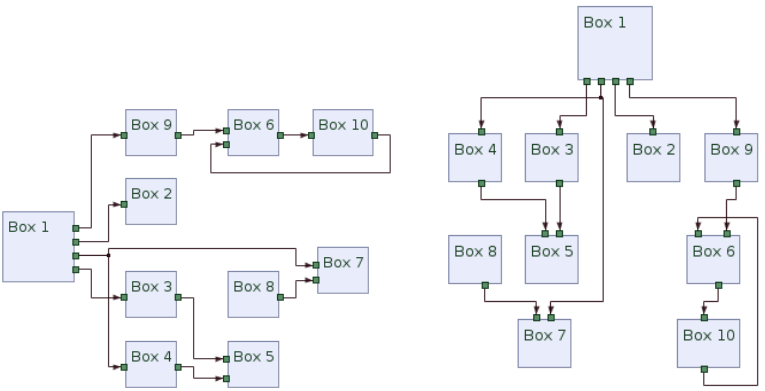
For example, the self-loop $(4, 4)$ in Figure 6.11 is assigned routing slots of rank 1 on the left, bottom and right side of vertex 4, while the edge $(2, 4)$ is assigned a routing slot of rank 2 on the bottom side of vertex 4.

As an output of this additional routing phase, the number of routing slots for the top and the bottom side of each vertex v, together with the given height of v, determines the amount of space that is needed to place v inside its layer. This information is passed to the node placement phase, so that the free space that is left around each vertex suffices for its assigned routing slots.

Experimental Results

Here we will look at some direct outputs of our hierarchical layout algorithm. A more detailed analysis of the algorithm is presented elsewhere [Spö09], and more results are shown in Section 5.4, where the algorithm is embedded into Ptolemy.

Figure 6.18 shows how the overall layout direction changes the output of automatic layout. In Figure 6.19 we see a hand-made layout from an official demonstration of SCADE, and the outcome of automatic layout for



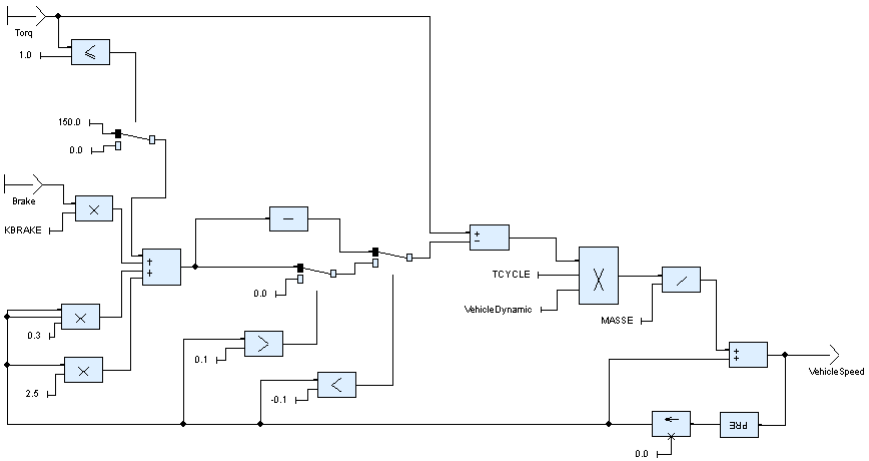
(a) Horizontal layout **(b)** Vertical layout

Figure 6.18. Output of hierarchical layout with different layout options

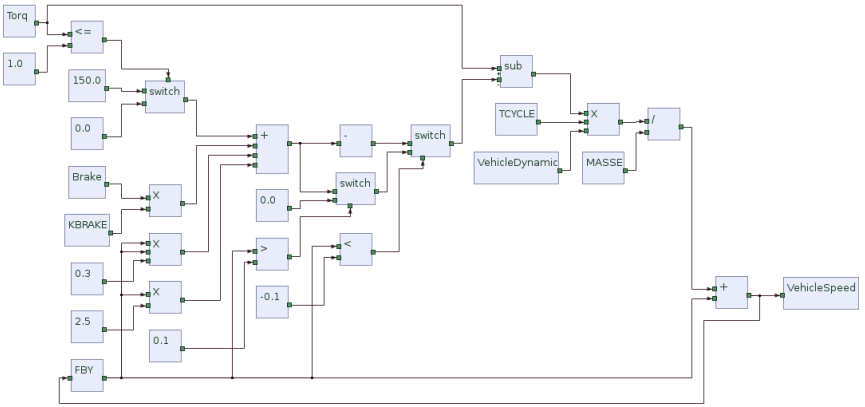
the same diagram, which models calculation of the speed of a vehicle for the environment simulation of a cruise control system. This example shows that the quality of the automatic layout is at least comparable with the carefully prepared manual layout. Layout of a compound diagram with connections to external ports is shown in Figure 6.20. Here we see that our algorithm is able to handle edge routing to external ports, even if they are located on the top or bottom side of the parent node.

Measurement data for the execution time of the hierarchical layout method is shown in Figure 6.21. Execution times were determined on an Intel Xeon 3 GHz processor for different randomly generated graphs. Each value was calculated as the average of the values for five random graphs of equal size, where for each graph the lowest execution time of five consecutive runs was taken.

Figure 6.21a presents measurements for generated graphs $G = (V, E)$ with varying $|V|$ and $|E| = |V|$ in logarithmic scale. The curve is roughly linear with an approximate slope of 1.16, hence, the overall runtime behavior



(a) Original SCADE diagram



(b) Hierarchical layout

Figure 6.19. Comparison of hand-made layout with automatic layout

is nearly linear[3] in the number of vertices. For graphs with about 25 000

[3]Real linear runtime behavior would yield a linear curve of slope 1 in logarithmic scale.

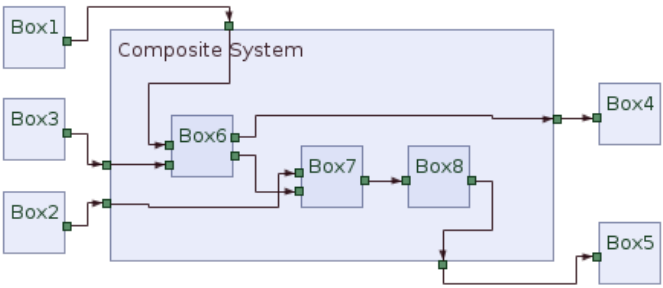
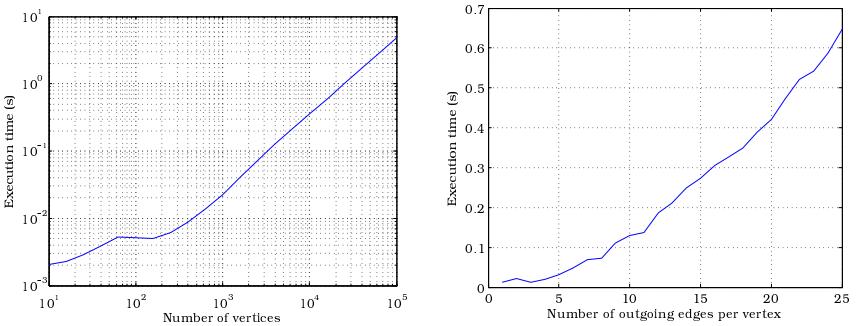


Figure 6.20. Edge routing to external ports



(a) Varying number of vertices, with one outgoing edge per vertex

(b) Varying number of outgoing edges per vertex for 100 vertices

Figure 6.21. Execution times of hierarchical layout

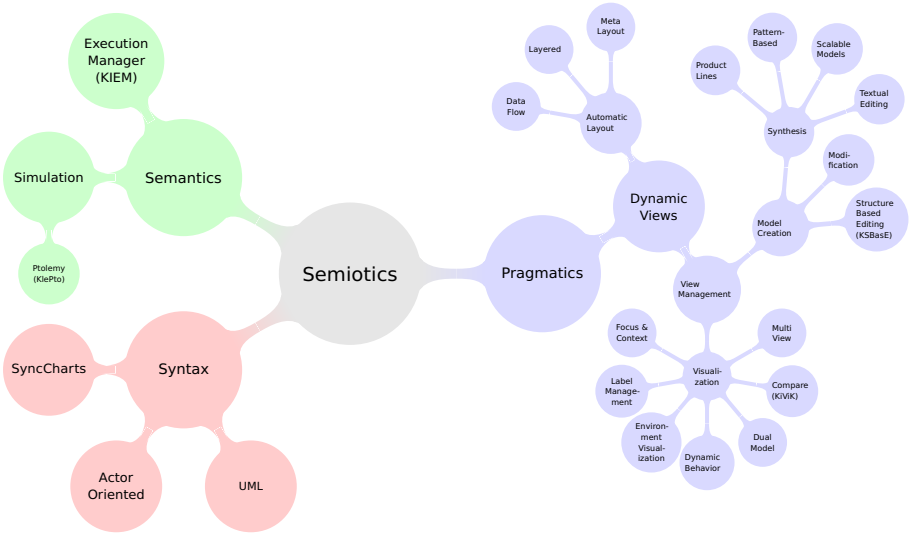
or less vertices the algorithm takes less than a second, which proves its suitability for automatic layout in a user interface environment.

The runtime behavior for generated graphs with a fixed number of 100 vertices and varying number of edges is shown in linear scale in Figure 6.21b. Here we see that the execution time highly depends on the average vertex degree, since layout for a graph with 2 000 vertices and 2 000 edges is 8 times faster than layout for 100 vertices and 2 000 edges. One reason

for this is that for vertices with a lot of incident edges the number of long edges that stretch over multiple layers is likely to be high, so that dummy vertices must be inserted to obtain a proper layering. The consequence is that the problem size rises with regard to the total number of vertices.

Chapter 7

Conclusion and Future Work



In this thesis the linguistic notion of *pragmatics*—orthogonal to questions of syntax and semantics—is extended to all practical user interactions with tools in the process of creating, maintaining or analyzing graphical models. This encompasses many activities and potentials for productivity enhancements. The field of pragmatics gets categorized using the Model-View-Controller paradigm to assist creation of a proper structure for approaches on such enhancements (Chapter 3).

I presented an integrated practical approach to graphical modeling in MDE. Technological stepping stones are provided that build upon each other to raise the abstraction level for user interaction with graphical models.

The key enabler is automatic layout of diagrams that opens a wide toolkit to change the abstraction level in a graphical model dynamically (Section 3.2). In particular, *focus & context* helps to guide the user to the currently "interesting" parts of the model. The concrete configuration of the focus and other effects is done by *view management*, a central infrastructure that should be under control by the user or at least the toolsmith. Therefore it offers convenient abstract interfaces for programming and customization

Several use cases benefit from this basic infrastructure (Section 3.3). Editing and maintaining models using structure-based editing should completely replace any freehand DND editing. Results of an evaluation also back this claim, showing reductions of about 48% for development time of graphical models (Section 4.9).

Model synthesis opens completely new ways to interact with graphical models. Especially textual editing with synchronized diagram and text views offers immediate graphical feedback while the user can freely switch between graphical or textual notation for editing.

Analyzing models becomes much easier by consistently using the diagram itself as feedback for simulations and other analysis such as dual models or comparison (Section 3.4). To that end, view management assists in seeing the important things at the right time, thus allowing a productive visual interactive debugging of models.

Next to the pragmatics-aware modeling concepts I presented the open source Eclipse based implementation project KIELER, the Kiel Integrated Environment for Layout Eclipse Rich Client (Chapter 4). It implements many of the approaches presented before, including the KIELER Infrastructure for Meta Layout, the KIELER View Management, the KIELER Structure-Based Editing, the KIELER Textual Editing, the KIELER Environment Visualization, and as support projects the KIELER Execution Manager and the KIELER Layout of Data Flow Diagrams. For most subprojects the focus is on genericity to be applicable to a wide range of modeling languages.

For demonstration and evaluation a set of graphical model editors has been developed (Chapter 5). The Thin KIELER SyncCharts Editor (ThinK-

Charts) is the major demonstrator. Others were an editor for actor-oriented data flow models and a discussion about some UML languages. In this context I also presented the integration of the KIELER automatic layout into the Ptolemy II graphical Editor Vergil, to validate the approach and APIs beyond the scope of Eclipse.

To summarize, this thesis presents an approach with an implementation towards taming complexity in graphical modeling. It might hopefully inspire other users and developers of graphical model-based design tools to harness the potential of modeling pragmatics. This concerns in particular the productivity gains made possible by replacing manual freehand DND editing with sophisticated automatic layout, and further concepts (structure-based editing, etc.) that build on this.

7.1 Lessons Learned

The KIELER is an experimental platform of the aforementioned concepts. The experience gained from implementing and using KIELER is summarized in the following.

A process for academic software development

Many great research projects in academic contexts are developed only by a hand full of people in the context of one PhD thesis, often discontinued when this single person leaves [JBH+10].

KIELER is aimed to persist as an experimental platform and to be extended in the future. At the current time the KIELER project has 34 contributors from the university with about 1.9 million lines of code (where maybe half of the code is generated) mainly written in the Java programming language[1].

A proper development process is required to cope with complexity in a fluctuating development team mainly consisting of students and a growing code base. The software practice used in the KIELER project is mainly inspired by the one developed at UC Berkeley for the Ptolemy project [RNHL99]. It is documented and continued at the project website and therefore only a summary of the most important aspects is given in the following.

Team Meeting A regular team meeting of all current developers of the KIELER team is held once a week. It is necessary to synchronize the different subprojects, schedule releases and to disseminate new general development strategies or conventions.

Reviews Coding quality in a team of student developers certainly is an issue due to the lack of development experience. However, the work done in the KIELER project is intended to provide such experience by learning from existing solutions and talking about mistakes. *Design* reviews are held in an early development stage of a student's work to

[1] For updated statistics see `http://www.informatik.uni-kiel.de/rtsys/kieler`

guide the major design decisions with the help of class structures and general interaction and user interface approaches. *Code* reviews are performed in later phases to look at some selected source code and to enforce coding conventions and good commenting and documentation in general. Usually a review takes about two hours and is attended by the author, a supervisor and up to two students as reviewers. One intention is to help the author with his or her concrete design or coding tasks, the other is that the other reviewers also learn about good or bad designs or code by reading and discussing them. Reviews emerged to be a key factor to increase quality and to make the difference between a subproject that shows a nice demo and one that somebody else can put to practice.

Project Management Tools Documentation and bug tracking are important tasks for which such a project should have tool support. It should be easy to access (so preferably web-based) and light-weight such that new developers can easily adopt the processes. In KIELER currently the *trac* system[2] is employed that offers a wiki, a simple ticketing system and tracking of source code changes. The wiki is heavily used to document the subprojects (next to the student's theses themselves) and to hold protocols of reviews and other meetings.

Nightly Build Integration of all subprojects is a difficult task. Without a dedicated person as integration manager—a typical situation in an academic context—a nightly build is essential to validate the interoperability between the different subprojects and also for the different platforms (currently KIELER targets Windows, Linux, Mac and Solaris). The nightly build can check the consistency of the essential interdependencies between the different modules (Eclipse plug-ins and features), generate documentation (Javadoc API) and finally create the executable bundles for testing. Currently the Hudson framework is employed as build framework[3], dedicated to Java projects. Unfortunately, the build framework landscape within Eclipse is not yet very clear.

[2] http://trac.edgewall.org/

[3] http://hudson-ci.org

Conventions For KIELER many coding conventions and guidelines have been selected to define a common style and to ease collaboration. The rules are formally specified as rules in static checking tools like Checkstyle[4] to integrate it also into the daily development work for the students. This helps to learn the rules on-the-fly and avoids to overwhelm students with a big set of requirements.

Analysis Code metrics can be used to show potential problems in a subproject. However, so far, in KIELER they have been little used. This could be extended in the future to assist design and code reviews in order to look at the most "interesting" parts first, just like *focus & context*. Currently the code is only rated according to reviews. A simple color rating—red, yellow, green—should indicate the maturity of the code. A Java class can be tagged with the agreed rating and an overview of ratings is created by the nightly build[5]

It took a couple of years now to create and establish the process and some students complain about the overhead it implies. However, the experience shows that it is absolutely necessary and worth investing the efforts of defining and implementing (e. g., nightly builds) it, even or maybe especially in this academic context. When working as graduated students in companies, many former KIELER students report how well this process has actually prepared them to do commercial software development. Still, the process must be light-weight and easy to adopt (e. g., learning by doing) in order not to overburden the students.

Licenses Learned

Eclipse is a platform that enables collaboration between many parties. Whenever people work together, rules are needed to control the collaboration. Whenever such rules are violated in society such that business might be endangered, legal practitioners come into play. Although experience is that computer scientists and developers do not like to think about these

[4] `http://checkstyle.sourceforge.net`

[5] `http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Help/SoftwarePractice/Reviews`

issues, at least they want to protect themselves against law suits. Therefore any released software usually ships under the terms of a specific software license agreement.

Licenses are a complicated topic and choosing one can have great implications on who will collaborate with a project in the future. Especially in the academic context there is an area of conflict between copyright and copyleft. Students and the public demand the latter while companies will usually not accept such licensing.

In the KIELER project currently the Eclipse Public License (EPL)[6] is used. The main reasons are the following.

- ▻ It is an open source license.
- ▻ It contains no copyleft statement, i. e., no clause that requires users of the software to make their software also open source.
- ▻ It is the prevailing license in the Eclipse community.

Especially the last reason eases to integrate a project into the Eclipse context. However, the license might be incompatible with other licenses (e. g., copyleft licenses like the General Public License (GPL)[7]) and therefore collaboration with projects outside the scope of Eclipse might be problematic.

The message here is not that the EPL is the best license available. The point is simply that projects should not underestimate the choice of the right license and really think about consequences and not just pick one license rather randomly. Changing a license later will induce great problems [JBH+10].

Benefits and costs of a rich platform

As base for the KIELER framework the Eclipse platform was carefully chosen (see also Section 4.1). First, only the benefits were evident, now some more experiences are gained, including some drawbacks. One has to trade the

[6] http://www.eclipse.org/legal/
[7] http://www.gnu.org/licenses/

benefits off against the drawbacks to find the right answer for one's own project.

The drawbacks are mainly the following.

Steep learning curve A big platform like Eclipse is a complex piece of software, or to put it bluntly, a big tangled mess of puzzle pieces. Sometimes it is well documented by help pages and tutorials, sometimes not. Sometimes the source code is well commented, but often it is not. Learning to program for Eclipse takes quite a while and many beginners get frustrated. E. g., even after about 5 years of Eclipse experience I would not consider myself an Eclipse expert. However, the system is rather diverse, many bundles in Eclipse have different qualities of documentation and community support. Hence, one should not judge about Eclipse as a whole but about specific subprojects and choose the ones to work with carefully. For example the Xtext community offers great support and the documentation is very good, while the initial source code is almost not commented at all. Debugging sessions can be very effort prone. Still, even associated projects can show absolutely different qualities. The EMF currently has great support while for GMF both detailed documentation as well as community support are disappointing.

Integration into existing code While the Eclipse modularization system promises to explicitly and unambiguously specify the interfaces between components, this is not always valid. Integrating the own project into existing other projects is an elaborate task and especially in the academic context people are often not used to read and debug other people's code.

One example can be meta layout and view management (see Section 4.2 and Section 4.3). Parts of meta layout had to be implanted into existing graphical modeling frameworks like GMF, which took some years, is not yet perfectly solved and is extremely difficult for beginners like new students. In contrast, view management is a completely new concept, where the core could be designed from scratch, which was both fun and a great learning opportunity. Having such a framework fully under ones own control gives a lot of freedom for further development.

Consistency of Bundles A problem of any big software system is to keep the

interoperability between single software components. When a developer implemented some functionality that uses other modules, this piece can break when the other modules change. Eclipse has explicit mechanisms to specify bundle dependencies and especially version constraints on single bundle granularity. This provides means to explicitly specify which version ranges of bundles work together and under which conditions the integration might break. However, these constraints have to be formulated manually. Therefore a developer would have to track when a certain feature of the other bundle changed, such that earlier versions will break the intended usage. This would be extremely effort prone or not feasible at all. Therefore the constraints are usually only estimated. Hence, problems can arise when users use plug-ins in Eclipse versions which are different from the developer's one. This is a serious issue and one major problem in Eclipse, where developers long for better assistance.

However, the benefits of using such a platform heavily outweigh these drawbacks, at least for larger projects. The major advantages are already discussed in Section 4.1. The fact that the Eclipse community offers so many other projects with topics orthogonal to ones own project emerges enormous synergies. One could hardly count all features of Eclipse and other projects that KIELER exploits in order to handle standard issues like files, preferences, window management and basic user interface tasks, not to mention the modeling projects like EMF and GMF that it builds on.

Still, project managers should be aware of the extra efforts that are required and should plan extra resources for them. Having a dedicated (long-term) integration manager with Eclipse know-how would certainly be a good investment.

Cumbersome abstraction of modeling tools

One experience of working inside existing modeling frameworks is that they were usually not designed to customize the pragmatics. They often have abstractions like the separation by MVC paradigm and offer explicit interfaces to be extended. For modeling this is usually only the syntax, i. e.,

the model part. Thus especially the DSL tools like Eclipse allow to exchange abstract and concrete syntax to generate tools for one's custom DSL. Usually the controller part is only partly defined, not specifying semantics for the DSL at all. For Ptolemy it is the other way round: the syntax is fixed and the semantics can be customized by selecting different directors or implementing new ones.

How the tools handle the view, however, usually is fixed and hidden in the implementation of the framework. Often the tool is explicitly designed to interact in small manual editing operations just changing a current state of a diagram a little bit. They are not designed to allow programmatic changes of the view and especially not to synthesize a complete diagram from scratch. Details of the rendering of the view are hidden, thus reading properties like current coordinates of low level items like ports can be difficult.

Therefore it was a good choice to implement such an interface to programmatically interact with views in KIML once and make this available in a more abstract and easy-to-use fashion in the view management framework.

Still, when designing a new framework, developers should always be aware that others might want to use it programmatically in the future, even user interface projects, which designers might not always think about somebody remote controlling them. Therefore a clean API should be created for any project that does not prevent this.

Abstraction vs. Customization

One approach in software engineering is the generative paradigm that is also employed by editor generator frameworks like GMF for diagrams and Xtext for textual DSLs. The idea is to specify the editor in abstract models and generate the implementation fully automatically. If the generation requires more details, then add another model that provides those and combine both descriptions into a more detailed generator model.

However, the experience of example editors in KIELER shows that there is always a need for customization left, because the frameworks never provide generated code that fully complies with the given requirements for a specific project. Some examples are given for the ThinKCharts editor in Section 5.1,

where rather fundamental additions to the original generated code had to be made to make the editor behave exactly as it should without breaking the generative base idea.

In Xtext this is solved a little bit more flexibly. There exist explicit hooks where one can add Java code that customizes certain behavior such as formatting of text or the linking of referenced objects. This helps to customize at the cost of breaking the abstractions.

Graphiti tries to address this by replacing the generative approach completely by a customizable API. This certainly will require more implementation efforts to come to a first running example, but it might pay off in the long term when it comes to tweak the last 20% of the tool. Therefore in my opinion the costs are justifiable if it really helps in the long term, which still has to be evaluated. Hence, it appears to be beneficial to try the Graphiti approach for a complex editor like ThinKCharts the next time. However, a good mixture of abstract specification and generation of common functionality combined with well designed customization capabilities would be optimal.

Diverse automatic layout problem

The experience with using automatic layout in KIELER shows that different graphical syntaxes demand very different handling to get appealing diagrams. The case where diagrams can be mapped directly to nested graphs that for example the GraphViz layout library can handle (see Section 6.2) is rather rare. Most languages have specific properties that require special handling in the placement or routing.

Therefore it is useful to have full control over the employed layout algorithms themselves. Hence, it would not be sufficient to use e. g., the GraphViz library alone. It would be better to be able to customize certain aspects, either by setting right properties or even by customizing the implementation. For a third-party tool like GraphViz this is only partly possible. Therefore recent projects in KIELER try to implement similar algorithms natively in the Java/Eclipse environment. They are designed in an extensible fashion in order to replace or extend certain aspects of the algorithms. This way the algorithms can be more easily customized to the actual diagram

languages instead of doing it the other way round, i. e., being limited by the existing layout capabilities in the design of the used diagram syntax.

Difficult proper presentation of pragmatic issues

Talking or writing about pragmatics is rather difficult, because it addresses processes of user interaction with tools. It involves diagrams that dynamically change over time. Hence, it is rather natural to present problems or possible solutions by doing a demo of respective tools. However, if one has to write a static piece of paper that can only contain static images, i. e., static views of models, it is challenging to make a clear point. Instead, a user has to *experience* what the problems are and the effects of the KIELER approaches.

Therefore the reader is encouraged to download the KIELER and to play or actually work with it or watch some of the demo videos provided on the project website.

Common feedback/criticisms

Here are some initial reactions on presentations of this pragmatics-aware modeling.

KIELER is automatic layout One first misunderstanding is that this work is not only about employing automatic layout. Most feedback is about this issue. People comment about the automatic layout and only rarely about the features that build on top of it (Chapter 3). The reason might be that consequently using automatic layout itself is such a paradigm shift that users have to get used to it first. Only then they might realize the additional benefits of the other building blocks. The requirements to a simple to use view management, the problems of diagram and text synchronization, where to apply focus & context, or how multiple different views of the same model can help, are often less discussed than the (often strong) personal feelings about automatic layout. However, these additional stepping stones have productivity enhancing potentials much beyond pure automatic layout. Therefore one should consider the whole story before deciding in favor or against automatic layout.

"I want full control" "And people like having feedback and control," as Gurr states [Gur99] (Section 1.2). There is the fear to loose control when handing the layout problem to the machine. Of course, there is some truth in this—when using a compiler, people cannot fully control how the assembler is written anymore. However, in practice, experience shows that one often is satisfied with *any* readable layout and thus does not invest efforts in making the layout sound with a freehand DND editor. Therefore in practice the issue of full control may be less relevant than it may seem at first. Still, when propagating automatic layout, one should listen carefully to the potential users and try to extract what it really is that they want control of. Often this is not the individual pixel-by-pixel placement, but something more abstract, that might even be integrated into automatic layout. This may lead to semi-automatic schemes described further below (even if, at the end, users may find it more convenient to use fully automatic layout after all, see "user feedback" below).

Graphical ≠ informal "I'm not a graphical person, I'm a formal person," once was a comment. The question whether diagrams or text/formulas are more formal is not addressed in this work. Graphical vs. textual is a question of syntax, and both diagrams and text can have arbitrarily formal or informal semantics. Therefore, the proposal here is to combine the best of both worlds and not to play text off against diagrams.

Existing layout buttons bad example Some users have already made rather dissatisfying former experiences with automatic layout. Several tools already offer rudimentary layout support and a simple user interface. Layout results are often not appealing and violate basic aesthetic criteria like overlapping. For example Eclipse GMF provides such "arrange all" button that tarnishes the reputation of automatic layout.

Layout algorithms not good enough One claim of this work is that automatic layout must be so good that people are willing to replace manual placing and routing by it (subsection 3.2.1). However, a common opinion—maybe also originating from the last point—is that the layout algorithms today do not meet this requirement. In contrast, we find very sophis-

ticated algorithms in the graph drawing community that provide very appealing results (Section 2.2). Still, they are often not employed in common tools. Therefore this work tries to bridge this gap between MDE and the graph drawing community.

Not aware of productivity loss Many decision makers seem to be unaware of the productivity losses of the usual freehand DND editing and manual static view navigation for diagrams (Section 1.2). Especially business people and academics are often unaware of the overhead. Practitioners of graphical modeling realize the benefits more immediately. Numbers from industry partners indicate that about 30% overhead is induced by manual layout. This is also backed by an evaluation of KSBasE that demonstrated about 50% less time for model creation (Section 4.9). However, current trends towards textual DSL modeling might indicate that some people already see drawbacks with the traditional graphical modeling. Still, the consequences should not be to replace diagrams by text but to enhance the pragmatics of diagrams.

Loose the mental map A spontaneous fear expressed for automatic layout is about loosing the mental map that one may already have of a diagram. This implies that one starts with a given layout and then applies automatic layout that is expected to radically change the placement. However, for rather stable layouters (subsection 3.2.1) and employing them consistently from the beginning, the mental map can be kept very well. Still for radical model changes the value of preserving the mental map is, in my personal opinion often overrated. Experience shows that a clean and reproducible auto layout results more comprehensible models than a very effort prone manual incremental layout that tries to change as little as possible. Especially when working with different people, maybe in different roles, adherence to a consistent layout style may be more important than preserving the mental map of an individual developer throughout the design process.

Semi-automatic layout Related to the aforementioned doubts about whether automatic layout can be "good enough", it is a common request whether one could combine automatic layout with manual "fine tuning". While

it is in general certainly possible to rearrange diagrams manually after pressing the automatic layout button, it is much harder to preserve such manual tweaks when performing the next automatic layout—especially when automatically combined in view management, e. g., for structure-based editing. In fact, none of the layouters we are aware of provides a general solution to this. However, what is possible is to let the user provide individual "layout hints", if they fit into the strategy for automatic layout. E. g., when using a layer-based layouter, one may let the user manually assign hierarchy layers to individual nodes, and such functionality is already integrated into the KLoDD layouter (subsection 6.2.2). Additionally, applying layout recursively to compound nodes allows to tag a complete subgraph hierarchy level as "manually layouted" and therefore keep any custom layout in that particular region. However, this is not yet possible for arbitrary subsets of nodes.

The user feedback after working with KIELER is rather positive.

Get used to it—don't want to miss it anymore For example users employing structure-based editing in practice appreciate the benefits very much [Ble10]. Even if it was unfamiliar to work only with auto layout in the beginning, users get used to it. Finally they find it hard to go back to other tools employing manual layout again.

Interactive layout overrated First, some users request ways to interactively influence the layout, either by specific means to configure the layout algorithm or by simply tagging regions as "manually layouted, don't touch!" Therefore many configuration options have been added to the KIELER Infrastructure for Meta Layout (KIML) (Section 4.2) including an interactive mode for some layouters and tagging regions as manual layout. While this seems to ease initial acceptance, in the long term often people get used to full automatic layout, such that these interactive features are only rarely used any longer.

User interface must be simple Users usually have no sense about the layout approach of a specific layout algorithm or requirements of the tool. Therefore the user interface to call layout must be simple and intuitive.

Otherwise people tend to not use it at all. One example is the first approach to the routing problem in the Ptolemy II editor Vergil (Section 5.4). It introduced five buttons, all changing the diagram massively in different ways while users without any background usually did not understand what the differences were. Therefore the functionality was barely adopted and required a different approach allowing a cleaner interface with only one button. While configurability is very good and important, this tells us that user interfaces have to be very clear and simple. They should always provide meaningful default configurations that lead to good results if the user is not willing to spend any efforts in understanding all customization options.

7.2 How to Adapt this in Practice?

If you read this dissertation and find the pragmatics-aware modeling approach presented here appealing but are not sure (1) whether you could employ the presented approaches in your own project, and (2) how you would start doing so, then this section should give some advice.

The first question is usually quite simple to answer. The approaches around view management and dynamic synthesis of views are so flexible that they should be applicable to most use cases. If one is working with any diagram syntax, then it may be worth looking into KIELER.

Considering the second question, a technical issue is whether one is already working in Eclipse or not. Doing so would simplify some technical aspects. However, if not, some concepts can also be used outside Eclipse as the Ptolemy example in Section 5.4 illustrates.

However, let us assume a freedom of choice and that one can work in the Eclipse platform and thus directly make use of the plug-ins provided by KIELER.

What kinds of pragmatics enhancing use cases are there? First one has to identify the use cases where you see potentials for enhancing pragmatics in your project. You should examine the design process for your own tool. What exact user interactions are required for which steps in the development process? Recapitulate the problem description in Section 1.2. Can you identify similar problems in your process? Also look at the use cases presented in Chapter 3 and see whether they fit to this problem or if one can come up with similar ones. View management is intended to be so flexible that you can implement your own use cases rather easily.

Which graphical editor can one use? The next question is from where to get the graphical view technically. Let us look at some different scenarios.

Existing non-GEF Editor There already is an Eclipse editor, but the framework is not GMF and not even GEF. Then one can create custom glue-code that interfaces between the KIELER Infrastructure for Meta Layout (KIML) and the existing editor, see Section 4.2.

Existing GEF Editor If the existing editor is based on GEF but was not generated with GMF, one will also have to customize the glue-code to KIML, but is is possible to reuse certain functionality of the existing glue-code, like reading the graph structure from GEF edit parts.

Existing GMF Editor En existing GMF editor is ready to go as the glue-code for GMF is already provided. One might want to configure default settings for layouters using the KIML extension points.

Special Syntax but no Editor If a special diagram syntax is used that requires a special editor, one can implement a custom editor, using GMF (or Graphiti in the future). Examples are given in Chapter 5.

Standard Syntax without Editor If one is not sure about a special syntax, one can try to reuse existing editors. In the Eclipse community, a wide variety of existing GMF editors does exist. One of the editors presented in Chapter 5 can be taken, such as ThinKCharts for state machines or KAOM for actor-oriented dataflow models. Alternatively, the project website might indicate that there are other new editors, like the simple KIELER Editor for Graphs (KEG). Other examples are offered by the UML, where for example the Papyrus project provides compatible diagram editors, see Section 5.3. Then model transformations can be employed to create new models for that editor. This can be done either to create views from scratch or to transform an existing model that does not have a corresponding graphical editor into a graphical view of an already available one.

What inputs to View Management? View management is intended to automatically show the user the "interesting" parts of the models for the task he or she is currently performing. Therefore this context, which view management requires, needs to be provided programmatically. Hence, one has to identify what information view management needs to formulate the rules in a view management combination. Some of these *triggers* (see Section 4.3) might be already available, such as reacting on user selection or a button press. Others might be very use case specific. Maybe one wants to visualize something from a model simulation or results of an analysis.

Then it would be necessary to implement a custom trigger following the interfaces given by KiVi and use them in the combination.

How to bring this together? The last and most important step is to combine all information by explicitly implementing the custom use case in a KiVi combination. Reacting on triggers, a combination schedules *effects*. If the available effects (see Section 4.3) are too limited, one can provide custom ones and thus literally do anything. However, the most important effects might involve the synthesis of new views or interact with existing ones like highlighting or layouting them, where one can benefit of the functionality of available effects.

7.3 Future Work

While many of the concepts of Chapter 3 could be implemented, only a rudimentary experimental evaluation has been made. There is much potential for more challenging evaluations of the approaches that I believe will further back the claims made in this thesis.

Furthermore, not all concepts presented could be implemented in the scope of this thesis. For example *label management* appears to have great potentials, as well as the other use cases that build upon the basic infrastructure, such as *scalable models*, *pattern-based modeling* and *product lines*. This thesis mainly focuses on using one diagram to present focus & context views in order to reduce model complexity in a given context. This could be extended by using multiple, even different diagrams next to each other, where for example one could be used as a navigational overview in which the user can select the focus, while another diagram shows a different view of this focus. This would be some sort of *multi-view modeling* with potentially different but associated models and/or views.

The core, view management, was carried to a useful abstraction level, where developers, mainly toolsmiths can control it. In a next step, the abstraction level should be even more increased to allow the tool user to control it. Thus, the definition of what is "interesting" in a model does not need to be pre-defined but could be customized by the users themselves.

The primary demonstrator was the ThinKCharts editor, the Thin Kieler SyncCharts tool. It is a GMF editor mainly basing on compound graphs. The implementation of KIELER should be extended to natively cover a wider variety of languages, beyond the scope of GMF—maybe starting with Graphiti—, and with further language notations than used in Statecharts. Especially actor-oriented data flow models promise great benefits, as the problems are the same and the approaches seem to be applicable, but full solutions may be challenging.

Bibliography

[AK03] Colin Atkinson and Thomas Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, pages 36–41, 2003.

[AKSM85] Anjali Arya, Anshul Kumar, V. V. Swaminathan, and Amit Misra. Automatic generation of digital system schematic diagrams. In *DAC '85: Proceedings of the 22nd ACM/IEEE Conference on Design Automation*, pages 388–395. ACM, 1985. doi:http://doi.acm.org/10.1145/317825.317917.

[ALABM+01] Ákos Lédeczi, Árpád Bakay, Miklós Maróti, Péter Vlgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, 2001. doi:http://doi.ieeecomputersociety.org/10.1109/2.963443.

[And96] 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.

[And03] Charles André. Semantics of SyncCharts. Technical Report ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.

[Bar02] Roswitha Bardohl. GenGEd – a visual environment for visual languages. *Science of Computer Programming, Special Issue of GraTra '00*, 2002.

[Bay09] 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.

[BC84] Gérard Berry and Laurent Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, volume 197 of *LNCS*, pages 389–448. Springer-Verlag, 1984.

[BCF+08] Christopher Brooks, Chih-Hong Patrick Cheng, Thomas Huining Feng, Edward A. Lee, and Reinhard von Hanxleden. Model engineering using multimodeling. In *Proceedings of the 1st International Workshop on Model Co-Evolution and Consistency Management (MCCM'08), a workshop at MODELS'08*, Toulouse, September 2008.

[Bec09] Nils Beckel. View Management for Visual Modeling. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, October 2009. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nbe-dt.pdf`.

[Bee94] Michael von der Beeck. A comparison of Statecharts variants. In H. Langmaack, W. P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*, pages 128–148. Springer-Verlag, 1994.

[BEK+06] Enrico Biermann, Karsten Ehrig, Christian Köhler, Gnther Kuhns, Gabriele Taentzer, and Eduard Weiss. Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In *Model Driven Engineering Languages and Systems (MoDELS'06), LNCS*, volume 4199/2006, pages 425–439. Springer Berlin/Heidelberg, 2006.

[Ble10] Joachim Bleidiessel. A domain specific language for railway control. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2010.

[BLS+09] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. An example is worth a thousand words: Composite operation modeling by-example.

In *Model Driven Engineering Languages and Systems, (MoDELS'09)*, volume 5795 of *LNCS*. Springer Berlin / Heidelberg, 2009. Available from: http://www.springerlink.de/content/y47168q478223177/, doi:10.1007/978-3-642-04425-0_20.

[Bra01] Jürgen Branke. Dynamic graph drawing. In M. Kaufmann and Dorothea Wagner, editors, *Drawing Graphs: Methods and Models*, volume 2025 of *LNCS*. Springer Verlag, 2001.

[Bru08] Cédric Brun. Comparing and Merging Models with Eclipse: an Update on EMF Compare. In *EclipseCon 2008*, Santa Clara, California, March 2008.

[BW97] Ulrik Brandes and Dorothea Wagner. A bayesian paradigm for dynamic graph layout. In *GD '97: Proceedings of the 5th International Symposium on Graph Drawing*, pages 236–247, London, UK, 1997. Springer-Verlag.

[BWGP01] Alan F. Blackwell, Kirsten N. Whitley, Judith Good, and Marian Petre. Cognitive factors in programming with diagrams. *Artificial Intelligence Review*, 15(1/2):95–114, 2001.

[CCG+08] Benoit Combemale, Xavier Cregut, Jean-Patrice Giacometti, Pierre Michel, and Marc Pantel. Introducing simulation and model animation in the mde topcased toolkit. In *Proceedings of the 4th European Congress EMBEDDED REAL TIME SOFTWARE (ERTS '08)*, 2008.

[CG07] Markus Chimani and Carsten Gutwenger. Algorithms for the hypergraph and the minor crossing number problems. In *18th International Symposium on Algorithms and Computation (ISAAC'07)*, volume 4835 of *LNCS*, pages 184–195. Springer, 2007.

[CGM97] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference*

on Management of data, pages 26–37, New York, NY, USA, 1997. ACM. doi:http://doi.acm.org/10.1145/253260.253266.

[CMS99] Stuart K. Card, Jock Mackinlay, and Ben Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, January 1999.

[CMT02] Rodolfo Castelló, Rym Mili, and Ioannis G. Tollis. A framework for the static and interactive visualization for statecharts. *Journal of Graph Algorithms and Applications*, 6(3):313–351, 2002.

[CON85] Norishige Chiba, Kazunori Onoguchi, and Takao Nishizeki. Drawing plane graphs nicely. *Acta Informatica*, 22(2):187–201, 1985. doi:10.1007/BF00264230.

[CP96] Micheal K. Coleman and D. Stott Parker. Aesthetics-based graph layout for human consumption. *Software – Practice and Experience*, 26(12):1415–1438, December 1996.

[CPPSV06] Luca Carloni, Roberto Passerone, Alessandro Pinto, and Alberto Sangiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations and Trends in Design Automation*, 1(1):1–204, 2006. Available from: http://chess.eecs.berkeley.edu/pubs/98.html.

[CSN07] Kai Chen, Janos Sztipanovits, and Sandeep Neema. Compositional specification of behavioral semantics. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'07)*, pages 906–911, San Jose, CA, USA, 2007.

[CTM03] Stuart M. Charters, Nigel Thomas, and Malcolm Munro. The end of the line for software visualisation. In *IEEE Second Workshop on Visualizing Software for Analysis and Understanding (VISSOFT'03)*, Amsterdam, The Netherlands, September 2003.

[DETT94] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: An annotated bibliography. *Computational Geometry: Theory and Applications*, 4:235–282, June 1994.

[DETT99] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[Die07] Stephan Diehl. *Software Visualization: Visualizing the Structure, Behavior and Evolution of Software*. Springer, 2007.

[DK82] Alan L. Davis and Robert M. Keller. Data flow program graphs. *Computer*, 15(2):26–41, Feb 1982.

[DKMT07] Ugur Dogrusöz, Konstantinos G. Kakoulis, Brendan Madden, and Ioannis G. Tollis. On labeling in graph visualization. *Inf. Sci.*, 177(12):2459–2472, 2007.

[Dou03] Bruce Powel Douglass. *Real-time Design Patterns: Robust Scalable Architecture for Real-time Systems*. Addison-Wesley, 2003.

[DOW08] Gero Decker, Hagen Overdick, and Mathias Weske. Oryx - sharing conceptual models on the web. Demo Session of the 27th International Conference on Conceptual Modeling (ER), Barcelona, Spain, 2008.

[Ead84] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[EFK00] Markus Eiglsperger, Ulrich Fößmeier, and Michael Kaufmann. Orthogonal graph drawing with constraints. In *SODA '00: Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 3–11. SIAM, 2000.

[EGB06] Thomas Eschbach, Wolfgang Guenther, and Bernd Becker. Orthogonal hypergraph drawing for improved visibility. *Journal of Graph Algorithms and Applications*, 10(2):141–157, 2006.

[Eic05] Holger Eichelberger. *Aesthetics and Automatic Layout of UML Class Diagrams*. PhD thesis, Bayerische Julius-Maximilians-Universität Würzburg, 2005.

[Eig03] Markus Eiglsperger. *Automatic Layout of UML Class Diagrams: A Topology-Shape-Metrics Approach*. PhD thesis, Faculty of Information and Cognitive Science, Eberhard-Karls-Universität Tübingen, 2003.

[EJL+03] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003. doi:10.1109/JPROC.2002.805829.

[ELMS91] P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the mental map of a diagram. In *Computergraphics*, 99, pages 24–31, 1991.

[Esc08] Thomas Eschbach. *Visualisierungen im Schaltkreisentwurf*. PhD thesis, Institut für Informatik, Albert-Ludwigs-Universität Freiburg, June 2008.

[ESS06] Hajo Eichler, Markus Scheidgen, and Michael Soden. A metamodelling framework for modelling semantics in the contect of existing domain platforms. Technical report, Department of Computer Science, Humboldt-Universität zu Berlin, 2006.

[Est08] J.A. Estefan. Survey of model-based systems engineering (MBSE) methodologies, Rev. B. Technical report, INCOSE MBSE Focus Group, May 2008.

[EV06] Sven Efftinge and Markus Voelter. oAW xText: A framework for textual DSLs. In *Eclipse Summit Europe*, Esslingen, Germany, October 2006.

[Fav04] Jean-Marie Favre. Towards a basic theory to model model driven engineering. In *In Workshop on Software Model En-*

gineering, WISME 2004, joint event with UML2004, Lisboa, Portugal, October 2004.

[Fav05] Jean-Marie Favre. Foundations of model (driven) (reverse) engineering : Models – Episode I: Stories of the fidus papyrus and of the solarus. In *Post-Proceedings of Dagstuhl Seminar on Model Driven Reverse Engineering*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2005. Available from: http://drops.dagstuhl.de/opus/volltexte/2005/13/.

[FKRvH05] Hauke Fuhrmann, Jens Koch, Jörn Rennhack, and Reinhard von Hanxleden. The aerospace demonstrator of DECOS. In *Proceedings of the 8th International IEEE Conference on Intelligent Transportation Systems (ITSC'05)*, pages 19–24, Vienna, Austria, September 2005.

[FKRvH06] Hauke Fuhrmann, Jens Koch, Jörn Rennhack, and Reinhard von Hanxleden. Model-based system design of time-triggered architectures—an avionics case study. In *25th Digital Avionics Systems Conference (DASC'06)*, Portland, OR, USA, October 2006.

[FL08] Thomas Huining Feng and Edward A. Lee. Scalable models using model transformation. In *1st International Workshop on Model Based Architecting and Construction of Embedded Systems ($ACES^MB$)*, September 2008. Available from: http://chess.eecs.berkeley.edu/pubs/487.html.

[FPP90] H. De Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990. doi:10.1007/BF02122694.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software—Practice & Experience*, 21(11):1129–1164, 1991. doi:http://dx.doi.org/10.1002/spe.4380211102.

[FSKdR05] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem P. de Roever. 29 new unclarities in the semantics of UML 2.0 State Machines. In *ICFEM*, volume 3785 of *LNCS*, pages 52–65. Springer, 2005.

[FSMvH10] 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.

[FvH09a] Hauke Fuhrmann and Reinhard von Hanxleden. Enhancing graphical model-based system design—an avionics case study. In *Conjoint workshop of the European Research Consortium for Informatics and Mathematics (ERCIM) and Dependable Embedded Components and Systems (DECOS) at SAFECOMP'09*, Hamburg, Germany, September 2009.

[FvH09b] Hauke Fuhrmann and Reinhard von Hanxleden. Enhancing graphical model-based system design—an avionics case study. Technical Report 0901, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, January 2009.

[FvH09c] Hauke Fuhrmann and Reinhard von Hanxleden. On the pragmatics of model-based design. Technical Report 0913, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2009.

[FvH10a] Hauke Fuhrmann and Reinhard von Hanxleden. On the pragmatics of model-based design. In *Foundations of Computer Software. Future Trends and Techniques for Development—15th Monterey Workshop 2008, Budapest, Hungary, September 24–26, 2008, Revised Selected Papers*, volume 6028 of *LNCS*, 2010. doi:10.1007/978-3-642-12566-9_7.

[FvH10b] Hauke Fuhrmann and Reinhard von Hanxleden. Taming graphical modeling. Technical Report 1003, Christian-

Albrechts-Universität zu Kiel, Department of Computer Science, May 2010.

[FvH10c] Hauke Fuhrmann and Reinhard von Hanxleden. Taming graphical modeling. In *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, LNCS, Oslo, Norway, October 2010. Springer.

[Gan04] Emden R. Gansner. Drawing graphs with GraphViz. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, November 2004.

[GGB+05] Jan Gulliksen, Bengt Göransson, Inger Boivie, Jenny Persson, Stefan Blomkvist, and Åsa Cajander. Key principles for user-centred systems design. In *Human-Centered Software Engineering—Integrating Usability in the Software Development Lifecycle*, volume 8 of *Human-Computer Interaction Series*, pages 17–36. Springer Netherlands, 2005. Available from: http://www.springerlink.com/content/r340755323560170/, doi:10.1007/1-4020-4113-6_2.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gir06] Martin Girschick. Difference detection and visualization in UML class diagrams. Technical Report TUD-2006-5, Department of Computer Science, TU Darmstadt, August 2006.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co, New York, 1979.

[GJ83] Michael R. Garey and David S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4(3):312–316, 1983. Available from: http://link.aip.org/link/?SML/4/312/1, doi:10.1137/0604033.

[GJK+03] Carsten Gutwenger, Michael Jünger, Karsten Klein, Joachim Kupke, Sebastian Leipert, and Petra Mutzel. A new approach for visualizing UML class diagrams. In *SoftVis '03: Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 179–188, New York, NY, USA, 2003. ACM. doi:http://doi.acm.org/10.1145/774833.774859.

[GKNV93] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230, 1993.

[GLL99] Alain Girault, Bilung Lee, and Edward A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18:742–760, 1999.

[GM98] Carsten Gutwenger and Petra Mutzel. Planar polyline drawings with good angular resolution. In *GD '98: Proceedings of the 6th International Symposium on Graph Drawing*, volume 1547 of *LNCS*, pages 167–182. Springer-Verlag, 1998.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience*, 30(11):1203–1234, 2000.

[GP92] T. R. G. Green and M. Petre. When visual programs are harder to read than textual programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings ECCE-6 (6th European Conference Cognitive Ergonomics)*, 1992.

[GP96] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *J. Visual Languages and Computing*, 7(2):131–174, June 1996.

[Gre89] T. R. G. Green. Cognitive Dimensions of Notations. In *Companion Proceedings of the CHI '98 Conference on Human*

Factors in Computing Systems, pages 443–460, Cambridge, UKDepartment of Computer Science, University of Calgary, 1989. Cambridge University Press.

[Gur99] Corin A. Gurr. Effective diagrammatic communication: Syntactic, semantic and pragmatic issues. *Journal of Visual Languages & Computing*, 10(4):317 – 342, 1999. Available from: http://www.sciencedirect.com/science/article/B6WMM-45FSBDW-J/2/13d54f220d651a773799a381c18f2e64, doi:DOI:10.1006/jvlc.1999.0130.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HFPvH06] Stephan Höhrmann, Hauke Fuhrmann, Steffen Prochnow, and Reinhard von Hanxleden. A versatile demonstrator for distributed real-time systems: Using a model-railway in education. In Amund Skavhaug and Erwin Schoitsch, editors, *Proceedings of the ERCIM/DECOS Dependable Embedded Systems Workshop at Euromicro 2006*, Cavtat/Dubrovnik, Croatia, August 2006.

[HLN+90] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[HM76] J.W. Hunt and M.D. McIlroy. An algorithm for differential file comparison. Technical Report 41, Bell Laboratories, July 1976.

[HM77] Hartmut Haberland and Jacob L. Mey. Editorial: Linguistics and pragmatics. *Journal of Pragmatics*, 1:1–12, 1977.

[HM10] Berthold Hoffmann and Mark Minas. Defining models - meta models versus graph grammars. In *Ninth International Work-*

shop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2010), volume 29 of *Electronic Communications of the EASST*, Berlin, Germany, 2010.

[HR04] David Harel and Bernhard Rumpe. Meaningful modelling: What's the semantics of "semantics"? *IEEE Computer*, 37(10):64–72, 2004.

[HY02] David Harel and Gregory Yashchin. An algorithm for blob hierarchy layout. *The Visual Computer*, 18:164–185, 2002.

[Jac07] Steffen Jacobs. Konzepte zur besseren Visualisierung grafischer Datenflussmodelle. Student resarch project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, February 2007. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sja-st.pdf`.

[Jac08] Steffen Jacobs. Automatisierte Validierung von Modul-Konfigurationen in der Integrierten Modularen Avionik. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, January 2008. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sja-dt.pdf`.

[JBH+10] Johannes Jacop, Marcel Bruch, Stephan Herrmann, Jens von Pilgrim, Mirko Seifert, Claas Wilke, Birgit Demuth, Matthias Hanns, Simon Kronsreder, Stanislav Elinson, and Maximilian Kögel. Eclipse & academia. *Eclipse Magazin*, 6.10, October 2010.

[JBK06] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, pages 249–254, New York, NY, USA, 2006. ACM. `doi:http://doi.acm.org/10.1145/1173706.1173744`.

[JM03] Michael Jünger and Petra Mutzel. *Graph Drawing Software*. Springer, October 2003.

[Kan96] Goos Kant. Drawing planar graphs using the canonical ordering. *Algorithmica*, 16(1):4–32, 1996.

[KASS03] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the use of graph transformation in the formal specification of model interpreters. *Journal of Universal Computer Science*, 9(11):1296–1321, 2003.

[KD10] Lars Kristian Klauske and Christian Dziobek. Improving modeling usability: Automated layout generation for simulink. In *Proceedings of the MathWorks Automotive Conference 2010*, 2010.

[KK89] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[KM02] Oliver Köth and Mark Minas. Structure, Abstraction, and Direct Manipulation in Diagram Editors. In *DIAGRAMS '02: Proceedings of the Second International Conference on Diagrammatic Representation and Inference*, pages 290–304, London, UK, 2002. Springer-Verlag.

[KMS94] Corey Kosak, Joseph Marks, and Stuart Shieber. Automating the layout of network diagrams with specified visual organization. *Transactions on Systems, Man and Cybernetics*, 24(3):440–454, March 1994.

[Kna10] Stephan Knauer. KEV – KIELER Environment Visualization – Beschreibung einer Zuordnung von Simulationsdaten und SVG-Graphiken. Student research project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2010. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/skn-st.pdf`.

[Kra07] Jeff Kramer. Is abstraction the key to computing? *Communications of the ACM*, 50(4):36–42, 2007. `doi:http://doi.acm.org/10.1145/1232743.1232745`.

[KT97] Konstantinos G. Kakoulis and Ioannis G. Tollis. On the edge label placement problem. In *GD '96: Proceedings of the Symposium on Graph Drawing*, pages 241–256, London, UK, 1997. Springer-Verlag.

[KW01] Michael Kaufmann and Dorothea Wagner, editors. *Drawing Graphs: Methods and Models*. Number 2025 in LNCS. Springer-Verlag, Berlin, Germany, 2001.

[KW02] Michael Kaufmann and Roland Wiese. Maintaining the mental map for circular drawings. In *GD '02: Revised Papers from the 10th International Symposium on Graph Drawing*, pages 12–22, London, UK, 2002. Springer-Verlag.

[LA94] Ying K. Leung and Mark D. Apperley. A review and taxonomy of distortion-oriented presentation techniques. *ACM Transactions on Computer-Human Interaction*, 1(2):126–160, June 1994.

[Lag98] C. R. Lageweg. Designing an automatic schematic generator for a netlist description. Technical Report 1-68340-44(1998)03, Laboratory of Computer Architecture and Digital Techniques (CARDIT), Delft University of Technology, Faculty of Information Technology and Systems, 1998.

[Lee03a] Edward A. Lee. Model-driven development - from object-oriented design to actor-oriented design. Extended abstract of an invited presentation at Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation (a.k.a. The Monterey Workshop) Chicago, September 2003. `http://ptolemy.eecs.berkeley.edu/publications/papers/03/MontereyWorkshopLee/`.

[Lee03b] Edward A. Lee. Overview of the Ptolemy project. Technical Memorandum UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA, July 2003.

[Lee09] Edward A. Lee. Finite state machines and modal models in ptolemy ii. Technical Report UCB/EECS-2009-151, EECS Department, University of California, Berkeley, Nov 2009. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-151.html.

[LNW03] 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.

[LR96] John Lamping and Ramana Rao. Visualizing large trees using the hyperbolic browser. In *CHI '96: Conference companion on Human factors in computing systems*, pages 388–389, New York, NY, USA, 1996. ACM Press. http://doi.acm.org/10.1145/257089.257389.

[Luk10] Adriana Lukaschewitz. Transformation von Esterel nach SyncCharts in KIELER. Bachelor thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2010. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/adl-bt.pdf.

[LW99] Y.K. Leung and G. Wilson. WinFold: a folding editor for collaborative writing. *Communications, 1999. APCC/OECC '99. Fifth Asia-Pacific Conference on Communications and Fourth Optoelectronics and Communications Conference*, 2:1073–1078 vol.2, 1999. doi:10.1109/APCC.1999.820449.

[Mar91] Florence Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, October 1991.

[Mat10] 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.

[MELS95] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages & Computing*, 6(2):183–210, June 1995.

[MFvH09] Christian Motika, Hauke Fuhrmann, and Reinhard von Hanxleden. Semantics and execution of domain specific models. Technical Report 0923, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2009.

[MFvH10] Christian Motika, Hauke Fuhrmann, and Reinhard von Hanxleden. Semantics and execution of domain specific models. In *2nd Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe 2010) at conference INFORMATIK 2010*, GI-Edition – Lecture Notes in Informatics (LNI), Leipzig, Germany, September 2010. Bonner Köllen Verlag.

[MGH05] Akhil Mehra, John Grundy, and John Hosking. A generic approach to supporting diagram differencing and merging for collaborative design. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 204–213, New York, NY, USA, 2005. ACM. doi:http://doi.acm.org/10.1145/1101908.1101940.

[Min06] Mark Minas. Generating meta-model-based freehand editors. In Albert Zündorf and Dániel Varró, editors, *Proc. of the 3rd International Workshop on Graph Based Tools (GraBaTs'06), Satellite event of the 3rd International Conference on Graph Transformation*, volume 1 of *Electronic Communications of the EASST*, Natal, Brazil, September 2006. Available from: http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/83.

[MJ03] Benjamin Musial and Timothy Jacobs. Application of focus + context to UML. In *APVis '03: Proceedings of the Asia-Pacific symposium on Information visualisation*, pages 75–80, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[MM08] Sonja Maier and Mark Minas. A static layout algorithm for diameta. In *Proceedings of the Seventh International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008)*, volume 10 of *Electronic Communications of the EASST*, Budapest, Hungary, March 2008.

[MM09] Sonja Maier and Mark Minas. Pattern-based layout specifications for visual language editors. In Paolo Bottoni, Esther Guerra, and Juan de Lara, editors, *Proc. of the 1st International Workshop on Visual Formalisms for Patterns, satellite of VL/HCC'09*, volume 25 of *Electronic Notes of the EASST*, Corvallis, OR, USA, September 2009. Available from: `http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/351`.

[Mor38] Charles William Morris. *Foundations of the theory of signs*, volume 1 of *International encyclopedia of unified science*. The University of Chicago Press, Chicago, 1938.

[Mot07] Christian Motika. Modellbasierte Umgebungssimulation für verteilte Echtzeitsysteme mit flexiblem Schnittstellenkonzept. Student research project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, October 2007. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-st.pdf`.

[Mot09] Christian Motika. Semantics and execution of domain specific models—KlePto and an execution framework. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2009. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-dt.pdf`.

[Mül10] 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`.

[Mye98] Brad A. Myers. A brief history of human-computer interaction technology. *interactions*, 5(2):44–54, 1998. doi:http://doi.acm.org/10.1145/274430.274436.

[Obj04] Object Management Group. MOF 2.0 Query/Views/Transformation RFP, April 2004. http://www.omg.org/docs/ad/02-04-10.pdf.

[Obj05] Object Management Group. Unified Modeling Language: Superstructure, version 2.0, Aug 2005. http://www.omg.org/docs/formal/05-07-04.pdf.

[Obj06] Object Management Group. Meta Object Facility (MOF) Core Specification, v2.0, January 2006. http://www.omg.org/spec/MOF/2.0/PDF/.

[OWK03] Dirk Ohst, Michael Welle, and Udo Kelter. Differences between versions of UML diagrams. *SIGSOFT Softw. Eng. Notes*, 28(5):227–236, 2003. doi:http://doi.acm.org/10.1145/949952.940102.

[PCJ96] Helen C. Purchase, Robert F. Cohen, and Murray James. Validating graph drawing aesthetics. In F. Brandenburg, editor, *Proceedings of Graph Drawing Symposium*, volume 1027 of *LNCS*, pages 435–446. Springer Verlag, 1996.

[Pet95] Marian Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, June 1995.

[Pet08] Anne-Kathrin Peters. Musterbasiertes Layout von Statecharts. Masters thesis, Universität Hamburg, Falkultät für Mathematik, Informatik und Naturwissenschaften, Department Informatik, June 2008.

[PHG06] Helen C. Purchase, Eve E. Hoggan, and Carsten Görg. How important is the "mental map"? — An empirical investigation of a dynamic graph layout algorithm. In Michael Kaufmann and Dorothea Wagner, editors, *Graph Drawing*, volume 4372

of *LNCS*, pages 184–195. Springer, 2006. Available from: http://dblp.uni-trier.de/db/conf/gd/gd2006.html.

[PS91] Amir Pnueli and M. Shalev. What is in a step: On the semantics of Statecharts. In *TACS '91: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 244–264, London, UK, 1991. Springer-Verlag.

[PT90] F.N. Paulisch and W.F. Tichy. EDGE: An extendible graph editor. *Software: Practice and Experience*, 20(S1):S63–S88, 1990.

[PTvH06] Steffen Prochnow, Claus Traulsen, and Reinhard von Hanxleden. Synthesizing Safe State Machines from Esterel. In *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, Ottawa, Canada, June 2006.

[Pur97] Helen C. Purchase. Which aesthetic has the greatest effect on human understanding? In *Proceedings of Graph Drawing Symposium, Di Battista, G. (ed)*, volume 1353 of *LNCS*. Springer Verlag, 1997.

[Pur02] Helen C. Purchase. Metrics for graph drawing aesthetics. *Journal of Visual Languages and Computing*, 13(5):501–516, 2002.

[PvH04] Steffen Prochnow and Reinhard von Hanxleden. Visualisierung komplexer reaktiver Systeme – Annotierte Bibliographie. Technical Report 0406, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Kiel, Germany, June 2004.

[PvH06] Steffen Prochnow and Reinhard von Hanxleden. Comfortable modeling of complex reactive systems. In *Proceedings of Design, Automation and Test in Europe Conference (DATE'06)*, Munich, Germany, March 2006.

[PvH07] Steffen Prochnow and Reinhard von Hanxleden. Statechart development beyond WYSIWYG. In *Proceedings of*

the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07), volume 4735 of *LNCS*, Nashville, TN, USA, October 2007. doi:10.1007/978-3-540-75209-7_43.

[Ree79] Trygve Reenskaug. Models – Views – Controllers. Technical report, Xerox PARC technical note, December 1979.

[RL08] Jean-Michel Boucheix Richard Lowe. Learning from animated diagrams: How are mental models built? In *Diagrammatic Representation and Inference: 5th International Conference, Diagrams 2008*, page 266, Herrsching, Germany, September 2008. Springer-Verlag New York Inc.

[RNHL99] John Reekie, Stephen Neuendorffer, Christopher Hylands, and Edward A. Lee. Software practice in the Ptolemy project. Technical report, Gigascale Semiconductor Research Center, April 1999. http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII8.0/ptII/doc/coding/sftwareprac/software-practice.pdf.

[RR00] Jason Robbins and David Redmiles. Cognitive support, UML adherence, and XMI interchange in Argo/UML. *Journal of Information and Software Technology. Special issue: The Best of COSET '99*, 42(2):79–89, 2000.

[San94] Georg Sander. Graph layout through the VCG tool. Technical Report A03/94, Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken, October 1994.

[San96a] Georg Sander. A fast heuristic for hierarchical Manhattan layout. In *GD '95: Proceedings of the Symposium on Graph Drawing*, volume 1027 of *LNCS*, pages 447–458. Springer-Verlag, 1996.

[San96b] Georg Sander. Layout of compound directed graphs. Technical Report A/03/96, Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken, June 1996.

[San04] Georg Sander. Layout of directed hypergraphs with orthogonal hyperedges. In *GD 2003: Proceedings of the 11th International Symposium on Graph Drawing*, volume 2912 of *LNCS*, pages 381–386. Springer-Verlag, 2004.

[SB92] Manojit Sarkar and Marc H. Brown. Graphical fisheye views of graphs. In *Proceedings of the ACM SIGCHI 1992 Conference on Human Factors in Computing Systems*, pages 83–91, 1992.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF Eclipse Modeling Framework*. Eclipse Series. Addison-Wesley, Pearson Education, 2 edition, 2009.

[Sch90] Walter Schnyder. Embedding planar graphs on the grid. In *SODA '90: Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 138–148. SIAM, 1990.

[Sch08] Arne Schipper. Layout and Visual Comparison of Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2008. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ars-dt.pdf`.

[Sch09] 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`.

[See97] Jochen Seemann. Extending the Sugiyama algorithm for drawing UML class diagrams: Towards automatic layout of object-oriented software diagrams. In *Proceedings of the 5th International Symposium on Graph Drawing (GD '97)*, volume 1353 of *LNCS*, pages 415–424. Springer, 1997.

[SF07] Markus Scheidgen and Joachim Fischer. Human comprehensible and machine processable specifications of operational semantics. In *Model Driven Architecture- Foundations and Applications*, volume 4530 of *LNCS*. Springer-Verlag, 2007.

[SFvH09a] 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'09)*, Potsdam, Germany, 2 June 2009.

[SFvH09b] Miro Spönemann, Hauke Fuhrmann, and Reinhard von Hanxleden. Automatic layout of data flow diagrams in KIELER and Ptolemy II. Technical Report 0914, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2009.

[SFvHM10] Miro Spönemann, Hauke Fuhrmann, Reinhard von Hanxleden, and Petra Mutzel. Port constraints in hierarchical layout of data flow diagrams. In *Proceedings of the 17th International Symposium on Graph Drawing (GD'09)*, volume 5849 of *LNCS*, pages 135–146. Springer, 2010. doi:10.1007/978-3-642-11805-0_14.

[SG08] Maik Schmidt and Tilman Glötzner. Constructing Difference Tools for Models Using the SiDiff Framework (Informal Research Demonstration). In *ICSE 2008 Companion Proceedings, 30th International Conference on Software Engineering*, Leipzig, May 2008.

[SGD05] Ahmed Seffah, Jan Gulliksen, and Michel C. Desmarais. An introduction to human-centered software engineering. In *Human-Centered Software Engineering—Integrating Usability in the Software Development Lifecycle*, volume 8 of *Human-Computer Interaction Series*, pages 3–14. Springer Netherlands, 2005. Available from: http://www.springerlink.com/content/r604rw4446271577/, doi:10.1007/1-4020-4113-6_1.

[SM91] Kozo Sugiyama and Kazuo Misue. Visualization of structural information: automatic drawing of compound digraphs. *IEEE Transactions on Systems, Man and Cybernetics*, 21(4):876–892, Jul/Aug 1991. doi:10.1109/21.108304.

[Spö09] 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.

[ST99] Janet M. Six and Ioannis G. Tollis. Circular drawings of biconnected graphs. *LNCS*, 1619:57–73, 1999.

[Sta74] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, Wien, December 1974.

[STT81] 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.

[Sug02] Kozo Sugiyama. *Graph drawing and applications for software and knowledge engineers*. World Scientific Pub Co Inc, 2002.

[SV02] Georg Sander and Adrian Vasiliu. The ILOG JViews graph layout module. In *GD 2001: Proceedings of the 9th International Symposium on Graph Drawing*, volume 2265 of *LNCS*, pages 469–475. Springer-Verlag, 2002.

[SZN04] C. Schneider, A. Zündorf, and J. Niere. CoObRA - a small step for development tools to collaborative environments. In *Workshop on Directions in Software Engineering Environments; 26th International Conference on Software Engineering*, Scotland, UK, 2004.

[Tam87] Roberto Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM Journal of Computing*, 16(3):421–444, 1987.

[Tay96] Conrad Taylor. What has WYSIWYG done to us? *The Seybold Report on Publishing Systems*, 26(2), September 1996.

[TBWK07] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. Difference Computation of Large Models. In *ESEC-FSE '07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 295–304, New York, NY, USA, 2007. ACM. doi:http://doi.acm.org/10.1145/1287624.1287665.

[TDB88] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man and Cybernetics*, 18(1):61–79, 1988.

[Tou07] Antoine Toulmé. Model Comparison Panel. In *EclipseCon 2007*, Santa Clara, California, March 2007.

[Tra10] Claus Traulsen. *Reactive Processing for Synchronous Language and its Worst Case Reaction Time Analysis*. PhD thesis, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, 2010. http://eldiss.uni-kiel.de/macau/servlets/MCRFileNodeServlet/dissertation_derivate_00003253/ClausTraulsen.pdf.

[TT86] Roberto Tamassia and Ioannis G. Tollis. A unified approach to visibility representations of planar graphs. *Discrete and Computational Geometry*, 1(1):321–341, 1986.

[TT08] Malte Tiedje and Claus Traulsen. Designing a reactive processor with Esterel v7. In *Proceedings of the Workshop on Model-Driven High-Level Programming of Embedded Systems (SLA++P'08)*, Budapest, Hungary, April 2008.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000. doi:http://doi.acm.org/10.1145/352029.352035.

[vDvKSW02] Steven van Dijk, Marc van Kreveld, Tycho Strijk, and Alexander Wolff. Towards an evaluation of quality for names placement methods. *International Journal of Geographical Information Systems*, 2002. Available from: `citeseer.ist.psu.edu/698990.html`.

[vH09] 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.

[VM95] Gerhard Viehstaedt and Mark Minas. DiaGen: A generator for diagram editors based on a hypergraph model. In Amihai Motro and Moshe Tennenholtz, editors, *Proc. 2nd International Workshop on Next Generation Information Technologies and Systems (NGITS'95)*, pages 155–162, Naharia, Israel, June 1995.

[Völ08] Jonas Völcker. A quantitative analysis of Statechart aesthetics and Statechart development methods. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2008. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jovo-dt.pdf`.

[182] World Wide Web Consortium (W3C). Scalable vector graphics (svg) 1.1 specification. W3C Recommendation, edited in place 30 April 2009, 2003. `http://www.w3.org/TR/2003/REC-SVG11-20030114/`.

[WEK01] Roland Wiese, Markus Eiglsperger, and Michael Kaufmann. yFiles: Visualization and automatic layout of graphs. In *GD 2001: Proceedings of the 9th International Symposium on Graph Drawing*, volume 2265 of *LNCS*, pages 588–590. Springer-Verlag, 2001.

[Wil97] Graham J. Wills. Nicheworks—interactive visualization of very large graphs. In *GD '97: Proceedings of the 5th Interna-*

tional Symposium on Graph Drawing, pages 403–414, London, UK, 1997. Springer-Verlag.

[WMP+05] Pak Chung Wong, Patrick Mackey, Ken Perrine, James Eagan, Harlan Foote, and Jim Thomas. Dynamic visualization of graphs with extended labels. In *INFOVIS '05: Proceedings of the Proceedings of the 2005 IEEE Symposium on Information Visualization*, page 10, Washington, DC, USA, 2005. IEEE Computer Society. doi:http://dx.doi.org/10.1109/INFOVIS.2005.11.

[WPCM02] Colin Ware, Helen Purchase, Linda Colpoys, and Matthew McGill. Cognitive measurements of graph aesthetics. *Information Visualization*, 1(2):103–110, 2002.

[XS07] Zhenchang Xing and Eleni Stroulia. Differencing logical UML models. *Automated Software Engg.*, 14(2):215–259, 2007. doi:http://dx.doi.org/10.1007/s10515-007-0007-3.

[ZGH+07] Nianping Zhu, John C. Grundy, John G. Hosking, Na Liu, Shuping Cao, and Akhil Mehra. Pounamu: A meta-tool for exploratory domain-specific visual language tool development. *Journal of Systems and Software*, 80(8):1390–1407, 2007.

[ZL08] Ye Zhou and Edward A. Lee. Causality interfaces for actor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, pages 1–35, April 2008. Available from: http://chess.eecs.berkeley.edu/pubs/473.html.

Appendix A

Tool Examples and Categorization

There are different aspects of the tools discussed in the following. For more detailed categorizations of Human-Machine Interface (HMI) aspects refer to the literature [Die07].

Window style The tools either present only one window in the operating system or multiple windows. In the one-window solution, multiple different aspects are usually shown in different sub-windows, which are arranged within the main window. Hence, the application itself has full control of what aspect of the tool is shown where. When using multiple windows, the different aspects are shown each in a separate Operating System (OS) window and therefore usually not much guidance is given on how to arrange those windows properly if not provided by the OS itself.

Focus Every development task that needs to interact with the model—e. g., editing—needs to present the model to the user. If the model is too large for the screen, it needs a strategy how to display only the relevant part as the *focus*. Most of the presented tools use explosive zooming and display only the fixed static view that the tools support, either with hidden or visible hierarchy as discussed in Section 1.2.

Context For every development task the user has to keep the necessary model parts in mind. If the model is too large to fit entirely into the screen, only a part is shown as the focus. To still be able to orientate in the model, the tool can present the *context* in which the current focus is embedded. Many tools use a small overview of a bigger model part and/or a tree view of the whole model.

Multiple Views Following the Model-View-Controller (MVC) pattern, a view is only one representation of a model. Sometimes it is useful to create multiple different views of the same model artifacts, e. g., in different abstraction levels. While we try to do this dynamically and with view management even automatically (cf. subsection 3.2.5), we are not aware of any tool that supports this. However, single tools have a separation between model and view and allow to create multiple views on the same model. Still, these views have to be created by hand.

Layout Usually the layout of a diagram has to be performed manually as all investigated tools are freehand editors because we are not aware of any other approach that is a full-blown IDE. However, some tools offer little layout help such as grids, alignment, distribution or nudge[1] tools. Some tools even provide rudimentary automatic layout algorithms.

[1] push objects a few pixels in some direction

Ptolemy

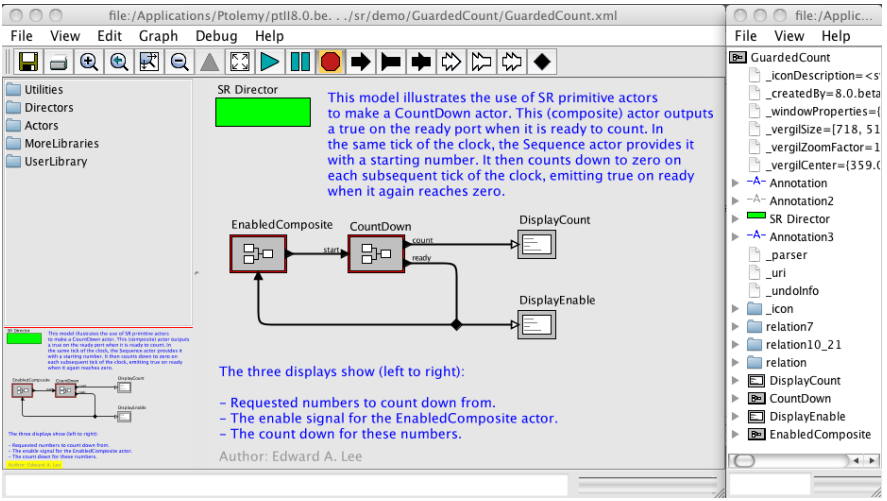


Figure A.1. Screenshot of the Ptolemy Editor Vergil v.8

The Ptolemy II editor Vergil presents a multi-window user interface. The main window shows the model focus as hidden hierarchy only, the actor library, and a small outline view onto the current model part. Navigation to another actor opens another of these windows on top of the other which has to be rearranged manually. Next to the outline there is a tree or an XML view available, which opens in another window and displays the context in a non-graphical representation. The Ptolemy persistence format does not separate view information (e. g., coordinates) from structure information and hence, there is only one diagram available for a model.

Vergil includes a rudimentary routing mechanism that is always used to render connections. It implements a plain orthogonal routing style and, hence, the user only has to place nodes and cannot directly route edges. However, the routing approach is rather straight, such that it often results in connection-node-overlaps. Therefore users often introduce dummy-relation

nodes to actually reroute connections manually. In addition to the approach presented in Section 5.4, there is a rudimentary layout algorithm included in Vergil which does not conform to essential aesthetic criteria such as avoiding node-overlaps.

Matlab/Simulink

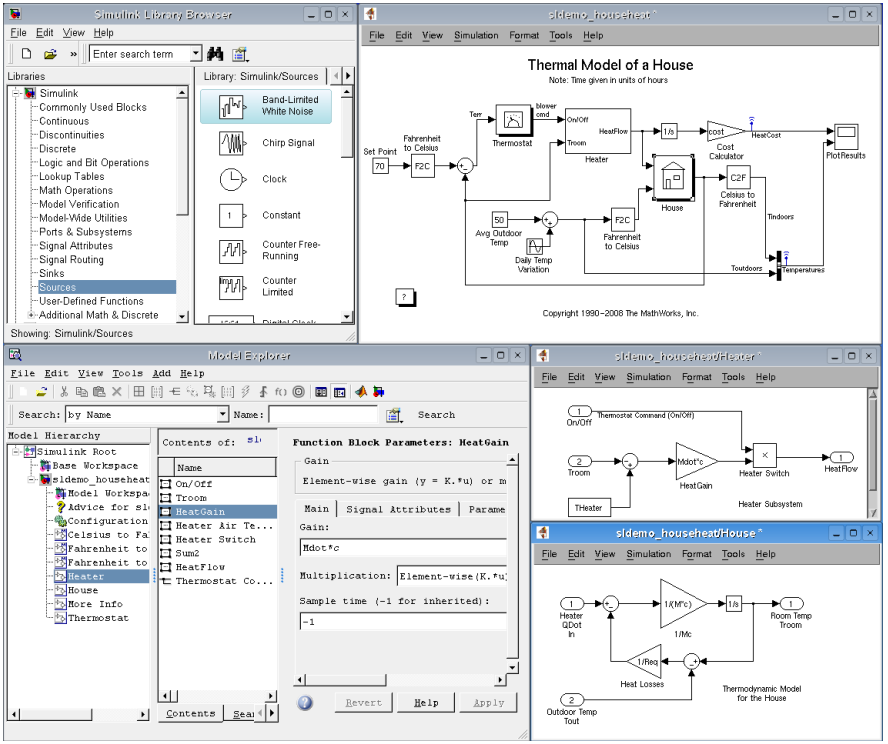


Figure A.2. Screenshot of Matlab/Simulink v.7.9

The Matlab Desktop is an integrated window with multiple functionality for the textual Matlab language itself. However, the graphical Simulink part on top of Matlab uses multiple windows. Every model focus is shown in a separate window using explosive zooming for navigation. Model and view are stored in the same file and, hence, there is no possibility to create multiple views of the same model. No outline view is available. A quite verbose model explorer window is provided as an enriched tree view to

navigate in a model. The operator library is also available as a separate window. Simulink uses only hidden hierarchy. Stateflow is a Statecharts dialect that can be embedded into a Simulink model. However, the editors are quite different and not well integrated. Stateflow allows visible or hidden hierarchy.

Simulink itself provides no automatic layout functionality. However, in the big community around Simulink, first approaches on this emerge, e. g., by Klauske [KD10].

ASCET

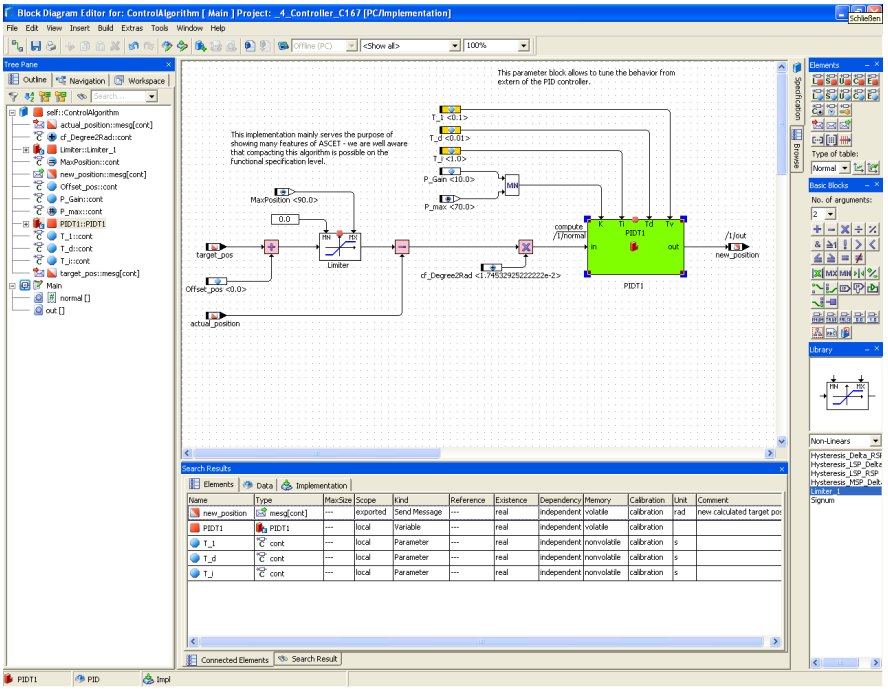


Figure A.3. Screenshot of ASCET v.6

The ETAS GmbH provides the Advanced Simulation and Control Engineering Tool (ASCET). It is a one-window IDE presenting a main canvas for the model part in the focus and some sub-windows. These offer a tree view for the context—no outline is provided—and some other information such as selection properties or operator libraries. Like all investigated data flow editors, ASCET only provides hidden hierarchy with explosive zoom navigation.

SCADE

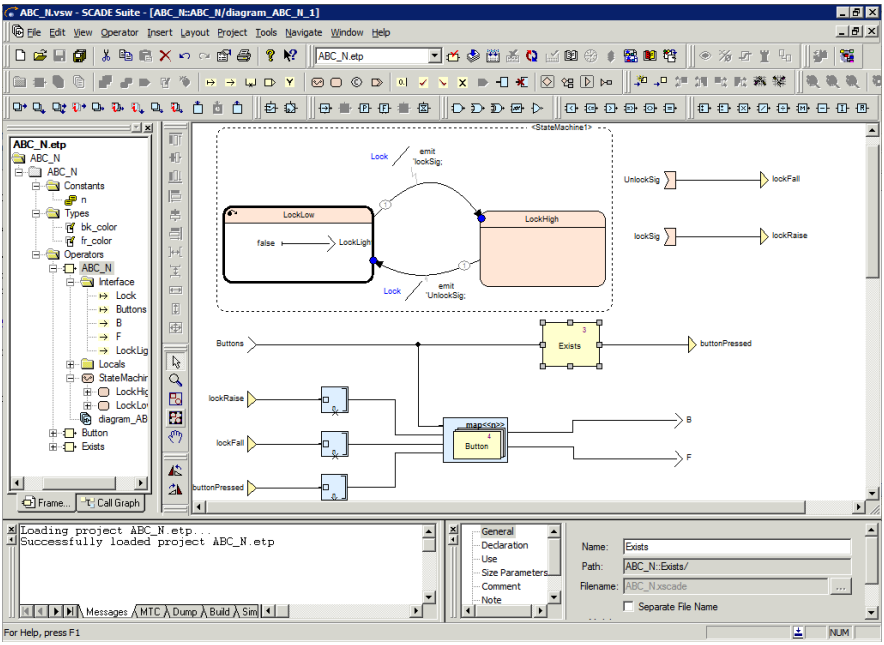


Figure A.4. Screenshot of SCADE v.6.1

The Safety Critical Application Development Environment (SCADE) was developed by Esterel Technologies. It is also a one-window IDE. It presents a central part for the focus canvases, however, multiple canvases may be arranged next to each other by the user if there is enough space. SCADE offers a tree view but no outline. Many possible tool bars have to be arranged by the user to avoid a button chaos. Since version 6 SCADE offers control flow and data flow diagrams that may be mixed into the same canvas. For the data flow parts only hidden hierarchy is displayed, while for the state machine parts also visible hierarchy is used. For navigation of hidden parts explosive zooming is employed.

SCADE offers some simple layout tools such as alignment, distribution and nudging (pushing objects a few pixels in some direction). To support the connection of many ports of two operators with each other, there are "connect by rank" and "connect by name" operations that create multiple connections between the corresponding ports.

Esterel Studio

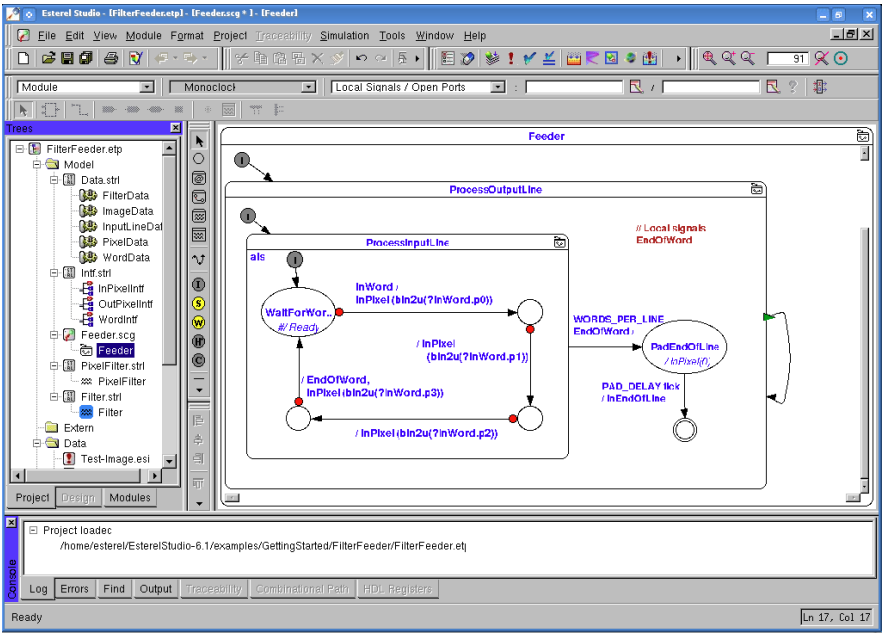


Figure A.5. Screenshot of E-Studio v.6.1

Esterel Studio is another one-window application. It presents SyncCharts, resp. Safe State Machines, in a focus canvas with a tree view and no outline. As discussed in Section 1.2, E-Studio allows different ways to display hierarchy; hidden or visible or hybrids. Still, the kind of hierachy visualization has to be chosen once for every model part. Multiple views for one model are not possible.

Alignment tools like in SCADE are also available in E-Studio.

Eclipse

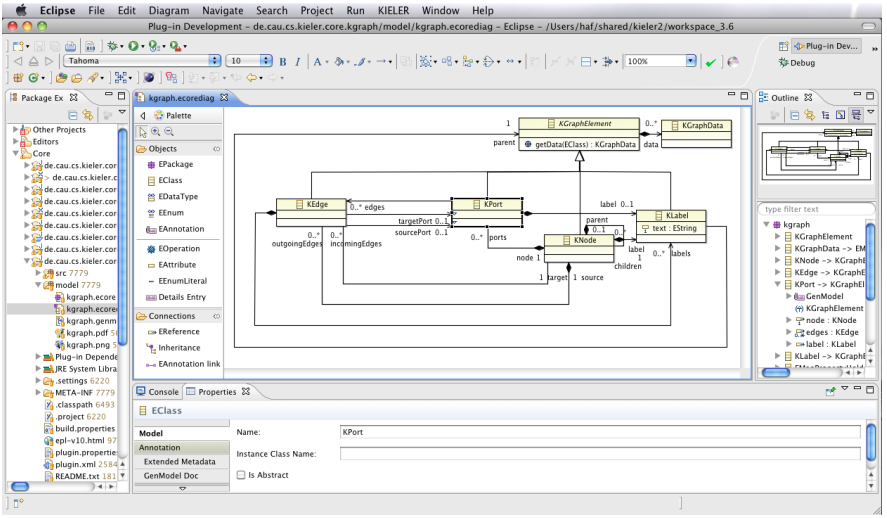


Figure A.6. Screenshot of Eclipse v. 3.6 with the Ecore Tools diagram editor as a typical example for a GEF based editor.

Eclipse itself only defines the window concept. It has one main window in which it manages two kinds of main entities: editors and views. This separation is important in Eclipse and has some implications. An *editor* is a representation of the contents of a resource—either a file or some resource from a network. A central area in eclipse is reserved for these editor subwindows. There may be multiple editors open at the same time. Usually they get stacked with tabs but they may also be displayed next to or on top of each other. However, one file may be opened by one editor only once. Next to editors there are *views*—not to be confused with the notion of model views we have used so far. A view is a subwindow that may be arranged around the editor area. It usually has no own input resource but is only a representation of a single aspect of something currently visible or selected in an editor. Common views are the project explorer, an outline

view and a properties view. The last always shows detailed properties for any object that is currently selected in an editor.

Features of graphical editors within Eclipse are not fixed, they are defined by the corresponding editor plug-in. As discussed in Section 1.1 for graphical editors common frameworks are often used such as the Graphical Modeling Framework (GMF) or Graphiti, which both employ the GEF library. They also offer common features. As one representative, Figure A.6 shows an Ecore Tools diagram editor for Ecore class diagrams. It provides an outline and tree view and alignment tools. There comes a rudimentary layer-based layouter with it, which is only of limited use for class diagrams; no parametrization from the UI is possible. In general, GEF based editors can use hidden or visible hierarchy. In case of visible hierarchy, the corresponding *compartments* can be collapsed or resp. expanded by the user. However, as no useful automatic layout yet exists, the user has to manually create these static diagrams. GMF defines a persistence format which separates models from views. Therefore, the user may create multiple diagrams for the same model.

Visual Paradigm

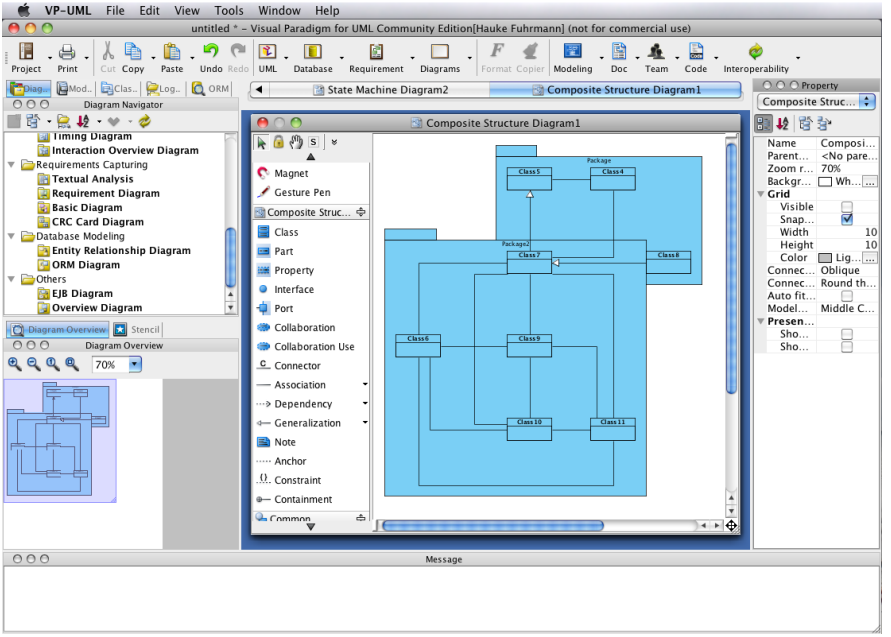


Figure A.7. Screenshot of Visual Paradigm v.8.

Visual Paradigm of the likewise called company is a UML modeling environment with some advanced features regarding pragmatics. It shows a one-window concept with explosive zooming where only one diagram at the time can be displayed. Hierarchy is mainly hidden. Only some simple ways of hierarchy are visible such as packages. Submachines in State Machine diagrams for example are hidden with explosive zooming only. It has an outline and tree view. Model elements may be reused in multiple diagrams, therefore multiple views can be created. Its focus is still on freehand editing and therefore it offers some helping tools to arrange the layout such as alignment and distribution tools, dynamically overlayed

guide lines for alignment, a broom tool (like the classic ArgoUML [RR00]) to make new empty space, and tools to ease selection, called "Handi-Selection", which selects all elements above, below, left or right of the current mouse pointer position.

Next to these helpers for freehand editing, there is a noteworthy collection of automatic layout algorithms available. According to documentation and the running tools, there are the following layout styles available: Orthogonal Layout, Hierarchic Layout, Directed Tree Layout, Balloon Tree Layout, Compact Tree Layout, Horizontal-Vertical Tree Layout, BBC Compact Circular Layout, BBC Isolated Circular Layout, Single Cycle Circular Layout, Organic Layout, Smart Organic Layout, Organic Edge Route Layout and Orthogonal Edge Route Layout.

Due to the missing visible hierarchy, the layout algorithms mainly only work on plain graphs, e. g., class diagrams. Some of them already also work on clustered graphs. Figure A.7 shows a class diagram with packages employing orthogonal layout. Some of the layouters will cause infeasible layouts when using packages, like creating package node overlaps. This seems to be also semantically incorrect as the tool merges the packages when the user tries to move them afterwards. However, for single plain diagrams the selection of layouters seems to be quite useful. The different algorithms can also be parametrized by the user, for example to specify the layout direction for hierarchic layout (aka layer-based layout).

Still, the tool focuses on freehand editing. Automatic layout seems to be only integrated as a helper that has to be manually triggered from the context menu. There are no features like structure-based editing or any form of focus&context available. The tool even advertises full round-trip engineering support and is able to generate Java and C++ source code from class diagrams. It can parse changes in the source code and integrate them back into the diagrams. However, the automatic layout functionality is not yet integrated with this feature such that the user has to update the layout him- or herself. Hence, there is nothing available like the view management approach that we propose in Chapter 3.

Online Tools: yUML

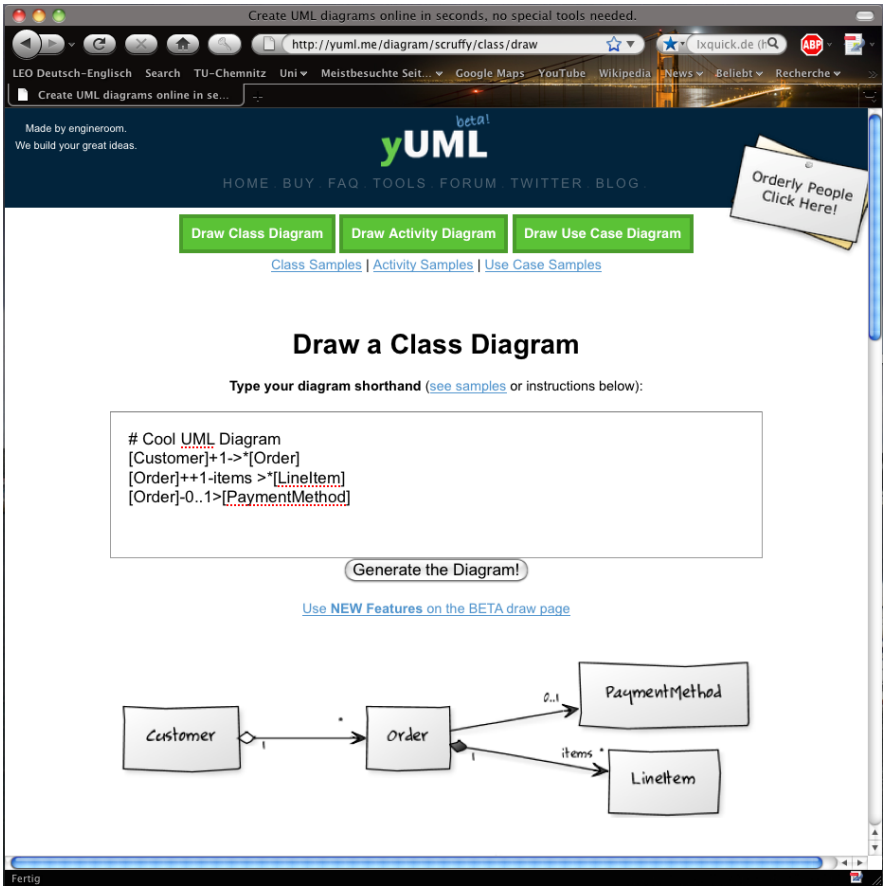


Figure A.8. Screenshot of yUML beta.

We take yUML[2] as a representative for online UML diagram tools basing

[2]http://yuml.me

on a textual notation. It presents itself as a website that can be shown by a normal web browser. No special techniques are required at the client side, e. g., no flash or extensive javascript.

It defines a textual syntax for Class, Activity and Use Case diagrams. The user can add the textual description in a little text field without any content assist. A little syntax overview is shown at the bottom of the page. A button triggers generation of the diagram which will be shown after some seconds below the text field. In the current beta version errors in the syntax are not caught and there is no feedback to the user.

The diagram is a bitmap and the tool offers links to it such that it can be embedded into any website conveniently. The link address stores the whole textual description of the diagram such that it seems the diagram is always regenerated on-the-fly when the image is accessed. This might explain some performance issues. Next to bitmaps, also a PDF can be exported as vector graphics.

The layout of the diagram is generated automatically, however, the algorithm seems to be a layer-based approach and looks very similar to GraphViz dot. Hence, it often results in suboptimal layouts for class diagrams, where it fails to follow conventions for the different routing of generalizations and associations. Additionally, relation labels often overlap.

There are other such approaches such as `http://www.websequencediagrams.com` or PlantUML, `http://plantuml.sourceforge.net`. The latter provides many UML diagrams rendered with the GraphViz dot layouter with results similar to the aforementioned tool. Sequence diagrams are layouted with a custom layouter, which produces quite good results. Its strength is its flexible technical possibilities, because it can be used on-line but also as a pure Java library as well as integrated as a plug-in into Eclipse. Integrations with real modeling environments, however, are not provided.

Online Tools: Oryx

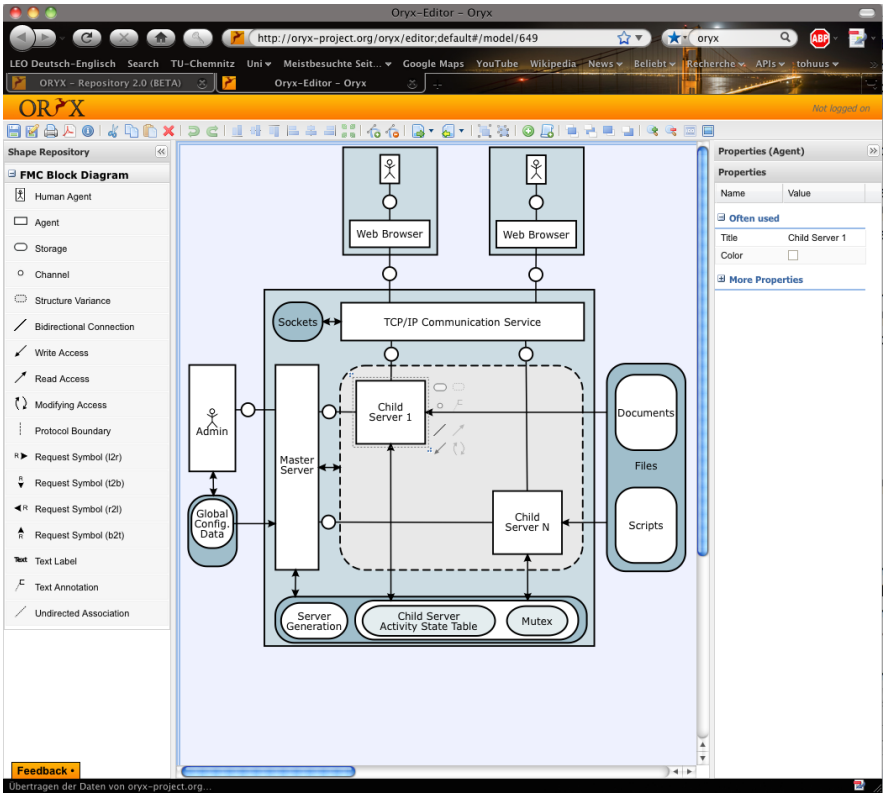


Figure A.9. Screenshot of Oryx with an FMC block diagram v.2.0 beta

Oryx [3][DOW08] is a web platform for model editors. While editors are implemented according to an API, they do not take shape as rich clients but use a web browser to render the diagram and act as an interactive client. Advanced interaction techniques for the web, such as javascript, are

[3] `http://oryx-project.org`

extensively exploited to create the user experience of a "normal" rich client. It presents a toolbar, a palette containing a diagram object library, the main canvas and a property frame. The main focus seems to be on the usage of the web medium and not on new pragmatics. Hence, it offers only the usual freehand DND editing. A few alignment tools are offered but no automatic layout functionality. There seems to be no separation between model and view and hence there is only one diagram available for a model and only one model canvas at a time. Some of the modeling languages contain visible hierarchy. Modularization into multiple diagrams or dynamic hierarchy seems to be not provided.

One advantage might be the possible usage of an online repository of models, which can easily be browsed. Hence, it is good means for collaborative modeling where sharing of diagrams is easily possible.

Summary

The shelves are full of UML tools where Visual Paradigm is one representative. Some have also advanced features regarding pragmatics. Gentleware for example provides the UML suite *Poseidon*, which offers similar freehand editing capabilities. *Apollo* of Gentleware is an Eclipse based Class diagram editor supporting round-trip engineering. It employs the commercial automatic layout library of yWorks to get a full diagram-source code synchronization for this single use-case. However, most of the tools are commercial and some even do not provide proper testing licenses, so we cannot give a full survey here.

In general, we got the impression that long-established modeling environments—especially for the development of embedded reactive systems—disregard the aspects of their tool pragmatics. Meanwhile in the market of software engineering tools employing the UML tools emerge that start to offer some single useful features addressing pragmatics. This might arise from the competition in this sector.