Christian-Albrechts-Universität zu Kiel

Diploma Thesis

Layout and Visual Comparison of Statecharts

cand. inform. Arne Schipper

December 18, 2008

Department of Computer Science
Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

Advised by:
Hauke Fuhrmann

Eidesstattliche Erklärung

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

Kiel,

_____

Abstract

Statecharts are a generally accepted method to model safety critical reactive systems, reducing modeling errors of the developers. As Statecharts are inherently graphical, two problems arise when working with them. One issue is to receive a sound layout of Statecharts automatically, the other addresses the comparison of Statecharts at the diagram level.

Much research has already been done considering layout and automatic layout of Statecharts. This thesis continues these works and provides an implementation of a framework enabling Statechart layout in Eclipse. Great importance is attached to meta layout facilities, enabling different layout types for different parts of a Statechart. This can be exploited by pattern-based layout, increasing the readability and comprehensiveness.

Working with projects usually results in the need to compare different versions of files. So far, there is no useful solution to compare Statecharts visually. This thesis evaluates several possible approaches and presents one promising one.

As a proof of concept, and for later use in a meta-tool framework, both proposals are implemented in Eclipse.

Keywords Safe State Machine, layout, meta layout, visual comparison, model differences

Contents

List of Figures

List of acronyms and abbreviations

CASE	Computer-Aided Software Engineering
CoObRA	The Concurrent Object Replication frAmework
CVS	Concurrent Versions System
DSL	Domain-Specific Language
ELP	Edge Label Placement
EMF	Eclipse Modeling Framework
EMOF	Essential Meta Object Facility
FUJABA	From UML to Java And Back Again
GEF	Graphical Editing Framework
GEMS	Generic Eclipse Modeling System
GMF	Graphical Modeling Framework
GPL	GNU General Public License
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
JNI	Java Native Interface
KIEL	Kiel Integrated Environment for Layout
KIELER	Kiel Integrated Environment for Layout for the Eclipse Rich Client Platform
KIML	KIELER Infrastructure for Meta Layout
KiViK	KIELER Visual Komparison
KSSM	KIELER Safe State Machine
MDA	Model-Driven Architecture

MDD	Model-Driven Development
MDE	Model-Driven Engineering
MOF	Meta Object Facility
MVC	Model-View-Controller
NLP	Node Label Placement
PCB	Printed Circuit Board
RCP	Rich Client Platform
RUP	Rational Unified Process
SCM	Software Configuration Management
SNF	Statechart Normal Form
SQL	Structured Query Language
SSM	Safe State Machine
STG	State-Transition Graph
SVG	Scalable Vector Graphics
SVN	Subversion (version control system)
SWT	Standard Widget Toolkit
UML	Unified Modeling Language
VCS	Version Control System
WYSIWYG	What you see is what you get
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations

"Wo aber Gefahr ist, wächst das Rettende auch."

Friedrich Hölderlin - Patmos

1 Introduction

There are many things people claim that *make the world go round*. Whether it is arguable what belongs to this group, one must say that embedded systems do. Embedded systems are computers that are integrated into larger environments, but are not recognized as such. They exist in cars, washing machines, toys. One can find them in nuclear power plants as well as in wind power stations, in medical devices just as in mobile phones. The list is vast. Just as mankind is getting more used to the convenience offered by those technical equipments, the more it is becoming a slave to them. Hence, embedded systems really make the world go round.

Software development in general, and in particular for embedded and real-time systems, has evolved a lot during the last decades. As such systems often serve in harsh and safety-relevant environments, close attention must be paid when developing them. In many systems any error can be life-threatening. Techniques in avoiding or reducing programming mistakes often involve the development environment. Many Integrated Development Environments (IDEs) support the user not only in things such as syntax highlighting and automated build processes, but also in modeling the system's behavior, leading to more robust programs.

Well-known techniques in modeling incorporate the Unified Modeling Language (UML), which is standardized by The Object Management Group. A huge number of development tools makes use of this standardized language to express the systems behavior in a graphical manner, helping the developer in understanding the system. Crucial points are the way how the information is formatted and shown to the user. That is by no means a trivial question. A poor graphical representation will diminish the intended gain in clarity and robustness of the development process. Conversely, it is of greater importance to the success of system modeling that the 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 from switching of different abstraction levels, when trying to map the textual description of the differences to the diagram.

To achieve the goal of a meaningful visualization of changes, three challenges have to be faced.

1. Synthesize the structural differences between the different versions of the model.
2. Map those changes to the existing diagram or diagrams, keeping layout issues in mind.
3. Present the changes graphically in a way that is easy to grasp from the developer.

Addressing point 1, it is essential to stress that getting the *structural* differences of a model is completely different from getting the differences of a normal text file (Kelter, 2007).

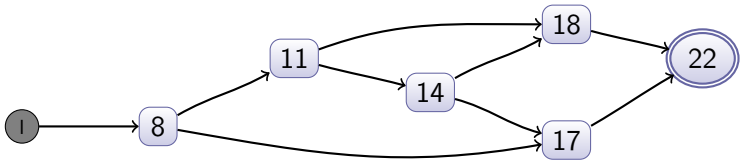
Point 2 and 3 interact together, but have nevertheless some differences. In many layouters, such as the dot-layouter of the graphviz suite (GraphViz, 2007), a model (in this case a graph description) is passed to the layouter, which performs the positional computation and outputs the result. If the graph is changed, the whole process for the entire graph is done again.

Normally this is performed without keeping the previous layout in mind, leading to a new overall look and confusing the developer. When working with a diagram, the user builds up a *mental map* (Eades et al., 1991; Misue et al., 1995), that means she or he keeps in mind the rough overall structure of the graph. This map of the structure should not change considerably during editing operations of the diagram. Such an editing step can be seen in Figure 1.1 [1]. Note the bad preservation of the mental map when adding node *69*, switching the positions of node *11* and of nodes *17* and *18* with the default configuration settings of *uDrawGraph* (Universität Bremen, 2005). Especially with larger graphs or models, it can be time consuming for the user to rebuild her or his mental map to incorporate the modifications. Adjusting just the changed parts of the graph could help in this case (Biedl and Kaufman, 1997). This also applies to performance issues, as in this scenario the whole graph does not need to be rendered again.

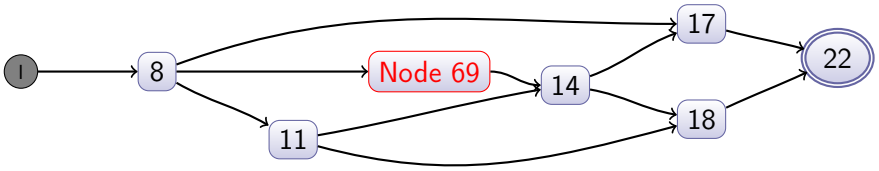
1.1. Related Work

Beginning with the diff tool (Hunt and McIlroy, 1976; Myers, 1986), 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. Papers of Chawathe et al. (1996) and Chawathe and Garcia-Molina (1997)

[1] The actual look of the statechart is not the one generated by the respective tools—here and later in the thesis—but an adapted presentation by the author to receive the same appearance throughout the thesis. However, the placement of the states is as calculated by the tools.



(a) original layout



(b) after inserting *Node 69*

Figure 1.1.: Example of automatic incremental layout, produced with *uDrawGraph* (Universität Bremen, 2005).

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. This method is applied to documents written in the Extensible Markup Language (XML) by Ohst et al. (2003a,b).

A good introduction in difference building gives Kelter (2007). An overview how difference algorithms are used in Software Configuration Management (SCM) tools give Conradi and Westfechtel (1998). Applications of that are the Concurrent Versions System (CVS) (GNU, 2006) and the designated replacement Subversion (SVN) (Collab.Net, 2008).

Considering the problem of synthesizing the structural differences of models, there are several approaches discussed nowadays. An important issue is to be aware of the fact that the semantics of the model has to be taken into account to achieve a meaningful result, as the same or a similar syntax may mean different behavior. Even though there are standards like the UML, this is not trivial. The semantics can differ from version to version of a standard, maybe just in small but important points. For every Domain-Specific Language (DSL) the algorithm for the differences has to be adjusted to map the exact meaning of the language. Moreover, the user must have a profound knowledge of the semantics to interpret the output of such an algorithm correctly. Finally, a general scheme perhaps for the UML does not fit all the aspects of the UML, but has to be refined to apply to the corresponding diagram types, as for example Statecharts or Class diagrams.

Considering real models, an interesting approach is used by The Concurrent Object Replication frAmework (CoObRA) of Schneider (2003); Schneider et al. (2004). The model elements themselves are considered as objects in a Version Control System (VCS). Every operation carried out 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 SiDiff (Schmidt, 2007; Schmidt and Glötzner, 2008; Treude et al., 2007; Wenzel and Kelter, 2006). Input models are transformed to an internal data structure. The structure-based diff is then executed with this 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 plugins in the round trip engineering tool From UML to Java And Back Again (FUJABA) (University of Paderborn, 2006).

A similar method to the previous is used by EMF Compare of Brun (2007, 2008) and Toulmé (2006, 2007), employed in the Eclipse Modeling Framework (EMF). This is a plug-in which extends the normal compare function of Eclipse (Eclipse Software Foundation, 2008) by the support for EMF models (Merks, 2008). It uses a two stage mechanism. First, it tries to find matches between the elements of the different versions with various metrics. Second, the generated matching model is processed to extract the differences, those being translated into *added*, *deleted* or *changed*. The matching and differencing algorithm was inspired by work of Xing and Stroulia (2005, 2007). EMF Compare is fully integrated into Eclipse.

Another work focussing on Eclipse is a plug-in suite of Mehra et al. (2005). 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, though the representation lacks some features.

SCADE from Esterel Technologies, Inc (2008) is another development environment used to design safety-critical applications. It has a plug-in—*SCADE Model Diff*—to analyze differences between two *SCADE* models or two versions of a model. A screenshot can be seen in Figure 3.10. 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 so called location window, exhibiting two graphical models—*SCADE* models—with highlighted differences. Furthermore it is possible to generate a textual report of the changes. No support for Statecharts is given and the user handling is complicated.

Finally, there are borrowings to several techniques in the tools *Poseidon* and *Apollo* of Gentleware AG. The mechanism for the round trip engineering depends on a model diff facility. That applies as well to the automatic layouter, which is capable of producing two types of layout: an automatic layout for imported diagrams, and an incremental layout while working with a diagram. If for example new classes are added to an existing class diagram, they are aligned smoothly without altering too much of the existing chart. The layouting engine used is from yWorks (2005).

1.2. Objective

The main objective of this thesis is to develop a method to visualize the differences of distinct versions of a model, in particular of a Statechart. The goal also includes a case study to verify if this approach is reasonable in production environments. As there are many tools and algorithms available that already perform parts of the requirement of the visual comparison, an important part of the thesis is searching and evaluation of relevant, existing methods. The implementation will not be done from scratch, but an existing modeling framework will be used. This tool, the KIELER (CAU Kiel), is actively developed at the Christian-Albrechts-Universität Kiel. It emerged from another experimental layouting tool, the Kiel Integrated Environment for Layout (KIEL) (The KIEL Project, 2006). A more detailed description will be given in Section 5.1.

To sum up the objectives:

- Identify promising model diff and layout algorithms
- Combine them to a visual diff
- Implement the result in KIELER as a proof of concept
- Evaluate the implementation in a case study

1.3. Overview

Chapter 2 provides an overview of model-based system design in general. The main benefit of this philosophy is exposed and serves as a further motivation for this thesis. Single topics involved with this thesis are covered more in detail. That are particularly Statecharts, their representation and semantics, as well as concepts for layouting the very same.

In Chapter 3 state of the art tools in model comparison and layout are presented and compared. This applies to a certain extent also to methods visualizing structural differences. Drawbacks are discussed and benefits to model based system design of such mechanisms are explained.

Chapter 4 defines the visual diff problem and elaborates an approach to it. The theoretical foundations are stated taking into account several usability and aesthetic metrics.

As a proof of concept, Chapter 5 shows the implementation using the KIELER framework based on the Eclipse plug-in concept. A short description of the KIELER framework is also given.

In Chapter 6 the adequacy of the approach is measured by two case studies, operating on production models taken from industry.

The thesis closes with a conclusion in Chapter 7, summarizing the results and contribution and giving an outlook to future research.

"Es versteht sich von selbst, dass die Bourgeoisie die modernen Kriegsmethoden, welche immer mehr mechanisiert und wissenschaftlich vervollkommnet werden, in Revolutionszeiten auch gegen den "inneren Feind" gebrauchen wird. Nun gibt es, um eine siegreiche Revolution gegen konterrevolutionäre Angriffe zu verteidigen, bloß zwei Möglichkeiten: Entweder man bringt die Arbeiter dazu, dass sie alle Kriegsindustrie lahmlegen, oder man versucht, dieselben Mittel zu ergreifen, um die Bourgeoisie mit ihren eigenen Waffen zu bekämpfen. Es ist ohne weiteres klar, dass im letzten Fall die größten Schwierigkeiten entstehen."

Artur Müller-Lehning

2 A Brief Review of Model-Based System Design

This chapter gives a short overview over model-based system design in general. As explained in the introduction, software development has evolved significantly during the past decades. The model-driven approach is appealing and as it sets the foundation for this thesis, it shall be explained more in detail. An introduction to model-driven development and the philosophy behind it will be given in Section 2.1. As this thesis mainly applies to one component of this technique, Statecharts, an overview of them will be given in Section 2.2. Issues pertained to layout will be addressed in the last section.

2.1. Model-Based System Design

Models play an important role in science (Frigg and Hartmann, 2006). Examples of such are the billiard ball model of gas, the double helix model of the DNA or different models to explain the nature of light. In science, there exist different semantics of a model. A standard meaning of a model is an entity that fullfills the laws and axioms of a mathematical or logical theory. More practical models are models of phenomena and models of data. The latter is a processed, which means corrected and idealized, set of data obtained from surveys or experiments. More relevant for model-based system design are models of phenomena. These models serve as a scientific representation of systems or nature. Characteristics of these models are for example scale models, idealized models and analogical models. That are those idealized models that help in design. Idealized models are a simplified version of the original entity they are meant to represent. Simplification can take place by stripping away unnecessary information and by distorting facts that are hard to understand

or to transform into the model, leading to a sound and comprehensive explanation of a phenomenon.

Models do also help in developing software. Emanating from the software crisis (Dijkstra, 1972) in the late 60s, structured programming was a first attempt to create more reliable software. A program was split into smaller parts, which performed certain tasks of the program. Confusing statements were considered harmful (Dijkstra, 2002), spaghetti code was vanishing. The object-oriented approach experienced greater attention, though it lasted to the 90s until it was used widely. The design philosophy was to make use of encapsulation, modularity, polymorphism, and inheritance when developing software. Creating software this way was often supported inherently by the programming language of choice. As it was possible to abstract directly from the requirements into the code, using classes and objects, this can be seen as a start of modeling software in the stricter sense.

A thorough planning of software development is a key factor. Many processes, tools and paradigms helped the software designers to challenge this. Designing a program was often supported by drawing diagrams mapping the dependencies and functions of a project or just single files thereof. However, it was not possible to generate code automatically out of this system description. Likewise, as there was no formal specification of a modeling language, there hardly was any support in tools, helping the developers with boring and error-prone tasks, like plausibility and constraint checking. Another drawback was the lack of a standardized process incorporating the whole product development cycle, which means collection of requirements, design, implementation, testing and deployment (Kruchten, 2003).

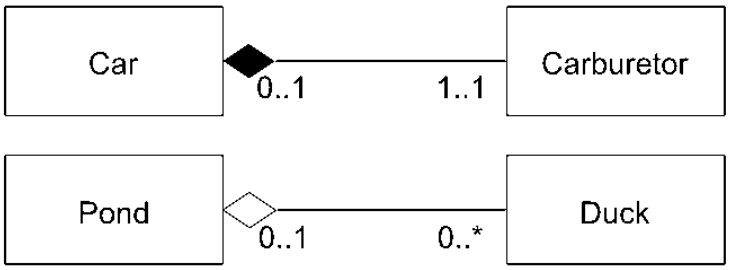


Figure 2.1.: An example of an UML class diagram

Emerging in the 90s, the UML (The Object Management Group) is a protype of a modeling language combining the features mentioned above. In the newest specification, 2.1 (The Object Management Group), there are several different diagram types that help the developer to model a system. Usually a system will be represented by using a class (see Figure 2.1) or a state machine diagram, but the UML offers also many diagrams to support requirement specification and other techniques related to a complete development process. Those techniques are known as Model-Driven Development (MDD) and Model-Driven Engineering (MDE).

When talking about model-based system design in the embedded world, those systems are typically of reactive nature. A system in this understanding is an entity interacting with the environment. It often consists of many distinct parts, though in the exchange with the environment the system can be seen as one object. Reactive systems, in contrast to transformational and interactive ones, are in constant interaction with the environment.

The development of reactive embedded systems faces two main problems. The systems often have to serve in harsh environments and to meet safety critical requirements, therefore they have to be designed and created more thoroughly than standard products. Second, because of those requirements and of the reactive nature they are inherently complex. This is also due to the number of parts such a system consists of and to the dynamic manner in general.

To address the dynamics as well as the concurrency of reactive embedded systems—which is immanent as they have to react in multiple ways to the environment—state-based and dataflow languages have been developed. A well-known formalism describing states and transitions are finite automata. As they lack certain important features, *Statecharts* have been developed.

2.2. Statecharts

With finite automata it was possible to visualize the behavior of a system. The visualization was in that sense an advantage, as it was easier for the human developer to understand the complex behavior of a system at a glance, at least with smaller models. A main drawback of finite automata is the fact that only one state can be active at a time. To describe concurrent activities, many new states have to be added to the system, resulting in an exponential blow-up. Furthermore there is no possibility to express structuring, which means depth, hierarchy and modularity.

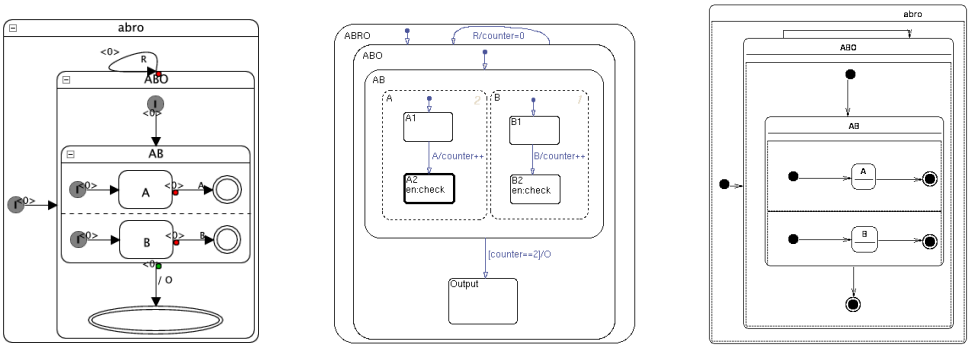
Statecharts (Harel, 1987, 1988) were developed to address these difficulties. They extend Mealy machines basically by hierarchy, orthogonality, broadcast of events and data. The hierarchy adds depth to the automata, so that a logical structuring becomes feasible. By introducing orthogonality, which is conceptually parallelism, the exponential growth of the states can be avoided. With the broadcast concept, two active orthogonal components can communicate with each other. The hierarchy plays an important role when expressing different levels of abstraction of the real world system in a state diagram and keeps the number of states small. Components can be modeled in different levels of detail, providing the developer with a better understanding of the single entities that the whole consists of. Furthermore, analysis can be accomplishable in contrary to unstructured diagrams. That applies to the concept of orthogonality as well.

The original Statecharts of Harel is just one in a family of different dialects. Several others have emerged nowadays. The state machine style used by *Esterel Studio* (Esterel Technologies, Inc, 2007) is called SSM (André, 2003), the successor of *SyncCharts* (André, 1996). Other dialects are *Argos* (Maraninchi, 1991) and the UML variant of

state diagrams, the *State Machines*.

There is a great variety of tools to create and edit *Statecharts*. Among the commonly used are IBM's *Rational Rose* (Rational Rose Realtime), *Simulink/Stateflow* (Mathworks Inc., 2006) from The Mathworks and the above mentioned *Esterel Studio*.

State machine dialects vary in syntax and semantics. This will be covered in the following subsections. Furthermore there are differences in the representation of the charts depending on the modeling tool and the version thereof. In Figure 2.2 there are three Statecharts variants compared with each other. The well-known *ABRO* example is used to depict the differences. Due to problems in the Eclipse UML tools, there are no transition labels drawn in the third example. The most obvious differences follow from the graphical representation of the single elements, as well as from the layout. A difference in the semantics is the possibility to provide the transitions of SSMs and *Stateflow* with priorities, which is indicated by angle brackets.



(a) ABRO in SSMs (b) ABRO in Stateflow (c) ABRO in the Eclipse UML tools

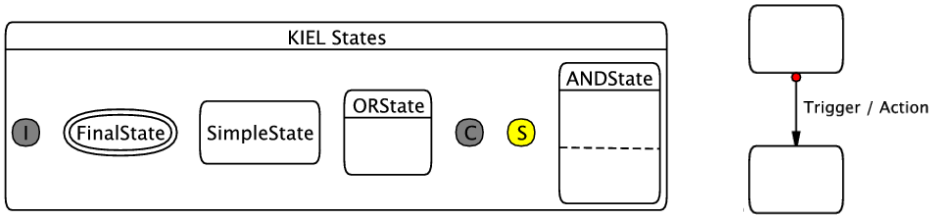
Figure 2.2.: Comparison of three Statechart dialects

2.2.1. Syntax

As this thesis deals with the layout of Statecharts, the main attention has to be drawn to the concrete syntax. Just a short explanation of the general syntax will be given, for a more detailed introduction, especially considering the SSM syntax, see for example Wischer (2006).

A Statechart consists basically of two different sorts of components. That are the states themselves and transitions. States appear in different types, drawn distinctly and comprising different behaviors. Transitions are objects connecting two states with each other. They exhibit two attributes, a *label* and (in some dialects) a *priority*.

In Figure 2.3 the most important states of the SSM editor of KIEL are displayed. The unnamed pictograms—from left to right—are: Initial pseudo state, choice pseudo state and suspend pseudo state. An OR state may contain other states, an AND



(a) The states vocabulary

(b) A transition with a label

Figure 2.3.: The different Statechart components of SSMs

state other states in parallel regions. A final state denotes the end of execution and a simple state is the normal modeling entity. The label of a transition consists of a trigger and an action. A priority is not drawn, if the transition is the only one leaving the state.

2.2.2. Semantics

Many problems arising when working with Statecharts result from a different understanding of the semantics. Due to the variety of the different dialects it is not easy to understand the exact meaning of a chart at a glance. This is complicated by the fact that the same syntax may have different meanings in distinct dialects, as identified by Crane and Dingel (2005). More than a decade ago Beeck (1994) has identified and compared 21 variants of Statecharts, and since then many new ones have been added. Two concepts introduced in Statecharts are the run-to-completion semantics, which applies to UML State Machines, and the synchronous character of the SSMs. Those differences have to be known, but then it is feasible to work and develop as this part of the semantics is clearly stated. Unfortunately there are other vaguenesses in UML State Machines (Crane and Dingel, 2005).

More and severe difficulties yield from an ambiguous definition of concrete semantics. In the 2.0 definition of the UML State Machines, Fecher et al. (2005) discovered 29 inconsistencies and ambiguities. Even if a clear explanation is given, the precise behavior can be hard to understand. Consider the illustration of the grandfather paradox in Figure 2.4. The semantics of the Statechart dialect in use has to be studied thoroughly to understand what will happen here. If event a is absent, then event b will be triggered in the left part of the diagram. But then, if b is detected in the right part, then a will be emitted, which contradicts with the *not* a before.

However, as this thesis mainly covers the problem of visual comparison of temporal versions of models in one Statechart dialect, this issue is not a big one in this context.

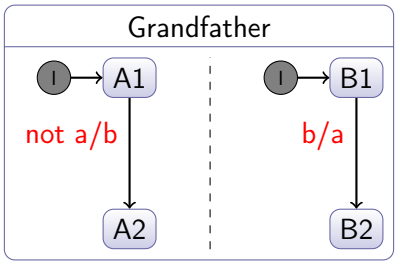


Figure 2.4.: The grandfather paradox (Barjavel, 1943)

2.3. Layout

When working with Statecharts, there is no way to leave out the visual representation, as one aspect of them is to give the reader a graphical description of their behavior. The graphical notation of code, i.e. the behavior, leads to significant differences compared to a textual programming language. The first and formal distinction is the number of dimensions used. The normal written code is just one-dimensional, in contrast to the two dimensional plane where the Statecharts are drawn.

While in source code it is desirable to comply with certain coding standards (see Kernighan and Plauger, 1982), to maintain readability, comprehensibility, maintainability and to avoid errors, that is also applicable to a two-dimensional language like Statecharts (Bell, 2006; Schaefer, 2006). This comprises for example not only a proper naming of identifiers in the code—similar to names of states in state machines—but also and especially a well chosen overall layout. The layout in drawing Statecharts includes the placement of the states, the labels and the transitions, which should adhere to a convenient metric. One aspect of such a metric is of aesthetic nature (see Castelló et al., 2002; Prochnow, 2008; Völcker, 2008). Another important factor is to provide the reader of the diagram with a meaningful secondary notation (see Petre, 1995). For instance, states acting often together or composing a logical unit should be drawn clustered, or the primary direction of the transitions should support the developer's natural reading flow, which is in most cases left to right. Peters (2008) analyses how patterns in the semantics can be transformed to patterns in the syntax, the layout of Statecharts, to improve readability and comprehensibility. A high potential is seen in this approach.

At present there is not much work available regarding a proper layout of Statecharts, as this is a comparatively young area of research. One investigation has been presented by Prochnow and von Hanxleden (2006), introducing a Statechart Normal Form (SNF).

However, as the components of a Statechart can be seen as parts of a graph, standard graph layout algorithms can be applied to Statecharts. Problems occur when rendering hierarchical Statecharts and care must be taken to display the labels correctly, as a work of Kloss (2005) illustrates. This method is in most cases just a workaround, high efforts have to be made to comply to aesthetic and semantics

standards, as most graph drawing algorithms are not optimized or suited for state diagrams.

An overview of visualization of Statecharts is given in Prochnow and von Hanxleden (2004). There is a significant body of work regarding graph drawing and layouting. Jünger and Mutzel (2003), Kaufmann and Wagner (2007) and Tollis et al. (1999) give a good introduction in their books. Prochnow and von Hanxleden (2004) give an overview of visualization of reactive systems in an annotated bibliography. Section 3.2 elaborates the layout of Statecharts.

Summary

On the previous pages a short introduction to model-based system design was given, as this software engineering technique serves as the application area of Statecharts and hence for layout and visual comparison. An overview of Statechart syntax and semantics was given, to the extend appropriate for this thesis. The general relevance of a proper layout was motivated and some open issues were already mentioned. The next chapter sets the stage for this thesis covering state of the art development tools.

Viðrar vel til loftárása.

Good weather for airstrikes.

Sigur Rós - Ágætis byrjun

3

State of the Art in Model Comparison and Layout

This chapter gives an overview of current techniques for model comparison and the layout of diagrams. After the introduction in modeling, Statecharts and layout in the previous chapter, these foundations are applied to state of the art software development tools. A set of comparison schemes, layout algorithms and tools will be identified to undergo a deeper inspection. Section 3.1 deals with model comparison, in the Section 3.2 layouting will be evaluated, always keeping the context of Statecharts in mind. Then, though limited due to the lack of a bigger choice, in Section 3.3 visualization of structural differences is analyzed. The chapter closes with a section sketching IDEs in use and a last section stating the benefit of the possibility of a visual comparison in contrast to the facilities available now.

The problem to be solved, the visual diff problem, is a complex one that touches several distinct topics of computer science. Therefore, it is split into the above mentioned constituting subproblems. This is convenient, as a complex problem can be reduced to smaller ones. A drawback is of course that if solutions to the smaller parts are found, they have to be rejoined into one procedure.

3.1. Model comparison schemes

This section describes some methods and approaches to compare models, in contrary to compare simple textual files, and elaborates on the synopsis presented in Section 1.1. The comparison of models is one of the two foundations needed for this thesis, to visualize differences in Statecharts.

The need for the comparison of models is manifold and is growing recently and constantly, as modeling itself is on the rise. However, the basic motivation for an

appropriate mechanism to get the differences of two (or more) models remains the same like the reason for textual files or code. When one or more users work with software projects and change the files as a natural consequence of the development process, it is important to see what changes occurred, especially within groups of developers. Another need for a sophisticated difference method is the updating and merging of files that were changed simultaneously by more than one editor. Today's tools are quite ingenious in this field.

Another application of a general model comparison could be to identify stable and unstable parts of a larger software project, to be able to see on which part the developers should concentrate. Within this method of comparison the emphasis does not lie on single files, but rather on directories and subprojects, and the method to achieve this would be quite trivial.

When talking about models, as described in Section 2.1, another question arises. The issue of the visual representation of the model and the implication thereof concerning the status in matters of modification. Specifically, will a model be regarded as changed when just the representation—the drawing of it—has changed while the underlying semantic model remains the same? This is an important question, since more and more models are created and edited by a graphical editor. Many of those editors allow *freehand editing*, which is to leave the positioning of elements to the user.

In many tools, for example Rational Rose Realtime and the back-end versioning system, the layout information is stored within the model. Every tiny movement of graphical elements is considered as a change, resulting in a undesired differences output. The tendency yielding from this behavior is to leave out layout information completely and limit difference computation to the pure semantic model.

However, there are use cases where users might find it helpful to get informed about the changes in the layout. This relates to the issues of secondary notation and aesthetics presented in the next section. Imagine the situation where one developer re-arranged some elements to gain a better understanding of the diagram, while nothing in the behavior was changed. It can be useful to report such changes to other users.

An even smarter system to find out model differences would also try to figure out how possible changes affect the behavior and display that to a user. In the Statecharts case, for example, the same behavior could be modeled in different, more or less understandable, ways. Applying a model transformation, resulting in different elements used in the model, but in the same behavior, should be detected as such: a simplification of the syntax, while remaining constant in the semantics. The model comparison algorithm would have to be extended by such a checker, which always depends of the concrete semantics, leading to a non-generic approach. An extension providing this possibility should be easier to deploy when working with structured models, than in the textual case with source code.

Expanding on this thought, it is also an issue how intelligent the differentiation engine should and can be. The more accurate the result is supposed to be, the more has to be known about the semantics of the objects to check against. An engine

which, for example, knows that the order of elements in a list affects the meaning, can depict this when producing the output of the changes. This is the granularity of the comparison.

There are several methods to detect changes. The easiest one is to use unique IDs for each element in the model. The advantage is that the matching will be completely accurate. A downside is, first, for every element there is the need for an ID, leading to a storage overhead. Second, if a user deletes a model element in an editor and decides afterwards that that was a bad idea, the re-added, semantically and syntactically equal element, will have assigned a different ID from the editor. A comparison will fail on these entities.

The general approach to synthesize the changes out of a model is to take both models and perform the analysis. An example for UML diagrams can be found in Girschick (2006). This standard procedure is called *offline* differences detection. An *online* variant would be to monitor the changes to a model in a separate file or in a database like in CoObRA. This is much more reliable and enables also a complete undo history, but the objects to compare have to be under this version management system from the very beginning. A matching of two independently developed models is not possible.

3.1.1. EMF Compare

EMF Compare of Brun (2007, 2008) and Toulmé (2006, 2007) aims to serve as the framework for comparison of model data in the Eclipse environment, which is explained in Section 5.1. It is implemented as an Eclipse plug-in and integrates well with the Eclipse UI and the existent compare facilities. In a normal Eclipse installation all files are compared with a standard textual engine. For Java source files there is support to compare them on a higher level, identifying nested classes and function definitions, and showing this in a structured manner to the user. After the installation of the EMF Compare plug-in, the file extensions of the models the users wishes to compare have to be registered. Having done this, every comparison command issued on such files opens the EMF Compare panel instead of the normal textual window.

EMF Compare goes one step further than the Java source code method and supports models that were—as expected by the name—created in EMF. So comparison of any of these models comes for free, and the results are, though being a generic approach for any model, quite satisfying, as one can see in Figure 3.1. However, due to the excellent plug-in support of Eclipse, it is possible to add extensions to address the needs of special models better.

Options which can be adjusted are the treatment of element IDs, which can be turned on or off. If turned on, elements are just considered equal if they have the same ID. Support is also offered to copy changes from one model to the other, to merge changes and to compare two models against a common ancestor.

Works of Xing and Stroulia (2005, 2007) inspired the developers to implement EMF Compare. So EMF Compare serves also as a reference implementation for the

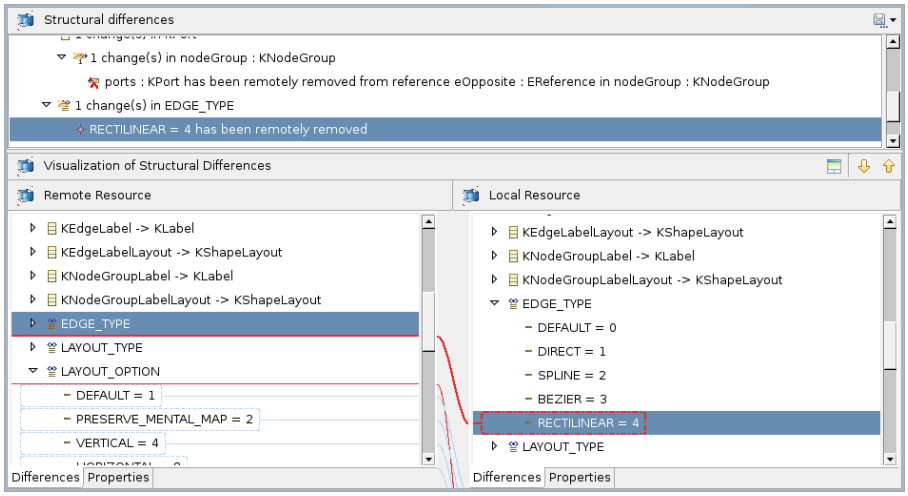


Figure 3.1.: EMF Compare

methods described in the respective papers. The overall algorithm uses two main steps, a *matching phase* and a *differentiating phase*. In the first stage, the two models are browsed and matching elements are identified. The matching engine uses several metrics to declare elements matching, those are, in the following order:

1. *Type similarity:* analyzing the respective metamodel element

2. *Name similarity:* searching an attribute which could be the name and comparing it

3. *Value similarity:* analyzing the whole attributes values

4. *Relations similarity:* gaining information from the relations the element has with others

This yields good results when for example elements have been deleted and added again, or when just the order of elements in the textual model file was exchanged, but this order does not imply any semantic change. If the order should matter, it must be modeled in the EMF model itself, and then a possible change would be detected. In this first step the *matching model* is created, being the superset of the two models. Common elements to both models just appear once in this model, everything that could not be matched is also added.

In the second step a *difference engine* walks through the match model, computing changes in terms of *added*, *deleted*, or *updated* (comprising *moved*) for each of the two input models. Full support for attributes and references in the model is given. An overview of this process is given in Figure 3.2. Having obtained the difference model, it is possible to merge the changes and to export the differences.

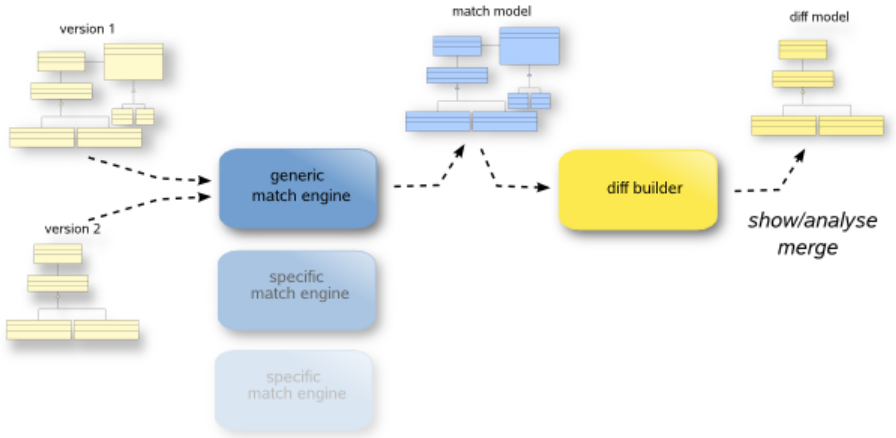


Figure 3.2.: Schema of the EMF Compare process (EMF Compare Team, 2008).

3.1.2. SiDiff

SiDiff, developed by Schmidt (2007); Schmidt and Glötzner (2008); Treude et al. (2007); Wenzel and Kelter (2006), goes a similar way like EMF Compare. It can compare arbitrary models, which have to be written in an XML-based file representation. These input models are transformed using Extensible Stylesheet Language Transformations (XSLT) to an internal graph-like data structure.

The main contribution of that work is the time this implementation consumes to compare models. Whereas prior implementations were in the order of magnitude of $O(n^2)$, resulting from the pairwise matching attempts of elements, SiDiff gets along with $O(n \log n)$. To achieve this, a special tree structure is used internally. The steps performed are:

1. *Hashing phase:* All elements are hashed. Same hash values lead to an immediate recognition of equality and those elements are left out in further stages.

2. *Indexing phase:* Special S^3V trees are created to support the search for similar elements. In those multi-dimensional trees, elements with a high probability to be equal are arranged closely, in terms of the Euclidean distance, to each other.

3. *Matching phase:* The matching phase is similar to EMF Compare, and the final differences are also computed as in the latter tool.

The output depicts the operations that have been applied to one model to obtain the other. SiDiff displays these operations as *attribute change*, *reference change*, *move*, and *structural change*. With this output a custom tool can be fed to further process this information and display the changes or invoke a merge mechanism.

3.1.3. CoObRA

The Concurrent Object Replication frAmework, developed at the universities of Kassel and Siegen by Schneider et al. (2004), focuses on the monitoring of changes during the editing of the model. This differs from the approaches of the previous projects, where these changes first had to be computed. The advantage is the complete and correct catalogue of changes for a model and the fully reversible history. CoObRA is implemented in Java and freely available for download. A realization can be found in the tool FUJABA. CoObRA uses an own repository with an optimistic locking mechanism to monitor the changes to model. The problem of merging different versions has now been relocated from merging of existing models to the merging of editing steps. Some effort is needed to get this done.

To enable the monitoring mechanism, the objects that should be put under version control have to implement a special interface provided by CoObRA or extend a special adapter class. The objects are now aware of changes to themselves and announce them as *creation* and *removal* (of the whole object), *altering* of a field value and *adding/removing* of a value to a collection.

The developers observe that computing the changes does not produce much overhead and that therefore the implementation scales well. Working with the objects locally without access to the repository is also possible, due to their optimistic locking concept, which can feed the changes later into the central storage. There is also a second version of this software available, though unfortunately none of the versions is actively developed anymore.

3.1.4. Pounamu

Pounamu[1] is a meta-CASE tool developed by Zhu et al. (2007) in Auckland. This tool does not perform the actual comparison of models, but a nameless add-on thereof introduced by Mehra et al. (2005). For this reason, the headline is Pounamu. This plug-in does not only support comparison of models and output of the differences, but also an annotation of these changes in the model's diagram itself.

The plug-in concentrates on diagram comparison. Each diagram the users are working on is checked into a central repository, the file format of choice is XML. The versioning system is analog to SVN or CVS. When a user checks out an updated model which is newer than his or her local copy, the compare process is started locally in the application. The two versions of the model are transformed, like in *SiDiff*, into an internal Java graph. A distinction is made between diagram data responsible for the view and domain data describing the actual model.

An algorithm similar to the ones presented before tries to find matches in the two passed models. Notable is that this implementation works with unique *root* IDs for each type of element, and with object IDs for each object created. The algorithm also just works with models generated in the meta tool Pounamu, which can generally be

[1] Pounamu is the Māori name for several hard, durable, and highly valued types of greenstone (jade).

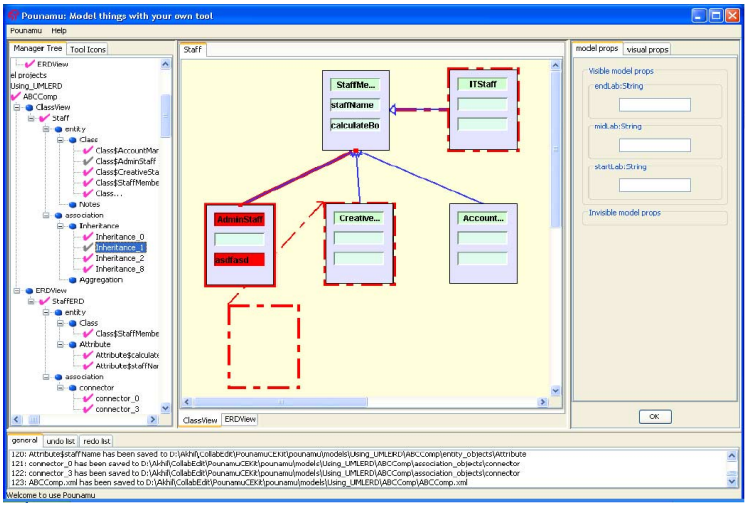


Figure 3.3.: Diagram comparison within Pounamu (Mehra et al., 2005).

regarded similar to diagram editor generators like GMF (see Subsection 5.1.2). From the match and differences model the changes are derived in terms of commands, also a well-known and used technique so far.

With this information the view of the diagram is updated. Due to the separation of view and domain data, it is clearly depictable if an element was just moved around or if it was really altered semantically. The Figure 3.3 gives an example of an annotated diagram with changes.

3.2. Layout algorithms

In this section the techniques for laying out graphs are described and a first rating is done. As already addressed in Section 2.3, good layout is a key factor for the correct understanding of Statecharts. For the rendering of Statecharts, in KIEL, for example, general algorithms for graph drawing are used (Castelló et al., 2002; Gansner et al., 1993). They are described in the first subsection. The next subsection bridges the gap between graph and Statechart layout, and the last subsection presents some graph drawing tools.

3.2.1. General layout methods

The following subsections present several types of layout algorithms. For another short overview refer to the work of Kloss (2005), for a more detailed explanation see Tollis et al. (1999). See North et al. (2003) for a comparison of spring-based, hierarchical, orthogonal and other types of layouts. For most of the described algorithms, and for most algorithms in general, a planar graph is needed, which is a graph with no edge crossings. If the graph to be displayed does not satisfy this constraint, there are

several techniques to make the graph planar (Kaufmann and Wagner, 2001, p. 29). Normally, there are some shadow nodes inserted before the actual layout, and after the calculation of the positions they are removed again, implying a postprocessing of the edges.

Hierarchical

Hierarchical layout, often also called *layered*, arranges edges of directed graphs in hierarchical layers, depending on a predecessor and successor relationship defined by the directed edges of the graph. This type of layout is—after the work of Sugiyama et al. (1981)—also known as *Sugiyama*-style. Previously only capable of directed, acyclic graphs, it is now possible to lay out any type of graph, using techniques such as re-orienting of edges or by adding shadow nodes. The steps performed to achieve a layered layout are basically as follows:

1. Map each node to a layer, using the successor relationship resulting from the direction of the edges.
2. Readjust the edges of each layer to minimize edge crossings.
3. Apply coordinates to each node, try to distribute the nodes of each layer in an appealing manner.

A well-known implementation of this algorithm can be found in the GraphViz suite, specifically the Dot program contained in it (Gansner et al., 1993). As Statecharts have inherently directed edges—the transitions—a hierarchical algorithm applied on Statecharts yields quite good layouts. This is evaluated by Kloss (2005), where he finally chooses GraphViz Dot to lay out Statecharts. Because of these results and own experiences, as well as of results of Prochnow (2008); Völcker (2008), the Dot algorithm was chosen as the default layout back-end for the implementation. An application can be seen in Figure 3.4(d).

Orthogonal

Orthogonal layout distinguishes itself by an orthogonal drawing of the edges. Therefore it is often called *Manhattan-routing*. It is based on the *Topology-Shape-Metrics approach*, as described in Tamassia et al. (1988). The three elements *topology*, *shape* and *metrics* can be found in the three steps executed in the algorithm:

1. *Topology:* The topology of the nodes is computed. That is the distribution of the nodes of the graphs, while minimizing the crossings.
2. *Shape:* The edges are orthogonalized, trying to gain the fewest bends, yielding the shape of the graph.
3. *Metrics:* The exact positions of the nodes and edges are calculated, while trying to minimize the space used.

A popular application of the orthogonal layout is in UML diagrams and in Printed Circuit Boards (PCBs), an example is given in Figure 3.4(e).

Radial

Radial drawing of graphs is a special case of displaying rooted trees. As most trees can also be seen as a layered representation, namely considering all the nodes with the same distance from the root as in the same layer, it is similar to the layered or hierarchical method described above.

A radial drawing takes a tree and places all the nodes on perimeters of concentric circles, dependent on the distance of the root node, which is placed in the center. An example can be seen in Figure 3.4(a). Several means are available to choose a root node from arbitrary graphs.

Radial views of graphs are used to illustrate relationships in social networks, with the entity of most interest drawn in the the center, or other application areas where certain dependencies should be depicted.

Circular

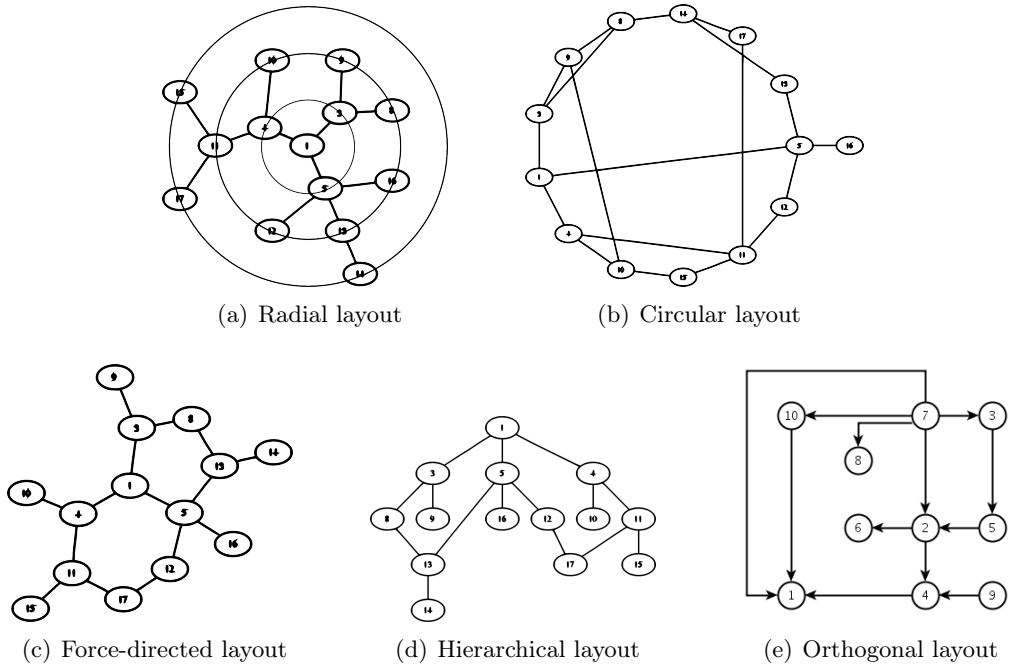
Circular graph layout is a drawing scheme where, as the name already says, all nodes are placed on the perimeter of one or more circles. In contrast to the aforementioned radial method, circular algorithms try to place the nodes in that way that the (or a possible) flow of the graph is represented by the way the nodes are aligned on the circle.

In Figure 3.4(b), when comparing to the radial layout, this can be easily seen. Notable is also the absence of a centered root node. A drawback of this method is the great number of edge crossings, resulting from the convention to draw straight lines between the nodes. That results in the fact that all connections between not neighbored nodes are routed through the interior of the circle.

The circular method is only suitable for small sets of graphs, but can be useful in some cases. A representation of a loop with several single steps can be much clearer if using a uniform circle to display it (Peters, 2008, p. 39).

Force-directed

Using this method, the graph is transformed into a physical system. Nodes are seen as particles repelling each other, edges are modeled as springs contracting themselves. An algorithm computes a state where all the forces in this system are in equilibrium. This can often be parameterized regarding the duration of the computation. The more computation time the algorithm is granted, the smoother the layout will be. If there is just limited computation time, and the desired quality of the rendering is known beforehand, this algorithm can be a good choice. Force directed layout is often used to model the relations of entities in a network, physical as well as social ones. An example can be seen in Figure Figure 3.4(c).



(a) Radial layout (b) Circular layout

(c) Force-directed layout (d) Hierarchical layout (e) Orthogonal layout

Figure 3.4.: Some basic layout methods

Dynamic layout

The term dynamic layout does not refer to a class of layout algorithms as the ones listed before, but rather to how these drawings change over time. It is listed here as it is of interest when working with diagrams that change over time, such as Statecharts, and the user has built up a mental-map of the layout and its inherent secondary notation.

It is desired to maintain the rough positions of the elements of the drawing over time, even when elements are deleted or new elements are inserted. Those constraints impose high demands on the algorithm and are explained in works of Brandes and Wagner (1997) and Branke (2001).

3.2.2. Layout of Statecharts

All those above presented layout algorithms were originally developed to lay out graphs, consisting of nodes and edges. Though graphs are structurally similar to Statecharts, there are some relevant differences. Some of the main disparities are:

- Nodes in graphs have no or at least in most cases a small, fixed size, states in Statecharts vary in size according to labeling and hierarchy.
- Most standard graphs have no labels applied to the edges, transitions in Statecharts can have multiple labels. An example are priorities drawn at the tail

and triggers and effects at the center of a transition.

- Standard graphs do not use hierarchy, whereas in certain Statechart dialects hierarchy and even inter-level transitions are used.

- The problem instances for graph drawing software and algorithms are often quite huge, comprising up to thousands of nodes and even more edges, whereas Statecharts usually contain typically up to 20 states per hierarchy level.

These issues are addressed in works of Castelló et al. (2002), in the cases of the varying size of states. The algorithms found in the GraphViz suite also work with user defined sizes of the nodes. According to the given size or a size resulting of the length of a node label, just the positions of the nodes are computed, taking the connection of the nodes via the edges into account.

Research considering labels is done by Castelló et al. (2002); Kakoulis and Tollis (1997). The labeling itself is separated into Node Label Placement (NLP) and Edge Label Placement (ELP). In most cases, this applies in general to the Statechart cases, the NLP comes for free, as the node labels are placed inside the state. Very often algorithms considering ELP start with an already laid out graph and then try to place the labels in a convenient way. This raises problems when working with dense graphs. The ELP of Dot, for example, takes into account the size of edge labels beforehand.

Treating of hierarchy in graphs is examined by Harel and Yashchin (2002). Inter-level transitions are still an issue, a proposal are *goto labels*, but those disturb the inherent flow of a Statechart. The Dot algorithm can handle some sort of hierarchy, called subgraphs, and even inter-level edges are possible, but in the implementation done for this thesis another approach is used, which also resulted from various other reasons. This is described in detail in Section 5.3.

Considering the above explanations, and following Kloss (2005), GraphViz, and especially Dot with its *Sugiyama* algorithm, seems best suited for use as the back-end algorithm to lay out Statecharts in KIELER.

Secondary notation

When reasoning about layout—or the representation—of Statecharts, the popular term secondary notation (Petre, 1995; Shu, 1989) has to be taken into account. While Statecharts, and other graphical languages, were very early hyped as some *ultima ratio* in the design of complex systems, soon some disillusion got the upper hand, resulting from different concepts of notation and representation of semantics. So the demand of good graphics is not only of aesthetic nature; the representation should aid the developer in understanding with its secondary notation.

Secondary notation is referred to as information transported not in the notation itself, but as the information in the way the notational elements are used. Notational elements of Statecharts are, for example, different types of states reflecting a certain semantic equivalent; a possible application of secondary notation can be a convention

to draw initial states on the left to reflect the natural flow of reading[2] of a developer in the charts itself. That helps readers to grasp at a glance some overall behavior of the modeled system. The problem is that there is no, and probably cannot be, a convention about the *one* secondary notation (Prochnow and von Hanxleden, 2004). However, some advice concerning position, size, color, and labeling can be given for a meaningful application of secondary notation (Green and Petre, 1996).

Another approach to abstract secondary notation is examined by Peters (2008). She proposes a pattern recognition mechanism for Statecharts, identifying commonly used patterns and using a determined graphical representation for them. She credits this approach with a high potential.

All issues concerning secondary notation are tightly connected with the drawing of the chart and therefore with the algorithm responsible for it. Standard graph drawing algorithms might meet their limits because of the high number of constraints introduced. Keeping the secondary notation in mind helps to choose the appropriate layout algorithms for distinct parts of a diagram.

Aesthetics

Closely connected to secondary notation, but mostly concerning syntax, though also touching semantics, aesthetics is an important field in the area of Statechart layout. A work concerning aesthetics in the UML environment is from Ambler (2005). Some aspects can also by applied to diagrams like Statecharts. A first approach to gain criteria for Statecharts is to adapt graph drawing aesthetics. Main building blocks thereof are:

- Node distance,
- Nodes should be placed on a grid,
- Nodes should be placed symmetrically,
- Edge length,
- Edges should not intersect each other,
- Avoid bends in edges, and
- Label placement.

A diploma thesis of Völcker (2008), based upon preliminary work of Chok et al. (1999); Prochnow and von Hanxleden (2006), solely deals with the aesthetics of Statecharts and how it could be parameterized. To incorporate their idiosyncrasies, some points had to be altered and new ones had to be introduced. Some relevant additions were:

- Minimum distance between states and transitions.

[2] At least for people using a language that is written left-to-right.

- Placement of the initial state.
- Placement of the final state.
- Place labels near source states.
- Place labels on the basis of transition direction.

Other criteria like symmetry were not considered that important and as with all collections of criteria, there are trade-offs between them, making it hard to find out the ultimate formula, already discovered by Purchase (2002).

As with the topic secondary notation, the demands for aesthetics in Statecharts drawing reflect themselves in these for the layout algorithm. Simply said, an aesthetic Statechart is better to understand, and that helps when changes in two of them should be explained to a user.

Editing Statecharts

Statecharts are well-known to be able to present complex systems to a developer the way he or she can understand the semantics of this system very well and fast. Problems can arise during the initial creation, and moreover during later editing steps, when rearranging of states is needed. This problem of finding new space for added elements and moving remaining elements to address aesthetics and comprehension issues is covered in Prochnow (2008) and was also a reason for the set back in graphical modeling described above.

The standard way of editing Statecharts is called *freehand editing*, allowing, but also forcing, the user to move the Statecharts at his or her own will. By contrast, some works propose *structural editing* of Statecharts (Prochnow and von Hanxleden, 2007). The user is not allowed to alter anything of the layout, but creation and altering of Statecharts is done by structural commands. The user requests to add or delete an element, the changes are reflected immediately in the graphical representation and a new layout is drawn automatically, using animation, to relieve the developer from the *pain*[3] of freehand editing. This is an advantage as well as a possible drawback, as the user is forced to accept the layout calculated by the layout algorithm, though those settings can be adjusted. Another aspect is that a potential mental map the user built can be disturbed by the new positions the algorithm proposes.

Structural editing has been credited with a high potential recently and the resulting assumption of an already sound layout will help later with the visual comparison.

3.2.3. Graph drawing tools

The following subsections present some graph drawing tools. This is of course just a small selection of the great variety of programs available. A more elaborated list can

[3] "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". A developer, according to Petre (1995).

be found in the books of Kaufmann and Wagner (2001, pp. 274) and Jünger and Mutzel (2003). Kolovos et al. (2008) state that a satisfying support for (automatic) layout is still missing in most commercial development environments.

GraphViz and DynaGraph

GraphViz (GraphViz, 2007) and Dynagraph (Ellson et al., 2003) are two popular software collections for the visualization of graphs. Though they are not developed together, they are listed and examined jointly, as some developers worked in both projects and Dynagraph was inspired by GraphViz. On this account they share the same language to describe their input graphs, the dot-language (Gansner et al., 1993).

With these two frameworks, the names say it all. GraphViz visualizes graphs, rendering the input with some of the algorithms listed in Section 3.2. Using techniques to reorientate directed edges, directed, undirected, acyclic, and cyclic graphs are supported. Dynagraph extends this concept by the possibility to draw diagrams dynamically. The term *dynamically* or *incremental* refers to the fact that the layout engine tries to position new elements in that way that the mental map of the user is not disturbed. An example was already given with Figure 1.1. New elements are added with commands forcing the engine to calculate a smooth new layout.

Kloss (2005) identified GraphViz as best suited for the layout of Statecharts. That was not because of the fact that GraphViz, especially the Dot program implementing the hierarchical[4] *Sugiyama* algorithm, does a perfect job, but there were simply no better free libraries available.

One drawback is, as already mentioned, the lack of hierarchy. It is possible to draw clusters, but the results when applied to Statecharts are limited. Positive features are the ability to set the node size and shape freely, that two labels for edges are supported and that these can be forced to be laid out at the tail and at the center of a transition, fitting to the standard labeling of Statechart transitions with priority and trigger/effect.

GraphViz can output the rendered graphs in various formats, textual formats with position and size attributes are supported as well as image data, for example. Dynagraph just produces an output that describes the topology of the graph enriched with size and position information and needs a compatible viewer or editor to display the actual diagram.

Both projects are open source and available for several platforms. As they are written in C/C++, for the linking to the Eclipse environment some effort was needed. That was also done by Kloss (2005).

[4]Keep in mind that hierarchical in this context means hierarchical or layered drawings of nodes within one level of hierarchy, not a hierarchically structured graph itself.

GUESS

GUESS (Adar, 2006), short for Graph Exploration System, is a data analysis and exploration tool for graphs. Exploration and analysis refer to the fact that large graphs showing complex relations, such as network structures or sample data, are intended to be visualized. Through an interactive interface, zooming and scrolling the graphs on-line is possible. A command line interpreter enables querying the structure of the graph as well as issuing several edit commandos to the model, being reflected directly in the visual representation.

The application is written in Jython, the Python interpreter in Java and uses an own language called Gython to manipulate the graph. Gython, an extension of Python, enables the user through its syntactic sugar to work with the graph easily. Several layout algorithms are implemented which yield good results, and with a short response time. For an example of a spring embedder layout see Figure 3.5.

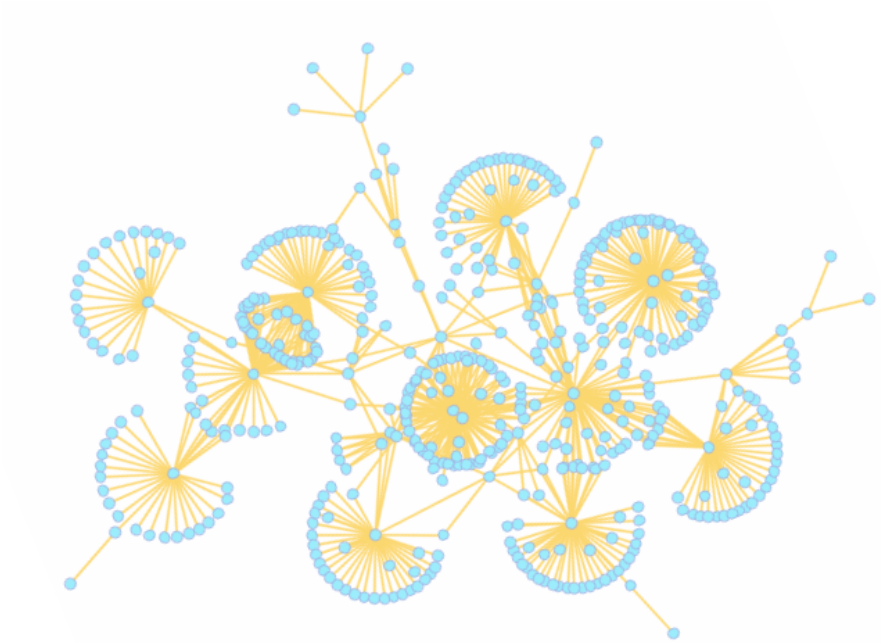


Figure 3.5.: Spring layout produced by GUESS

GUESS is available as open source software and comes with the GPL licence. Many other open source software projects are used by *GUESS* and are shipped with the same licence. Graph structures can be read in from the GraphML and Pajek[5] format. Normal files or even a SQL connection serve as storing back-end. A direct link to an R[6]-server is implemented and actively improved.

[5]Program for Large Network Analysis, see http://pajek.imfm.si/.
[6]Statistical software package.

yFiles

yFiles of yWorks (2005) is a Java class library also implementing several of the above mentioned algorithms. It shares several features with the GraphViz collection such as customizable sizes and positions. Incremental layout algorithms are also available. The main advantage for a deployment in Eclipse—that it is implemented in Java—is outweighed by the fact that it is not freely available.

yEd, shown in Figure 3.6, is the corresponding editor to display and edit graphs that are rendered with yFiles. The editor comes for free and uses the binaries of yFiles. In Wiese et al. (2003) quite impressive results show the power of yFiles. Tools from Gentleware AG, like *Poseidon for UML*, use yFiles and exploit also the incremental layout facilities.

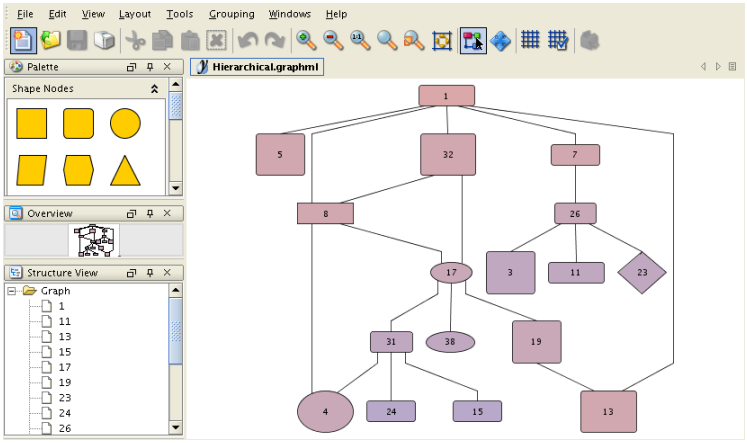


Figure 3.6.: Hierarchical layout in yEd.

DiaGen/DiaMeta

DiaGen and DiaMeta are software systems to generate powerful diagram editors, comparable to the Graphical Modeling Framework (GMF) in the Eclipse world, explained in Subsection 5.1.2. DiaGen uses thereby a grammar describing a model to *generate* an editor for this domain, DiaMeta uses *meta* models to specify the input for the model generator. The current reference implementation even uses the EMF for this purpose. Creating model editors this way is quite similar to the Graphical Modeling Framework approach in Subsection 5.1.2.

Due to this approach, there is of course no layouting capability given. But the developers show in several papers (Maier and Minas, 2007a,b) how easy it is to write general and specific layouters for editors for a special DSL. They have implemented dynamic, static, and pattern based layouters. Even an example of a layout engine for Statecharts is presented (Maier and Minas, 2008), which can be seen in Figure 3.7. They also decided to use the *Sugiyama* algorithm.

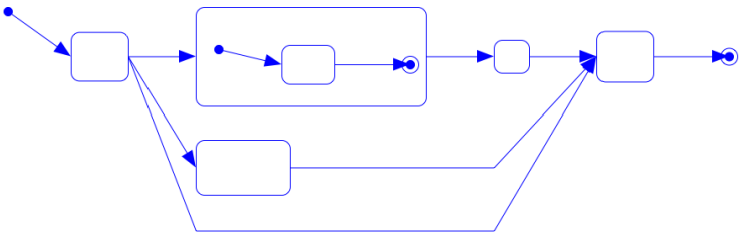


Figure 3.7.: Laid out Statechart in DiaMeta (Maier and Minas, 2008).

3.3. Visualization of structural differences

This section gives a short overview of how differences can be visualized. It is not limited to an output in diagram form, but to a general visualization. The main difference from a normal textual difference view is the structure of the document that is considered when performing the relation of the two objects.

The normal, plain diff is well-known, but does not work well on structured documents. The line-by-line comparison does not scale for objects and sub objects inside a model, which should be compared according to those semantical information.

First implementations emanated from the XML content of model files and used the XML elements as a base for structural comparison. The representation was, as for example in EMF Compare (see Subsection 3.1.1), in a tree structure. Applications working with this paradigm are *The Compare Utility* (Spark Systems, 2008) or the *XML Differencing tool* (Stylus Studio, 2008).

Girschick (2006) presents in his paper an algorithm to detect and display changes in UML class diagrams. To achieve better results, this algorithm is dedicated to UML class diagrams exclusively. Reports of changes are shown in a HTML page, including a Scalable Vector Graphics (SVG) view of the merged diagram and a textual description of the changes, as illustrated in Figure 3.8. The view of the diagram is a merged version of both, which can lead to unesthetic overlappings, when much has been changed between the two versions. Ideas of Ohst et al. (2003a,b) were picked up for this work.

Another example is the plug-in for Pounamu, already described in Subsection 3.1.4, using a more generic approach that is not limited to one kind of model. The method to display changes is more 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.

It can be noted that by now there are quite sophisticated solutions to perform a differentiation on structured entities. However, there is no useful research how these differences are presented best to the user, especially not when they are supposed to appear in the diagram itself. There are many works considering graphical languages,

perception and respresentaion, correlation between syntax and semantics, secondary notation, and many other things, but a sound solution for a graphical comparison has still to be found, if there is *the* correct one at all.

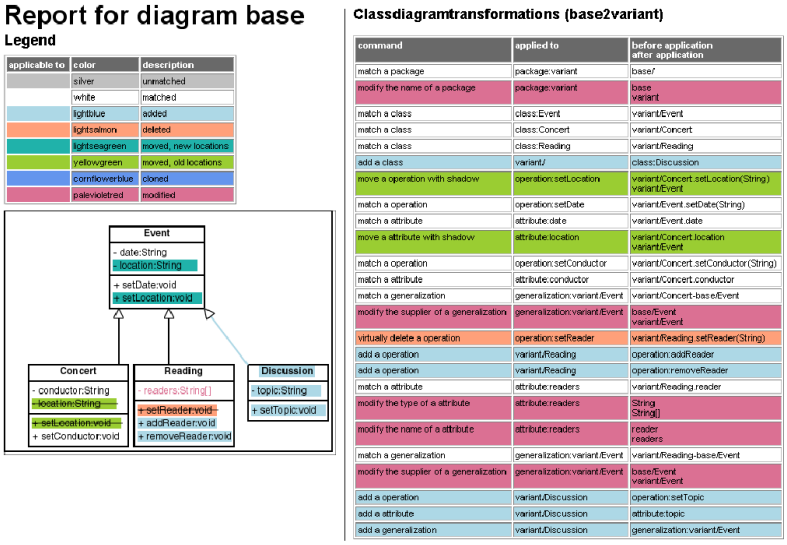


Figure 3.8.: HTML report of changes including the diagram, generated by *UMLDiff*$_{cld}$ (Girschick, 2006).

3.4. Implementation and drawbacks in today's modeling tools

Though there are some promising ideas presented in the previous section, there are not many implementations available on the market. Actually there were just two tools which could perform a real visual comparison of models, in the sense as meant by this thesis. One aspect with this is also the nomenclature. The terms *graphical*, *visual* and both terms in arbitrary compostions are used rather light-handedly. In this context, when talking of diagrams and graphical models, the term *visual* comparison of course refers to a display of the differences in the graphical model itself. The label *graphical* would also fit.

As every comparison that is presented to a user is inherently visual, there is normally no need to denote it like that. So, when companies use one of those words, they usually mean a display of differences where they use colors to highlight the changes, but limit it to some textual comparison, as known from standard textual diff tools widely available.

ecDIFF

One tool, ecDIFF, developed by ExpertControl GmbH (2008) and National Instruments, "*is a graphical differencing tool for Simulink models*"[7]. As Simulink models are, among other things, characterized by their graphical notation, one would normally expect that this term denotes a presentation of the differences in some diagrams themselves. In the standard setting, that is not the fact.

When comparing two models, the result is presented to the user as shown in Figure 3.9. Both models are presented in a tree viewer and the changes are displayed in a third sub-window at the bottom. When now clicking on a highlighted element, the respective diagram part is opened to help the developer to find out where the changes occurred. According to the datasheet, it should also be possible to see the differences by flashing the changed boxes in user-definable colors.

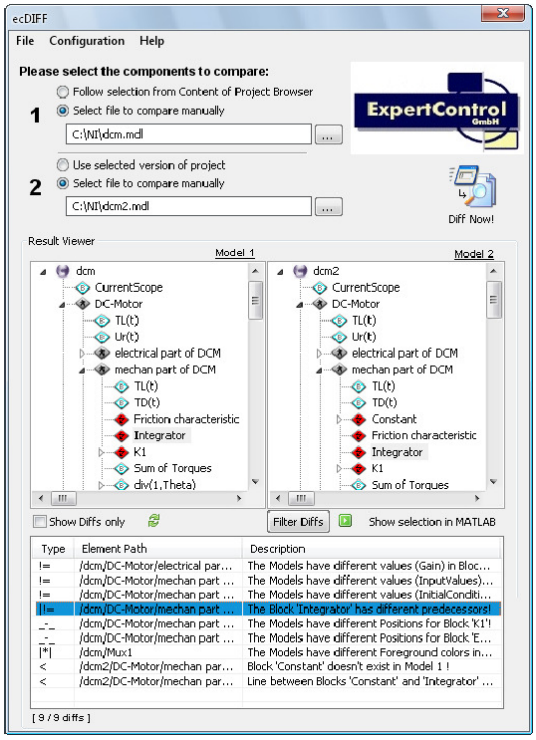


Figure 3.9.: Differencing of two Simulink models with ecDIFF. Source: ExpertControl GmbH (2008) homepage.

SCADE Model Diff

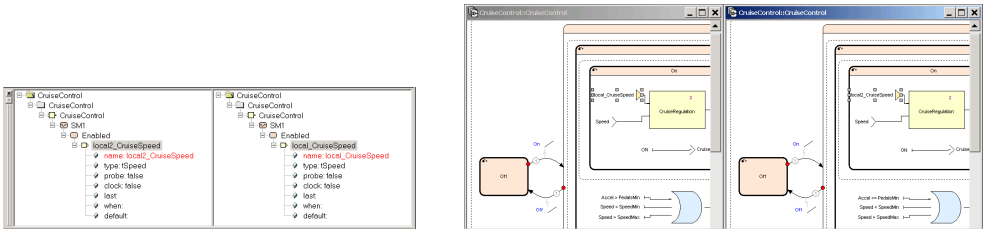
SCADE (Esterel Technologies, Inc, 2008) is a software development suite for safety-critical applications. *SCADE* comes with a graphical editor making use of a data

[7] Taken out of the ecDIFF datasheet, available at the homepage.

flow language and SSMs. Version 6.0 of the *SCADE* suite contains now the *SCADE Model Diff*, an add-on to the suite which can display differences of *SCADE* models in the diagrams themselves. The *SCADE* suite itself is capable of comparing models, the add-on serves as the link to the graphical world. Models are compared based on their semantics, leaving out any position information of the layout. Changes are denoted as *added*, *deleted*, *changed* and *moved*. These results are presented textually in a tab, using different symbols to visualize the type of the change.

Having selected a changed object in the diff tab, another window with a tree structure opens, illustrated in Figure 3.10(a), confronting the two versions of the model, similar to ecDIFF and EMF Compare. Another click in the tree viewer finally opens the diagrams, the elements under observance are highlighted with the standard selection marker, as can be seen in Figure 3.10(b).

Furthermore it is possible to generate a structured report of the changes that occurred. But that is just possible in a textual way, structured similar to the tree viewer, which is anyway better than a graphical report.



(a) Tree window (b) Locating changed objects

Figure 3.10.: *SCADE model diff* windows (Esterel Technologies, Inc, 2008).

Drawbacks

Apart from the pure lack of applications doing the visual comparison task well or at all, there are also some drawbacks in the available systems, which lead to this thesis. First of all, the current systems are not very adaptable to user preferences. Normally one just would not comprise the layout information in the comparison. The strict separation of domain and notation data as in *SCADE Model Diff* is a great advantage.

Second, the user handling is in both cases not satisfying. The developer has to click through the application to see the changes he or she desires, and really needs to perform many operations to do so.

Third, when comparing bigger diagrams, it can be painful to browse all the changes inside them, when there is no further support for this given to the user. This can get worse if the diagrams contain hierarchy, as in Statecharts.

3.5. Benefit of visual comparison to model-based development

That there is a strong need for a real visual comparison can be seen by the two applications listed in the previous section, which were introduced into the market in 2007 and 2008. The demand is high, but due to the new area of the topic there is still a lack of reasonable tools performing this.

Various scientific publications and developers using modeling tools stress the necessity for a visual diff. Ohst et al. (2003b) point out the fact that it is just not reasonable for two dimensional documents like diagrams to display possible changes in the old fashioned way, in two columns with corresponding elements facing each other. In most cases there is a reason why objects in a diagram are positioned like they are, and should not be realigned, at least not in just one dimension.

Another aspect is that changes in diagrams mostly affect some kind of behavior. This is particularly true for Statecharts. Depicting changes in the diagram itself helps the developer working with it to understand the resulting modification in behavior immediately.

Mehra et al. (2005) reason in their paper that graphical comparison is in that way an advantage, as this is the natural way to compare visual objects. The old approach, converting the differences to some structured text, is just a workaround due to the lack of better methods. Transforming the textual displayed changes back to the graphical world requires a "*hard mental operation*" (Green, 1989), which is unnecessary and should be avoided. This should be treated independently from the question if graphical languages are superior to textual ones at all.

The two different levels where the comparison takes place, textual and graphical, relate also to issues with the mental map the user builts up when working with a diagram. A user normally masks out, to some extent, the semantic structure behind graphical models, deriving the meaning from the graphical representation itself. Aligning the changes at the user's mental map helps to understand them considerably.

Summary

This chapter gave an overview of which techniques are currently being used to aid model developers. It started with model comparison and presented some tools. Then some layout algorithms were presented, as well as some tools, and the relation to Statechart layout was pointed out. The next section addressed the visualization of structural differences, afterwards some tools for that purpose were presented. At the end, the need for a *real* visual comparison was stressed and justified.

"`Wir` sagen und `Ich` meinen ist eine von den ausgesuchtesten Kränkungen."

Theodor W. Adorno - Minima Moralia, p. 217

4 An Approach to the Visual-Diff Problem

This chapter introduces the visual-diff problem and presents approaches to solve it. Section 4.1 lays out more in-depth the problem and its variations, emerging from concrete use cases. These will by identified, classified and examined to their feasibility of implementation in Section 4.2. Section 4.3 describes how differences of diagrams can be computed for this thesis, keeping the demands of visualization and development cycles in mind. Section 4.4 describes ways to help the user to understand and grasp the changes easily.

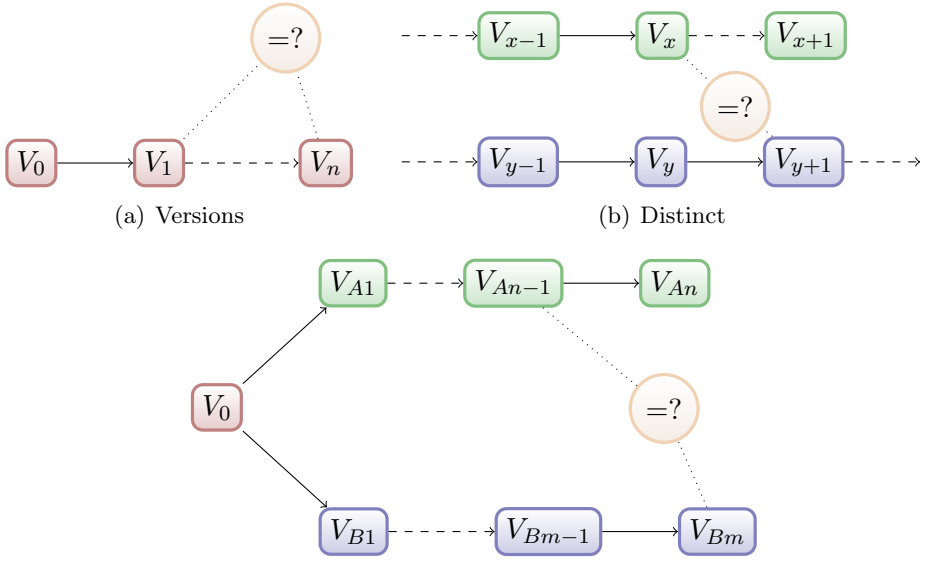
4.1. What and How to Diff

Before starting to extract differences of Statecharts, in order to represent these differences in an adequate manner, it is vital to identify the use cases that need a visual model diff. Another important issue is that also the way how development is done accounts differently to the problem and possible solutions. Take a look for example at standardized processes like the Rational Unified Process (RUP) or the waterfall model. The RUP (IBM, 2007) exhibits an iterative software development process. Applied to Statecharts this would mean a constant change to concrete models, according to the changes in the other phases. There would not be a continuous maturation of the diagram, but many discrete changes in certain windows of time. By contrast, patterns in development like the standard waterfall model make final freezes after each phase, so the development of diagrams following this philosophy is inherently different. There will be just one phase in which someone is working on the Statechart, and most of the time the development will go into one direction.

4.1.1. Scenarios

The comparison problem can been seen from two positions. From the developer's point of view, or from the Statechart's point of view, assuming this has one. One can argue that this is basically the same, but it is more suitable to address the task with the developer's eyes. The user wishes to compare Statecharts in different situations, leading to different requirements. Three main scenarios related to the user can be identified:

1. *Versions*: One developer wishes to compare two versions of the same Statechart she or he is working on.
2. *Distinct*: A developer wishes to compare two different Statecharts which do not have necessarily anything in common or a conjoint history.
3. *Branches*: One or probably more developers wish to compare two (or more) derived versions from a root one.



(a) Versions (b) Distinct

(c) Branches

Figure 4.1.: Different scenarios when comparing models

In Figure 4.1 these scenarios are illustrated to illustrate at which point of the development the comparisons can be made and which Statechart versions they may reference. This leads directly to various proposals on how to display the differences.

4.2. Visualization of Differences

For each of the previously presented scenarios several solutions seem feasible. This section gives a short overview about some of them and tries do to a first rating. The

question of how to actually get the differences on the structural level is left out until the next section.

4.2.1. Layout

A concern is the layout algorithm, which will be used to visualize the differences of two Statecharts. A well done presentation of the changes to the user is highly dependent on a sound layout. Of course, the employment of the layout plays also an important role, that is, for example, how two diagrams are merged or differentiated into one view, if such a method is used. Attention must also be paid to Statecharts that do not possess a specific layout from the outset, thus Statecharts that are freely drawn by the developer with or without any adherence to style guides.

To set at least some foundations for the display of differences, means for layout must exist in the complete framework. That is where the part *layout* in this thesis' title emanates. For certain proposals in Subsection 4.2.3, an already computed, aesthetic, layout is highly useful. This initial layout will be achieved by the implemented layout facilities. That adheres also to the claims of a SNF, where after each editing operation a layouter is called automatically, or where the user is advised to call a layouter to comply the demands of a SNF.

As the title of this chapter deals with the visual-diff problem, layout issues are not discussed here, but addressed in Chapter 5, were the concrete implementation is presented. This is also because Statechart layout has been investigated in some theses before (Kloss, 2005; Prochnow, 2008; Völcker, 2008). What is new is the ability to apply different layout algorithms to different parts of a Statechart, giving more flexibility when facing special needs.

4.2.2. The Mental Map

The mental map also relates to layout and is a topic widely discussed, as for example by Kaufmann and Wiese (2002); Lee et al. (2006); Misue et al. (1995); Purchase et al. (2006). Maintaining a mental map could aid the user when comparing two diagrams with each other, where the same elements are roughly at the same positions. It is therefore a construct worth taking into account when reasoning about means to enable visual comparison and should be remembered for the following proposals.

An actual work using the mental map is presented in Subsection 4.4.1.

4.2.3. Proposals

The classification of possible visualization mechanisms leads to the following taxonomy, which will be explained more explicitly thereafter. The two versions referred to can be selected by the user. This will generally be, but not necessarily, the actual version where he or she is working on and any older one. Where applicable, it will be kept in mind that the target framework to be used is Eclipse, as this affects the choice of existing software. The classification is as follows:

1. *Plain*: The two original layouts are just shown side by side, with colors or similar markers indicating differences.

2. *Animation*: A small animation or video is created, which shows the transition from one version to the other by morphing the Statechart.

3. *Pop-up*: Having enabled the compare mode the user can navigate through the one version of the Statecharts, which is annotated with modifications, and pop-ups will show in detail the changes in contrast to the other version

4. *Static merge*: A merge of the two versions is calculated. This merged model will by laid out from scratch, with (a) colors showing alterations from one release to the next or (b) two derived diagrams from the merged one showing the changes side by side.

5. *Dynamic merge*: Similar to the previous. The calculation of the merge remains the same. The layout is not computed from scratch but one of the original layouts serves as a reference for the merged layout, maintaining the mental map of the developer.

Each of the enumerated approaches will be explained more exactly and with some graphics, where appropriate.

Plain

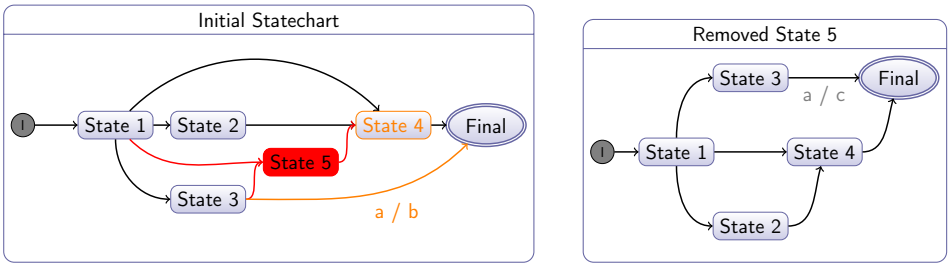
A side by side confrontation, in its static case as described here, is the simplest way of comparing entities. The first thing coming into mind as a prototype for this type of comparison is the ordinary textual diff, enhanced by a graphical representation showing the versions in two columns with corresponding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

There are several advantages in this mechanism. No new layout has to be computed[1]. Just the two existing layouts[2] 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 can not be detected in a graphical model at first glance.

Nevertheless, there are disadvantages. As can be easily seen in Figure 4.2, it is hard to compare the graphical structure if there was no or a bad layout applied to the Statecharts. The two Statecharts shown differ mainly just in the removed *State 5*, but it is not at all obvious from a short look at the diagrams. This is due to the

[1] Assuming that, in addition to the structural information of each version of the models, the positional information is saved, too. This will be the case with most modeling tools. KIEL, for example, also supports just to save structural information. A layout is computed prior to the display on the fly.

[2] If there is no notational information, a new layout must be computed. This can be done independly, and therefore easily. But one then has to consider if not one of the *merge* approaches would yield better results.



(a) Version 1 (b) Version 2

Figure 4.2.: *Plain* visual diff, added or deleted elements in red, changed elements in orange.

destruction of the mental map. Another source of problems can arise from the size of the Statecharts to compare. If they are sufficiently large, they will not fit onto one screen or they have to be displayed too small to actually see the changes. This can even get worse when comparing two diagrams versus a common ancestor.

This approach is similar to the concept Pounamu uses to display the model differences, as shown in Figure 3.3, except the fact that the Pounamu plug-in arranges the two diagrams as layers on top of each other.

A version of this form of comparison will be used in the implementation, but enhanced with several details, such as scrolling and zooming, to obtain a sound result.

Animation

Another approach could be to create a small animation of the changes. This should not comprehend the whole editing steps performed to get the actual result—as this could comprise several *addition*/*deletion*/*editing* steps—but just a straight transition from the elements of the one version to the state of the other version. The purpose is to maintain the mental map as well, so at least one layout must be known beforehand.

The advantage is that just one diagram of the Statecharts is needed and used to visualize the modifications. This will save up space on the workspace yielding more details or support of bigger diagrams, compared to the first suggestion. Drawbacks are there as well. The algorithm to compute the *morphing* is not trivial and may be time consuming, growing large with bigger diagrams and more changes. As the implementation is done in Java, certain frameworks could be used, as for example KIEL does with the *piccolo* framework Piccolo (2005). Morphing mechanisms can also be employed in Eclipse, reducing the need for computational power and specialized algorithms. Several other issues arise as well. It is unclear where to place states initially that have not existed in the previous of the two versions to compare, or how to report simple changes of the attributes of a state.

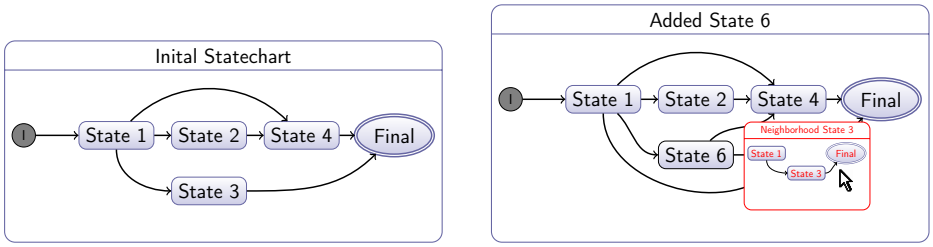
This option does look promising, even though it seems more complicated than others. It could be additionally implemented using the extension point facility for

visualization plug-ins described in Chapter 5. A main drawback is that animations cannot be used when printing out the changes in the diagram to plain paper, as they are inherently dynamic.

Pop-up

The third suggestion is in fact a variant of the first—*Plain*—and the fourth—*Static merge*—, using just one diagram instead of two to show the changes. The original layout of the one version used in the compare operation stays unaltered, normally this will be the most recent version of the two. The user has to select the second version he or she wishes to compare and has to enable the comparison mode. Then, when browsing the diagram and hovering over certain states, pop-ups will appear showing the neighborhood of the state under inspection as it was in the other version. For states with changed attributes a small generic pop-up could inform about them textually. The general appearance of it is symbolized in Figure 4.3.

The advantage is that for the one, the working version of the diagram, no new layout has to be computed. Just overlays will be generated, which is possible in Eclipse. The pop-ups showing changes must use a layout that allows the user to capture modifications quickly. That can be either the original layout, a newly computed layout, or a mixture of both.



(a) Version 1

(b) Version 2, original State 3 under red border

Figure 4.3.: Report changes using pop-ups. The neighborhood of State 3 in version 1 is shown as a pop-up when hovering over it in version 2.

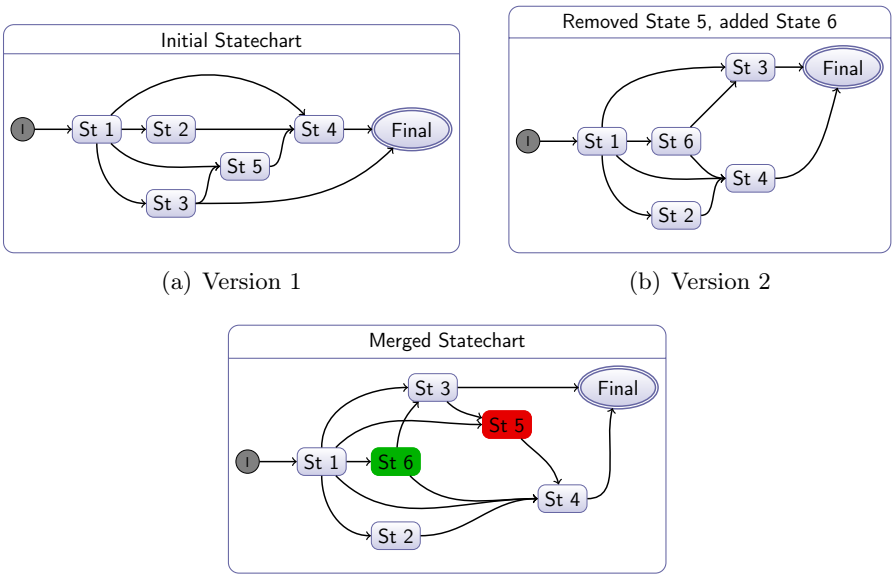
Problems could emerge if the neighborhood of a certain state is too populated, that is if the state under observation is connected to many other states. It will be hard to generate an appropriate overlay then, without covering too much of other relevant parts. Another disadvantage might be that it is not possible to show all the changes of the diagram at once, but just small parts. However, that could be the desired behavior under particular circumstances.

Static Merge

For the static merge concept, especially the term *static* needs a deeper explanation. Static refers to the fact that a layout is computed completely from scratch. The

diagram to render will be a merge of the two diagrams, as for instance a match model computed from one of the model diff algorithms presented in Section 3.1. This merged diagram will contain the superset of all elements from both diagrams. Objects just appearing in one of them will be marked accordingly to their statuses, which can be added, deleted, changed or moved.

The advantage is that no prior positions have to be taken into account when laying out the diagram, and that again just one diagram is needed to present the changes to the developer. Drawbacks are that there has to be done some computation for a new layout, and, what is worse, that the new layout will differ considerably from both previous layouts. This makes it hard for the user to identify diagram objects he or she is used to and to comprehend the changes that happened to them. This can easily be seen in Figure 4.4.



(a) Version 1

(b) Version 2

(c) Merged Statechart

Figure 4.4.: Static visual-diff, deleted *State 5* is marked red, added *State 6* is green.

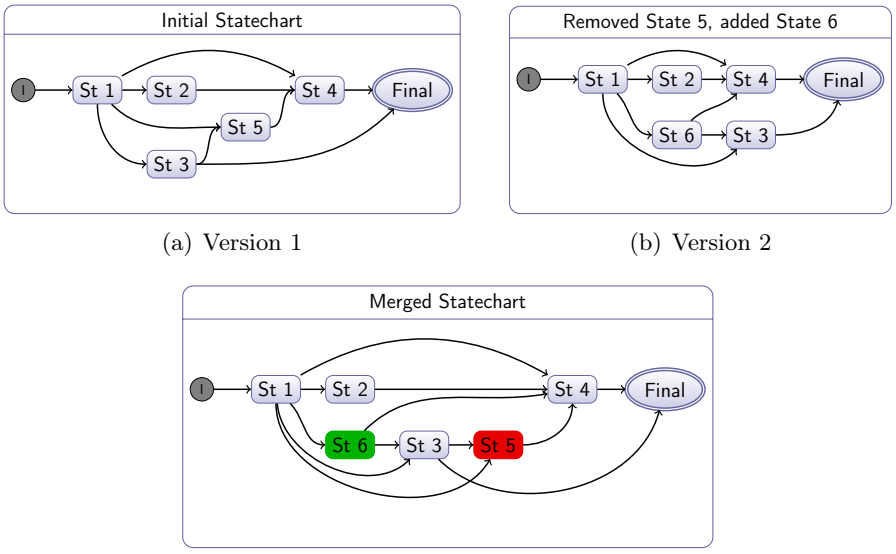
Dynamic Merge

The dynamic merge approach has also similarities to the Pounamu concept. In Pounamu both diagrams are just drawn layer-wise on top of each other, affected by no change in the layout. The dynamic merge tries to solve problems arising from this technique, namely different states in both diagrams with the same or overlapping locations. Therefore the Pounamu approach is not listed separately here, but the improvement to it, the dynamic layout.

The dynamic merge takes one layout of the two diagrams, typically of the newer version of the diagram, or the version that the user is more familiar with and has

built up a mental map for it, and uses these positions to render the merged diagram accordingly. There will be again just one diagram, but one tries to preserve the mental map as much as possible. Of course, this depends on the number and types of changes (Ohst et al., 2003b, Ch. 3.3). As the example in Figure 4.5 shows, the alignment of the states *1*, *2*, and *4* is preserved in the merged Statechart. This is an improvement compared to the outcome of the static merge. The second version of the diagram has also been laid out by an incremental algorithm taking the previous positions into account. There are some algorithms and tools, already presented in Subsection 3.2.3, which can gradually lay out graphs.

This approach is problematic, since with two changes the merged Statechart is already clearly different from both of the original versions. More changes would worsen this. This method is also not very economical; every deleted element still appears in the merged version. Many deletions and addings would make the layout unreadable, to provide the user with the changes would even be harder. The available incremental layout algorithm—DynaGraph—fails to preserve the mental map reasonably when the diagram is affected by more changes.



(a) Version 1

(b) Version 2

(c) Merged Statechart

Figure 4.5.: Incremental layout used to visualize differences, deleted *State 5* is marked red, added *State 6* is green.

The choice

Trying out the proposals manually showed that some of them are not adequate, and some, like *animation*, seemed to be too complicated to implement within this thesis. *Dynamic merge*—as presented here—turned out to be confusing after some testing, as one diagram layout had to be changed; it was even worse for the *static*

merge. The approach to place layers just on top of each other did not work out, as very often overlappings occurred. This can be seen, for example, in Figure 3.3, where a connection is covered by a node. Though not presented here as the most promising approach, the *plain* method, with several helper mechanisms, turned out to be best suited. However, as KIELER is an experimental platform to evaluate several approaches, which help in modeling, none of the presented ones should be declared completely useless.

4.3. Synthesizing the Structural Differences

Having decided on how to represent the changes to the user, a prerequisite that is to extract these changes from the given models. Several promising solutions were presented in the previous chapter, performing their tasks well. However, it is also of importance how and with which parameters the comparison is invoked. This also depends on the type of application.

Implications on the visualization and differencing algorithms emerge also from the lifecycle of a diagram, as already explained in Subsection 4.1.1. Ohst et al. (2003b, Ch. 5) point out that too many differences within a diagram would confuse the user more than giving him or her a support in understanding. This is particularly true when colors are used to indicate the differences. A highlighting is just useful when it draws attention to something, if everything is highlighted, nothing is won. As this also applies to textual files, a general advice is to perform a comparison regularly. Even the best program cannot support a user to understand changes between the initial version and the finished product.

When comparing diagrams, there is always the issue with the semantic information, often called *domain model*, and the layout information, often called *notation model*. When performing the comparison, one has to know what should be included in the compare process. This of course also depends on the actual model behind the diagram. Imagineable are models where the layout information is as important as the semantic information. In the area of Statecharts, though keeping the considerations of secondary notation in mind, the layout information saved in the diagram itself has been considered unimportant (Prochnow and von Hanxleden, 2007), as long as a good layout engine can produce an appealing, and moreover an understandable layout afterwards. For this thesis, also resulting from the considerations on which representation to use in the chapter before, just the domain model will be compared. For that reason, this section is titled *structural differences*, and when referring to comparison, the domain model is meant, unless stated elsewise. As said, other implementations for different models and editors, or different types of graphical representations for the user, might take the positional and size information into account.

Fortunately, when generating diagram editors with GMF, which is used to create the Statechart editor, a strict separation of layout and semantic information is the normal case. Unfortunately, other frameworks like Generic Eclipse Modeling System

(GEMS) do not take this approach.

Still, it is open so far what and how to compare exactly, which will be addressed in the next subsections.

4.3.1. Comparison of Two Models

The first case is to compare just two plain models with each other. There are generally two alternatives to get the differences, which were already mentioned when presenting works in the previous chapter:

- *offline*: This is the natural or standard variant when comparing models. Both models are taken and the differences are computed *offline*, that means subsequently, without knowing anything about the history of the models. Depending on the quality of the algorithms, this can lead to high-quality results. The biggest advantage is that it can be used anytime to any kind of model and in any editor. It is a highly generic approach.

- *online*: This method does not compute the changes afterwards, but monitors editing steps applied to a model. With this approach, the changes or differences are explicitly written down, and thus completely reliable. Additionally, a complete history of the editing steps comes for free. The main disadvantage is that the editor has to be prepared for that. Normally that entails considerably changes to the source code of the editor. Besides, models which were not created and altered with such an editor cannot be compared later on.

The online approach was not chosen due to several reasons. The chosen development environment—Eclipse—limits the ability to create an editor that monitors the changes. Second, and which is the main reason, the implementation should also work with diagrams imported into it. This leverages also the option to use the visual comparison with other editors than the Statechart one.

4.3.2. Comparison of Three Models

Three models are generally compared such that one model is the common ancestor, and the other two reside in different branches, edited perhaps by different developers. Though they are distinct, they still share some similarities, with each other and their ancestor.

A three way comparison will not be considered, but there are some hints in the papers that were already cited in Chapter 3.

4.3.3. Comparison Incorporating History Information

This title may sound odd, as it seems to coincide with the previous concept of *online* comparison, but means something different. Of course, a precondition for such a comparison is some history information of the models. If a complete history of a

diagram is available, a differencing algorithm can use this information to trace the development of the diagram and use this information when rendering the diagrams. This applies directly to the representations called dynamic merge and animation. An approach using a similar technique is also presented in Subsection 4.4.1.

The Choice

The choice of the comparison type—and tool—goes along with the choice of the representation, and both evaluations came from the manner of working with Statecharts. It seems most valueable for a user to have two versions of the Statechart facing each other, as he or she is used to from textual comparisons. As already explained, not to much time should pass between two comparisons, thus affecting the layout marginally. Therefore, history information could be left out, leading to a simple comparison of the models. This is all what is needed to get a representation as requested in Visualization of Differences in Section 4.2.

4.4. Mapping the Changes to a Graphical Representation

So far it has been decided how the general way to denote the changes to the user will be, namely with two diagrams next to each other, every version with its original layout. A general agreement is that changes should be marked by colors to support the users in finding them. This seems quite natural, as otherwise little would have been won, with just two diagrams drawn side by side. The coloring method is also broadly used when comparing textual files and many works dealing with structural comparison came back to this proposal.

To state it precisely:

Coloring: Each of the possible edit operations, of which there are *added*, *deleted*, *changed* and *moved*, should find its representation in the two diagrams, with the respective color.

That seems to be handy for small diagrams with few changes, but two problems arise when working with bigger diagrams. Too many changes would lead to a gaudy mix of colors, leaving the user unable to recognize anything. Furthermore, if huge diagrams are displayed next to each other, navigating through the changes can be tedious. This leads to a second claim, or even limitation, which is commonly accepted in literature:

Limitation of changes: Limit the changes to a meaningful extent, resulting mostly in smaller time slots for subsequent comparisons.

To address the navigation through the diagrams and the changes, there must be an easy way to find the changes the user is interested in, regardless how big the diagram is and how dense or sparse the differences turn out to be. A promising approach is to provide the user with a structured textual list of changes on the same screen as

the diagrams, supporting the colored graphical display, and enable a scrolling to and highlighting of the difference the user has clicked on. An adaptive zooming function is also valuable, to get a rough overview of the changes in the whole diagram, but to be as well able to see certain changes more in detail. To summarize:

Scrolling and zooming: Provide automatic scrolling and zooming facilities to direct the user to the elements he or she is interested in.

Prochnow and von Hanxleden (2006) elaborate the concept of a SNF and dynamic Statecharts with focus and context. With huge Statecharts, covering wide diagram areas, incorporating many hierarchy levels, but containing few changes, a similar concept is helpful. Transferred to a comparison view and other models than just Statecharts, this means that hierarchical elements with no changes in lower levels can be collapsed and the layout is computed again, leading to a lower consumption of space and a concentrated presentation of changes:

Focus-and-context: Allow for an optional focus-and-context representation of the changes in the diagram, masking out elements not of interest.

A drawback is of course the destruction of the mental map. But as this function can be switched off, the user can decide what is more valuable for her or him.

4.4.1. A Mixed Approach

Pilgrim (2007) has an interesting proposal, covering and combining the separately listed domains above, which are difference detection, visualization and mental map, incremental layout, and an appropriate representation. The main focus lies on the preservation of the mental map of the user when working with different versions of a diagram and automatic layout of them. A concept already mentioned, the separation between domain and notation models, is one of the foundations of his approach. Initially, the user creates a domain model, the notation model is automatically rendered. This is also in concordance with the SNF and the forced automatic layout of Statecharts (Prochnow and von Hanxleden, 2007).

$$\begin{array}{ccccccc} \ldots & \xrightarrow{t_{i-1}} & DM_i & \xrightarrow{t_i} & DM_{i+1} & \xrightarrow{t_{i+1}} & \ldots \\ & & \downarrow & & \downarrow & & \\ \ldots & \xrightarrow{t_{NM}} & NM_i & \xrightarrow{t_{NM}} & NM_{i+1} & \xrightarrow{t_{NM}} & \ldots \end{array}$$

Figure 4.6.: Transformation of notation models, after Pilgrim (2007)

When editing a diagram, changes are applied to it, denoted by the t_is in the upper part of Figure 4.6. The normal way of using an automatic layouter would be to query the domain model DM_i, to get the corresponding notation model NM_i,

after the rendering with a static layout algorithm. In this standard procedure, the transformations t_{NM} are not used. For every new version of the domain model a new notation model is computed from scratch, often resulting in a completely new layout.

The proposal is to also use the transformations t_{NM} for the notation model, which means for every new rendering of a new diagram version, take the actual domain model and the previous notation model into account. That is not new, as the explanations of, for example, GraphViz and DynaGraph, and yFiles in Subsection 3.2.3 show. The benefit of his work is to apply this concept to a revisions history of model diagrams. A comparison would also be possible with this approach, though again the editor has to maintain a complete history of domain and notation models. A proof of concept is given with his newly developed GEF3D framework (Pilgrim, 2008), showing even two models in layers on top of each other in a three dimensional diagram space, as can be seen in Figure 4.7.

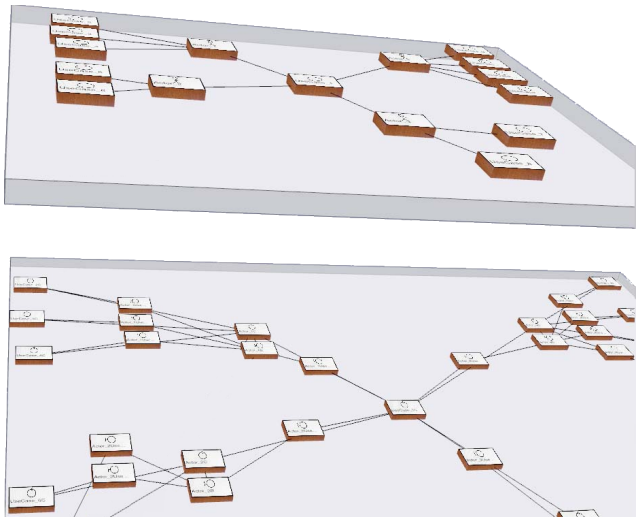


Figure 4.7.: GEF3d example, picture taken from (Pilgrim, 2008)

The Choice

There is actually no choice, because no existing proposals appeared useful for this application area. The points as listed above—coloring, scrolling and zooming, and focus-and-context—are chosen.

Summary

This chapter exposed the visual diff problem. Following possible scenarios at the beginning, a first emphasis was on the visualization of the differences with a preference for a side by side layout of the two diagrams. The differences will be computed solely

from the two domain models from scratch, and the visualization will be enhanced by colors and several user interface helpers for the developers. This constitutes the requirements for the implementation in the next chapter. The layout was addressed just shortly, as it will also be presented together with the implementation.

"Selbst ohne Ehrgeiz und Initiative haßt die kompakte Mehrheit nichts so sehr wie Neuerungen. Sie hat dem Neuerer, dem Pionier einer neuen Wahrheit immer Widerstand geleistet, ihn verurteilt und verfolgt."

Emma Goldmann - Minorities versus Majorities, in: Anarchism and Other Essays, New York, 1910

5 The Implementation

This chapter presents the implementation of two Eclipse plug-ins developed in the context of this thesis, the layout plug-in and the visual-diff plug-in. According to the naming scheme used in projects like KIEL and KIELER introduced below, the plug-ins follow a similar strategy. The layout plug-in is part of a bigger sub-project of KIELER, named the KIELER Infrastructure for Meta Layout, and the visual-diff plug-in is named KIELER Visual Komparison (KiViK). The writing with **K** instead of **C** is intentionally, of course, to achieve a symmetry already in the word itself, symbolizing the confrontation of models in the plug-in. Section 5.1 gives an introduction to the KIELER project and Eclipse. In Section 5.2 the Statechart data structure, which had to be implemented in order be able to build up models to apply the plug-ins on, is explained. The next two sections, Section 5.3 and Section 5.4, present the actual plug-ins in detail. Design considerations, especially concerning future enhancements, are presented at the beginning of each section and an elaborated explanation of each plug-in is given, including diagrams, features and drawbacks, problems faced and used third-party projects, as well as a small conclusion.

5.1. The KIELER Framework

The infrastructure used to deploy the plug-ins is the Kiel Integrated Environment for Layout for the Eclipse Rich Client Platform framework. This is an experimental software framework, originating from The KIEL Project (2006), to test new methods and paradigms related to embedded system development. The key part is the graphical model-based system design, conducted using concepts like dataflow languages and Statecharts. As the literal **L** in KIELER depicts, a focus is drawn on the layout and readability of those graphical descriptions of systems.

KIELER is a refinement of KIEL, as it provides support for more languages than just Statechart dialects and because it is ported to the Eclipse rich client platform. It makes use of several Eclipse technologies and libraries such as GEF, EMF and GMF—described below—to save development efforts and to comply to existing industry state-of-the-art standards in order to be used later on by the Eclipse community.

5.1.1. Eclipse Overview

To gain a better understanding of the implementation, some of the key concepts of Eclipse that have been used are presented. Those Eclipse technologies leverage the plug-ins as well as the KIELER framework.

The infrastructure behind Eclipse was changed to the OSGi Service Platform (OSGi Alliance, 2008) with version 3.0 in 2004. The Eclipse reference implementation of the OSGi specification is called Equinox (Eclipse Foundation, 2008a), basically a plug-in system allowing developers to easily enhance behavior of the Eclipse IDE or to develop applications on their own. If none of the manifold extensions of the standard Eclipse IDE is needed for a specific project, it is also possible to develop an application from scratch, just making use of some common functionality. This is known as the Eclipse rich client platform, consisting of Equinox OSGi, the core platform, the Standard Widget Toolkit (SWT) (SWT Community, 2008), JFace (Eclipse Foundation, 2008b) and the Eclipse Workbench (Figure 5.1). With these tools, it is possible to develop lightweight but nevertheless functional applications in a very fast and convenient manner.

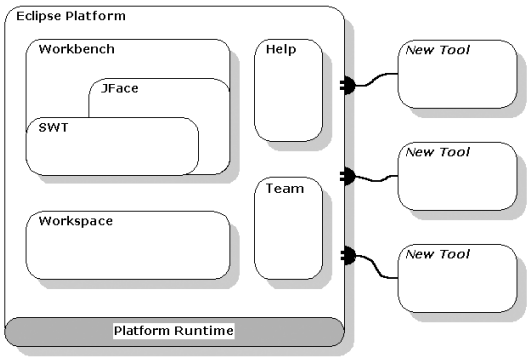


Figure 5.1.: Concept of the Eclipse platform architecture (Eclipse Software Foundation, 2008).

The Eclipse plug-in and bundle concept enables the developer to make use of a huge pool of existing code and libraries, packed in bundles that are easily installable through the Eclipse IDE itself or via downloadable packages.

One key feature provided by Eclipse and used by KIELER, maybe also one of its foundations, is the mechanism of *extension points*, as shown in Figure 5.2. Plug-ins can define new extension points which can be used by other plug-ins or can use

extension points provided by other plug-ins. There are no limitations in this concept, so that there can be a hierarchy of extension point usage and plug-ins can use and define such points simultaneously and vice versa.

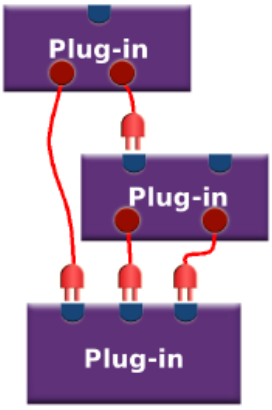


Figure 5.2.: Eclipse extension point mechanism (Eclipse Software Foundation, 2008)

An extension point is quite similar to an API and requires a formal description of the services it provides. During runtime, due to the concept of lazy loading of Eclipse plug-ins, the platform checks if defined extensions points are used by other plug-ins and activates them, if there is an operation triggering the use of an extension point. If no other plug-in implements a to be used extension point, the providing plug-in is informed and can react to this. This is basically like a dynamic way of using an API.

5.1.2. EMF, GEF and GMF

In the Eclipse world, like in the KIELER one, many confusing and similar looking acronyms are used. Three of the big ones among Eclipse components are the Eclipse Modeling Framework (EMF), the Graphical Editing Framework (GEF) and the Graphical Modeling Framework (GMF).

Eclipse Modeling Framework

EMF (Merks, 2008) is a modeling framework with light code generation support for the Eclipse platform. Models can be imported from various specifications like UML, XML or Java classes or created directly in the shipped editor. The model itself is represented in *ECore*, an appliance of the Essential Meta Object Facility (EMOF), a subset of the Meta Object Facility (MOF) (The Object Management Group, 2006), the metadata architecture of UML. The advantage of ECore is its simplicity in comparison to languages like UML, though it retains enough expressive power for most use cases.

Code generated by EMF comes with several nice features. The built program can easily create instances of the model, can query, manipulate, serialize and validate it. There are means to monitor changes to apply the Model-View-Controller (MVC) paradigm. It is possible to generate JUnit code. The resulting code incorporates not only the plain model, but also wizards and editors up to a complete Rich Client Platform (RCP) application. The RCP applications are quite powerful. For example, methods like code validation and transformation can be used. For an overview see the seminar homepage of the Real-Time and Embedded Systems Group (2008).

Graphical Editing Framework

GEF (Hudson, 2003) enables graphical editing of models within the Eclipse framework. It is geared to the MVC paradigm and presents the view and controller as appealing graphical elements, which can be highly customized by the developer. Figure 5.3 shows this concept. The model can be any model, but in many cases the developer will choose an EMF model (see also Graphical Modeling Framework). The representation—or view—is done by *draw2d*, the 2d drawing framework included with GEF. Entities called Editparts act as the glue between the figures and the model and constitute the controller of the MVC concept.

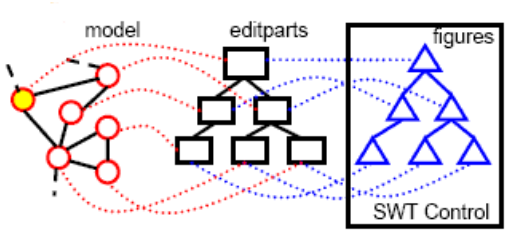


Figure 5.3.: GEF foundations, picture taken from the Eclipse GEF Homepage

Changes performed by the user to the model are processed with a request/command architecture. A customizeable palette offers tools to the user to manipulate and create new elements of the model in a graphical way, by default in the standard WYSIWYG fashion. This is in contrast to KIEL, where the structure-based approach is favored. The main purpose of GEF is to provide a powerful system to develop and build graphical editors, which themselves are often used to work with all kinds of models in a convenient way. The editors needed by programmers and users are often dedicated to a certain DSL, which settles in a particular model. In many cases it is a tedious piece of work to keep the model consistent with the respective editor. Changes in the model lead to changes in the editor and vice versa. Filling this gap is the intention of GMF.

Graphical Modeling Framework

GMF (Gronback, 2008) aims to combine the advantages of EMF and GEF while avoid-

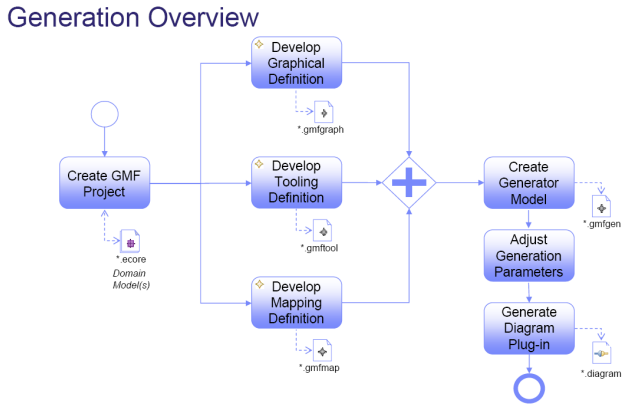


Figure 5.4.: GMF generation overview, picture taken from the Eclipse GMF Homepage

ing the problems arising from synchrony issues explained in the previous section. In the same manner like in EMF, the developer defines a model appropriate to the special requirements. The corresponding (graphical) editor is then generated in the same step as the Java code for the model itself. The developer has, of course, to adjust the look and shape of the view elements for the model elements and to choose the desired tools to work with the model.

This mechanism is exhibited in Figure 5.4. The start is the ECore file, which holds a description of the model. Next, the representation has to be defined, this is done in the **.gmfgraph* file. The same holds for the tools the developer wants to offer to the users, the corresponding file is **.gmftool*. In another file, **.gmfmap*, the connections between the graphical and the model elements are made. With all these three files the generator model is created, **.gmfgen*. Some adjustments can be undertaken in this file, too, finally the whole code for the graphical editor can be created with just one click.

Unfortunately, though this concept can save a lot of time when developing, and especially when changing editors, it has a steep learning curve, which levels out some of the advantages at the beginning. Another drawback is that one does not see all the code which is generated, as if it would be the case if one creates an editor on one's own from scratch. So it may be hard to identify and find errors. And, last but not least, customization can take a long time, and very often just the standard generated editors are not enough, even for standard use cases.

5.2. The KIELER Safe State Machine Data Structure

At the moment of writing this thesis, the development of KIELER is still at the beginning. Some core functionality is already provided and several sub-projects are

in their incubation phase[1]. But there was no tool to develop SSMs, and therefore no possibility to apply layout and comparison algorithms to them. The first challenge was to develop a Statechart editor.

The instrument of choice was GMF, though there are other toolkits to develop graphical editors with Eclipse, like the Generic Eclipse Modeling System (GEMS) (White et al., 2008). An evaluation of different editor builders will be covered in Bayramoglu (2009).

The foundation of each editor developed with GMF is the model, directly describing the desired semantics. As the semantics chosen within the KIELER framework regarding Statecharts serve SSMs (André, 2003). The SSM datastructure developed so far will be used and extended in further theses (Starke, 2009), where the emphasis is on the semantics and not on the graphical representation, though the semantics contributes to a correct and understandable layout.

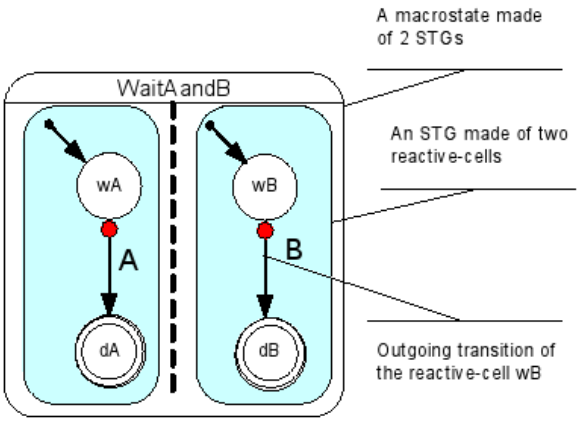


Figure 5.5.: SSM macrostates and STGs, taken from André (2003)

When transforming the SSM specification to an ECore model, which is shown in Figure 5.6, close attention was paid on performing this process transparently and adhering to the naming scheme as closely as possible. However, some compromises were made to comply to the notation already introduced and known from KIEL. Therefore, the original *macrostate* from the SSM specification is named *composite state* in the KIELER SSM model, called KSSM from now on to distinguish the two different Statechart models. Where possible, inheritance relations were used in the creation of the model. The main building blocks of the SSMs are *states* and *composite states*. Another entity, an *abstract state*, was therefore introduced of which both states inherit. It is possible in the ECore modeler to set such a class abstract, so that no instantiation of this class would occur later on. The same was exploited when introducing the various kinds of transitions in the specification.

[1]See the project homepage http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/ for an overview of sub-projects.

Other important elements required for a demonstration of a layout are labels. Most kinds of states have a label, this is always drawn within the boundaries of the state. States without a label are, for example, *initial* and *conditional states*. Transitions also hold different sorts of labels, depending on the type. The so called *normal termination* solely possesses an effect, which has to be drawn in the graphical representation. A *strong abortion*, for example, has a trigger and the possibility to set it immediate. Every transition, in contrast, can have a priority, commonly drawn at the tail of it, while the other labels are drawn at the center.

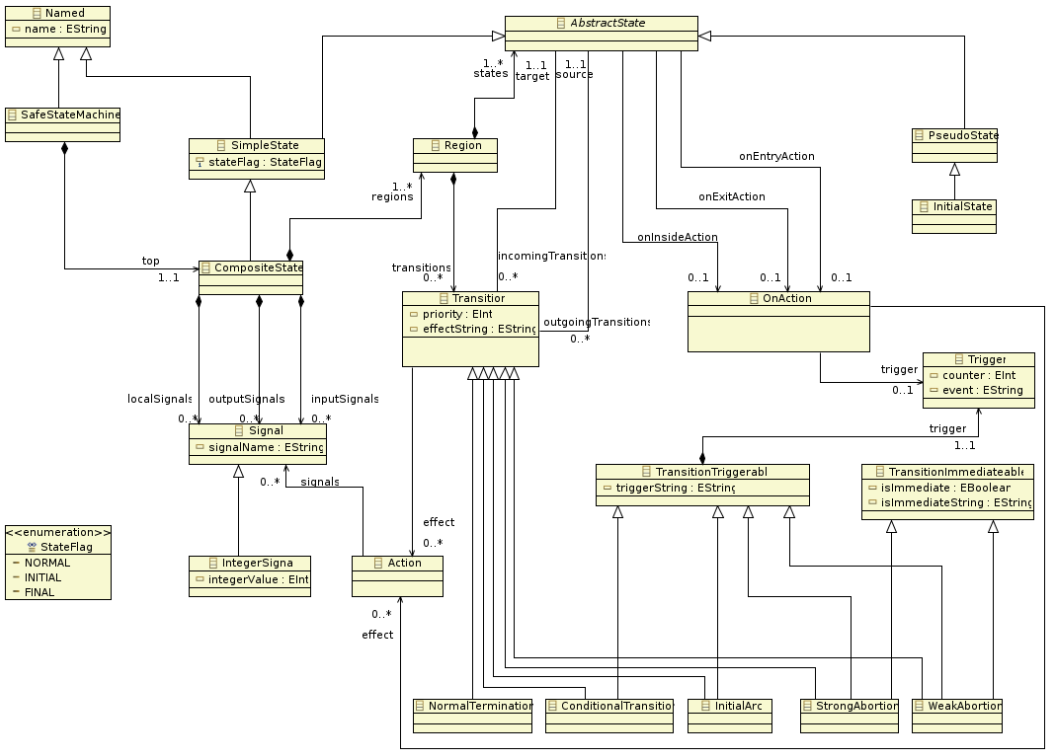


Figure 5.6.: The KIELER SSM Ecore model.

An advantage of the SSM structure is a strict containment policy. As a result, every Safe State Machine can be represented as a tree. This is exploited in the layout plug-in Section 5.3, when the Statechart is transformed into a *layout graph*. The SSM tree alternates *composite states* (macrostates in the original paper) and *regions* (State-Transition Graph (STG)[2]), with simple states as leafs, see Figure 5.5. This helps while modeling in the ECore editor and when setting the tooling, graph and mapping definitions (see Figure 5.4).

The most important elements of the KSSM in relation to a layout algorithm are the states, regions and transitions. Elements like triggers, signals and actions are

[2] An STG is a set of reactive cells, which is basically a set of some kind of states connected with transitions.

not taken into account, as they have no or just little graphical counterparts. To be able to work with transition labels, they were simply modeled as strings, which is enough for laying them out. The future EMF model may use another representation. As described before, GMF is not just used to bring the model into being, but also to create the classes and a structured editor, and, what is most important for the plug-in, the graphical editor. With the KIELER SSM editor it is possible to create simple graphical models of Safe State Machines. Though there is no functionality beyond the simple creation yet, it is sufficient for the demonstration of the models and acts as a starting point for a deeper implementation. The graphical editor is shown in Figure 5.7, with a very simple KSSM model and the toolbar on the right.

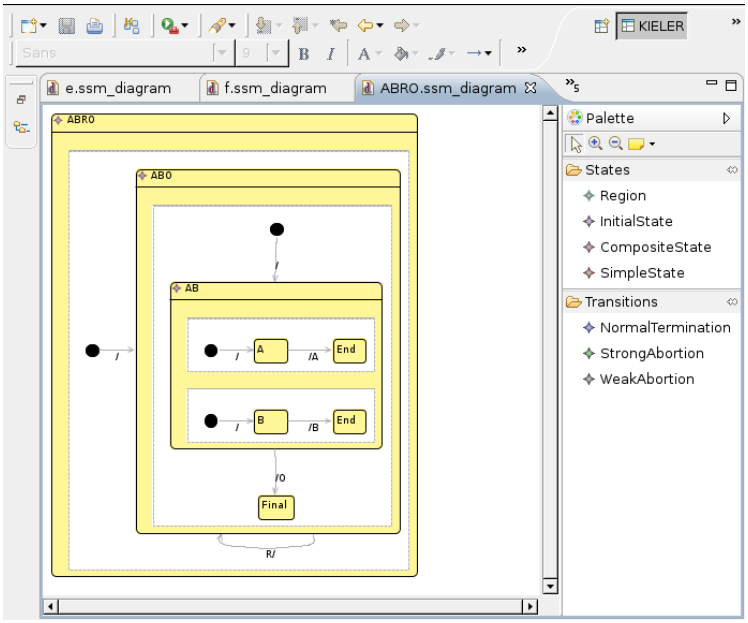


Figure 5.7.: *ABRO* in the graphical KIELER SSM editor, with the generated palette on the right. The layout was computed automatically.

Hard parts in the building of the graphical editor were, beside the complexity and elaborateness of GMF in general, the correct mapping of model elements to graphichal elements and an appealing choice of shapes and colors. The shapes are inspired from the normal SSMs and the colors in most cases, too. Problems arose also from the different types of layout that can be applied to container objects like regions and composite states, which later turned out to interact heavily with the layouter plug-in, so that it had to be adjusted several times.

The GMF approach led to an interesting advantage. A KSSM, like any GMF model, is saved to disk with domain information split from the notational information, though saved in one file. An example can be found in Appendix A. A benefit of this splitting of the model is used when comparing models in the visual-diff plug-in. As the structural differences should be displayed, it is sufficient to compare just the domain

model. That is much faster without all the additional layout information, as well as more reliable.

5.3. The Layout Plug-in

One demand of this thesis was the layout of Statecharts, one of the key research areas of the whole KIELER project. Though this thesis mainly deals with the layout of Statecharts, and in this case with a certain dialect thereof, SSM[3], attention should also be drawn that the layout mechanism remains useable for different kinds of diagrams. On the one hand to be able to be used by other diagram developers throughout the Eclipse community, on the other hand to act as a fundament for other languages to be implemented in KIELER, like *SCADE*/Stateflow.

5.3.1. Design Considerations

Before the implementation, several design considerations have been made. Some key questions were:

- What layouters to use?
- Structure of projects, packages and classes.
- What will be the general architecture?
- How to integrate into the Eclipse IDE?
- How to design the UI, the look and feel, and handling.

As for the layout/layouter case there has been some research regarding layout in general (Section 3.2) and for Statecharts (Prochnow, 2008; Völcker, 2008). It turned out that the GraphViz suite, and of this the Dot-layouter, delivers a satisfying piece of work when used to lay out Statecharts. The writing of a new layouter is beyond the scope of this thesis and therefore the GraphViz suite was chosen to serve as the layout back-end for the layouter plug-in. Works of Kloss (2005) laid the foundations for a connection of the GraphViz layouters to the Java and KIEL world[4]. However, the GraphViz layouters were not available for Eclipse.

At the beginning of the writing of this thesis, there was already a skeleton infrastructure for the KIELER framework. Concerning the packages and projects, the only task was to add some new ones into the existing hierarchy. The layout plug-in is divided into several Java projects and makes use of many packages, adhering to Eclipse and KIELER naming schemes. Classes were distributed to packages and named closely to the conventions used in other popular Eclipse plug-ins.

[3] One point relevant for layout issues of SSMs is that they forbit inter level transitions, which are transitions over composite state boundaries, whereas other dialects allow them.

[4] As it turned out at the end of the thesis, it is also possible to compile the GraphViz source directly for the use in Java projects.

While most user interface design decisions can be altered relative easily, which means to add a context menu or to move a button from the one toolbar to another, the core architecture had to be designed foresightedly, to enable other layout plug-ins and editor developers to contribute to the KIELER Infrastructure for Meta Layout.

For this reason, the layout infrastructure is designed highly modular. To link an arbitrary number of layouters to an equally arbitrary number of diagrams and editors, a common data structure is used. This structure, the `KLayoutGraph`[5] encodes all the relevant information of the diagram which is supposed to be laid out and serves as common means to provide this information to any layouter to be used.

A layout process is basically done as follows:

1. Translate the diagram from the representation of the editor to the `KLayoutGraph`.
2. Send the `KLayoutGraph` or parts thereof to one or more layouters; they enrich the data structure with the layout information.
3. Apply the information now in the graph to the elements of the diagram in the editor.

With a wise design of the layout graph, many diagrams and models can be integrated in this structure and be laid out with any layouter that understands the `KLayoutGraph`. During the writing of this thesis, this infrastructure was also used successfully to start the development of a layout algorithm for Dataflow models (Spönemann, 2009).

Questions about the integration into Eclipse, or the respective editors, as well about the user interface and handling were addressed pragmatically and adjusted during the implementation process.

As all models of editors generated with GMF share the same graphical elements, at the beginning of the thesis it was thought that one general algorithm for the translation of the graphical elements to and back from the `KLayoutGraph` was enough. When delving deeper into the matters, and when the beginning of the Dataflow editor started, it turned out that this was not the case. This will be covered more in-depth in Subsection 5.3.5.

5.3.2. General Architecture

As said before, a well designed architecture for the layout plug-in is vital. The foresighted structure turned out to be adequate when the implementation for an additional editor for Dataflow models started at the end of this thesis. The integration could be done relative seamlessly.

The following pages will describe the actual implementation. Several peculiarities of the environment for the plug-in have to be presented beforehand. As described

[5] The prefix *K* is used throughout this project to associate all the datastructures to KIELER and KIELER Infrastructure for Meta Layout (KIML).

in Subsection 5.1.2, the diagram consists of some building block elements, corresponding to the view, the model and the controller. This will also be of importance when reasoning about the possibility of a general layout (or translation) algorithm. Another specialty of GEF/GMF is that graphical elements must be repositioned and resized by commands, and that these commands just work in that cases that relative movements starting from the current position are possible, resulting in some of the limitations described later.

The main foundation for the layout plug-in is the `KLayoutGraph`. The data structure of the graph itself was modeled with the Eclipse ECore tools and the respective Java code was generated automatically. In Figure 5.8 a graphical representation of the ECore model is shown.

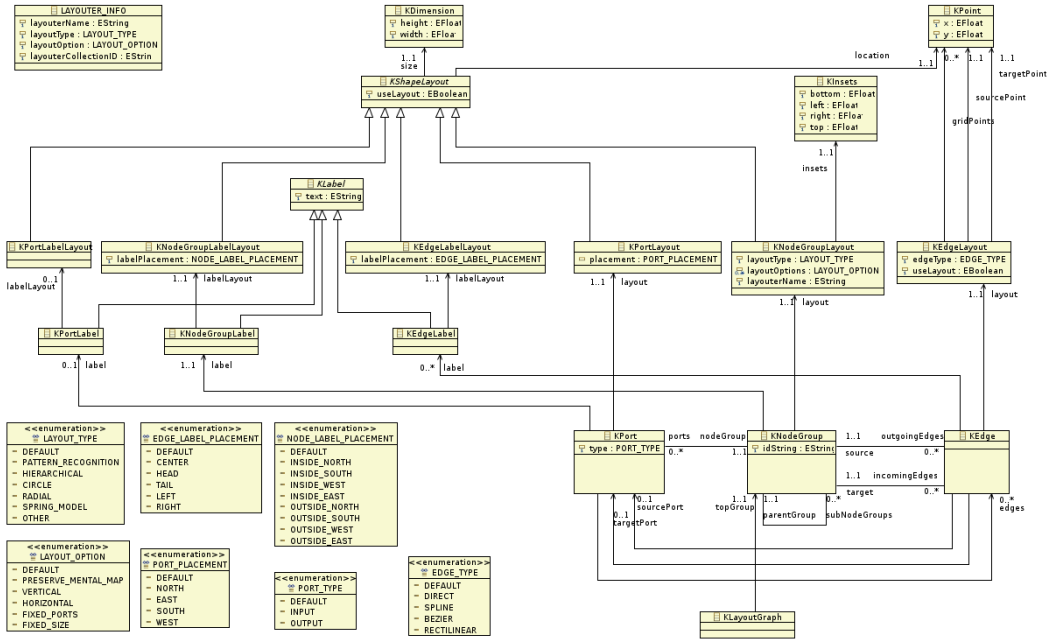


Figure 5.8.: ECore model of the `KLayoutGraph`

The core elements of graphical models in the GMF environment are nodes and connections in different flavors. A characteristic of so-called `NodeEditParts` is that they can contain other Editparts themselves. That implies already the main elements of the `KLayoutGraph`, nodes and edges. As nodes can contain other nodes, they are called `KNodeGroups`, and the connections are called `KEdges`. This is already sufficient to store the general structure or model of most diagrams. For that reason, which complies completely to the design of the Editparts, the graph is actually a tree, which enables one to lay out every node or leaf independently.

During the development of other diagram editors, namely the Dataflow editor, the need for some more elements in the layout graph arose. A specialty of Dataflow models is for example their input and output ports. As they should generally be laid

out in a special manner, for example input ports left, output ports right, this new entity had to be incorporated into the layout graph.

With these parts of the graph, the model can already be transmitted to layouters, reading out the structure and applying a layout computation to it. To store the computed layout, additional facilities are needed. Therefore, new objects to store the layout information are added to any element of the layout graph as a reference. With this separation, the graph can be used with or without the positional and size information and remains more flexible.

As another demand was to be able to render distinct groups of graphical Editparts separately, each layout information of the node groups can be annotated with the desired layout type and even the actual layouter to be used. Other elements of the `KLayoutGraph` are IDs, different types of labels for edges and nodes, and general options for every group and for the whole graph.

The process taking place during a call for a new layout for a diagram is illustrated in the sequence chart in Figure 5.9.

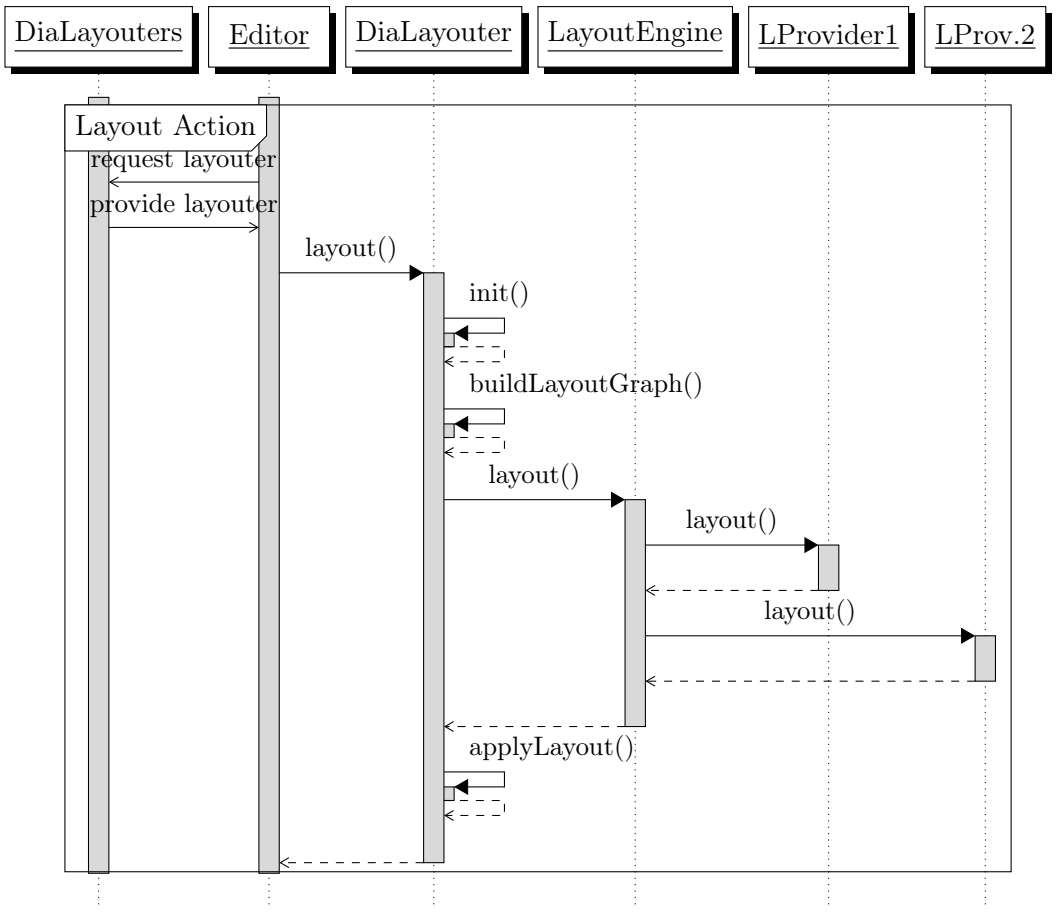


Figure 5.9.: Schema of the KIML layout process

In the first step the editor or viewer which wants to apply a layout to a diagram request the class `DiagramLayouters` for an instance of a layouter for the concrete diagram. The mapping from the diagram to the layouter is done with the ID of the editor, an attribute any GMF editor possesses.

With this *diagram layouter*, the editor calls the layout method and provides the object it wants to be laid out. That can be a selection of edit parts, the whole diagram, or a single *Editpart*, among other things.

The `init` method of the diagram layouter is responsible for extracting adequate elements of the diagram for the layout process from the provided object. Then, the diagram layouter builds up the corresponding `KLayoutGraph` for the provided elements.

With this data structure the `layout` method of the `KimlRecursiveGroupLayouterEngine` is called. This method walks through the provided layout graph and calls for every level of hierarchy in it the best fitting *layout provider*, according to the layout hints attached in every node group.

The respective layout providers are responsible to render their provided node group. They must also write back the size of their rendered part of the diagram to enable the layouters of the higher hierarchy levels to perform their task correctly.

Once the whole `KLayoutGraph` has been laid out, that means that all the position and size information of the whole diagram is available in the graph, the diagram layouter is responsible to resize and move the elements of the editor according to the information in the data structure. This is mostly done by constructing commands, which is the way to do this in the GEF world.

5.3.3. Created Extension Points

The Eclipse extension point mechanism is used to connect the diagram layouters and the layout providers to KIELER and KIML, especially those which will be created by other developers. For each of the diagram layouters and layout providers a new definition of an extension point was created.

The first extension point for the diagram layouters in called *kimlDiagramLayouter* and the picture of it is shown exemplary in Figure 5.10. Information which has to be provided to fulfill the definition of the extension point in a class extending `KimlAbstractLayouter` and an editor ID. The editor ID is used to map the right diagram layouter to the respective editor. As each of the—with GMF—generated editors is dedicated to one type of diagram or model, this assignment is useful. At startup of KIELER, the class `DiagramLayouters` is loaded and queries all plug-ins providing a diagram layouter through this extension point. The diagram layouters are stored in appropriate data structure. The purpose of the diagram layouters and the need for more than just one is explained in the next subsection.

The second extension point is called *kimlLayoutProvider* and is responsible for the registration of layout providers in the KIML framework. Its definition is even simpler, just a class has to be provided, extending `KimlAbstractLayoutProvider`, which performs the actual computation of the layout, given the `KLayoutGraph`. The same

mechanism as with the other extension point is used: at startup all plug-ins providing layout providers are searched and found providers are stored in a data structure to be used later on.

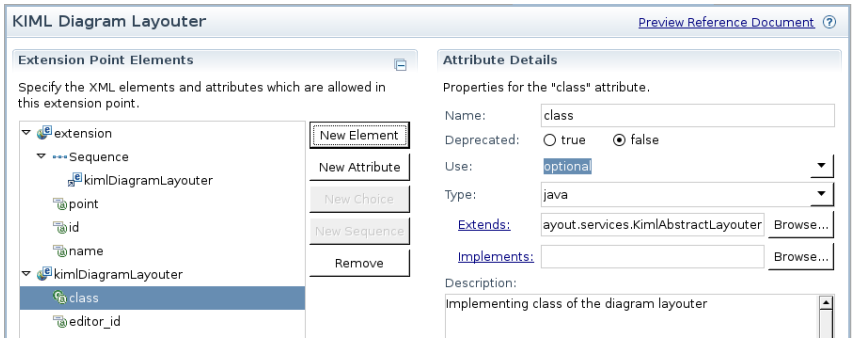


Figure 5.10.: The *kimlDiagramLayouter* extension point description.

5.3.4. Layout Classes in Detail

In this subsection the main layout classes, as implemented, will be described in detail. A class diagram illustrating the structure and the connection of the classes can be found in Appendix B, as it is too huge to display it here. All these classes are located in the sub package services of the layout plug-in package, except the `KimlGenericDiagramLayouter`, which is in the KIML UI project, and the respective layouters. Some of them are just abstract classes which have to be extended by concrete implementations to serve the special requirements. The classes are mostly like already mentioned in Figure 5.9:

- `KimlAbstractLayouter`: The abstract base class for diagram layouters. They translate the elements of the editor to the `KLayoutGraph` and back.
- `KimlGenericDiagramLayouter`: Generic implementation extending the `KimlAbstractLayouter`.
- `DiagramLayouters`: Singleton instance for all the available diagram layouters.
- `KimlAbstractLayouterEngine`: The abstract base class for the layout engines.
- `KimlAbstractLayoutProvider`: The abstract base class to be implemented by layout providers, which are responsible for the actual computation of the sizes and positions.
- `KimlNullLayoutProvider`: Dummy class, if no layout provider available.
- `LayoutProviders`: Singleton instance for all the available layout providers.

KimlAbstractLayouter

If a diagram in an editor should be laid out, all what is needed is the dedicated DiagramLayouter for this diagram type. Any implementation of a diagram layouter has to extend the abstract class `KimlAbstractLayouter`. The methods that need to be implemented are:

- `init`
- `buildLayoutGraph`
- `applyLayout`

Besides, there are other methods like `getSettings` and `getLabelProvider`, which can be overwritten to fulfill special needs of certain models. However, the core functionality results from the three methods above. Once they haven been implemented, it is possible to call the already existing function `layout` of the DiagramLayouter, which starts the process described in Figure 5.9.

The most important function is `buildLayoutGraph`, which is responsible for the translation of the diagram into the `KLayoutGraph`. In most cases, a knowledge of the underlying graphical meaning of single elements is helpful, if not even necessary.

The function `applyLayout` must implement the application of the size and positional information now contained in the `KLayoutGraph` to the elements in the editor. This is much easier when using maps to maintain links between elements of the diagram and the layout graph. Problems arose in the concrete implementation, due to some GMF specialties.

KimlGenericDiagramLayouter

This is the first implementation that was written to fulfill the `KimlAbstractLayouter` methods. At the beginning of this thesis, it was tried to use one layouter for the translation of all models, as all models become manifest as nodes and edges in the GMF diagram editor. Soon it became evident that this was not the case. The first tries for a KSSM model were quite satisfying, but with other models, for example the ones of the Dataflow editor, or simple test models, the need for a specially dedicated layouter arose. Differences of the models are, for example, the ports of Dataflow models, several different concepts to model the compartment Editparts like composite states and regions, and the labels, those of states as well as of connections. All these graphical counterparts of the elements could not be identified easily and lead to a quite complicated structure of the code, just to address a small number of models. Therefore, a very simple and clear implementation serving as a generic translator was written. Some options help to adjust the generic layouter to certain models.

The `KimlGenericDiagramLayouter` does a promising job, as can be seen with the UML Statemachines example in Section 6.2.

DiagramLayouters

This class, implemented as a globally available singleton, serves as the instance providing all available diagram layouters. If a developer wants to contribute a new diagram layouter, maybe in combination with his or her editor, or for an existing editor, he or she has to register the layouter through the Eclipse extension point mechanism. The layouter is registered for its dedicated editor through the editor's ID, which is a string.

At startup, the `DiagramLayouters` singleton asks for all available layouters on the system and stores an instance of each of them in a map, enabling access to them over the editor ID. When an editor, or viewer, wants to lay out a diagram, it queries the `DiagramLayouters` for a layouter with its ID. The editor receives the instance of the respective layouter and calls, as illustrated in Figure 5.10, the `layout` method with the object to be laid out. If no dedicated diagram layouter for a certain model can be found, then the `KimlGenericDiagramLayouter` will be returned.

KimlAbstractLayouterEngine

This class is responsible to cope with possible hierarchy in the models. In Figure 5.9 it is depicted with *LayoutEngine*. One concrete implementation developed is the `KimlRecursiveGroupLayouterEngine`. As the name foreshadows, this class can work with a hierarchically `KLayoutGraph` in that sense that for every level of hierarchy a new layout provider is called, being responsible for just a portion of the layout.

The `KimlRecursiveGroupLayouterEngine` is also the default engine used at the moment, as layout providers in general have limited abilities when it comes to hierarchy. In the function navigating through the `KLayoutGraph`, the best fitting layout provider is received by a call to the `LayoutProviders` singleton, explained in a paragraph below. As it is imagineable that there will be sometime implementations of layout providers that can handle hierarchy, it is explicitly stated that the recursive engine is just one possibility and developers are free to extend the abstract class on their own. The only method that needs to be implemented is layout, taking the whole `KLayoutGraph` and expecting it to be annotated with layout information on return.

KimlAbstractLayoutProvider

This is the class any concrete layouter, in this thesis called layout provider, due to the several occasions where the term layout is appropriate, has to extend. The most important function is `doLayout(KNodeGroup)`, which is responsible to lay out the sub node groups of the provided `KNodeGroup`. As working currently with the above mentioned recursive implementation of the layout engine, one level of hierarchy is sufficient for the layout providers. However, it remains to be seen if that has to be changed when finishing the Dataflow layouter Spönemann (2009).

Besides, there are some maintenance methods, to query the concrete layout provider about its capabilities and to toggle the status.

KimlNullLayoutProvider

This is a dummy implementation for the abstract `KimlAbstractLayoutProvider` and does simply nothing. Its only purpose is to provide at least one layout provider and to prevent null pointer exceptions, when someone wants to start a layout process and no other concrete layout provider has been implemented by a developer.

More useful implementations are presented in Subsection 5.3.6.

LayoutProviders

This singleton is the analog to `DiagramLayouters` and holds all the instances of registered layout providers. The layout providers must also be announced to Eclipse via extension points. At startup this singleton collects all layout providers and serves then as the manager of them to the rest of the system. The recursive layouter engine, for example, queries for every node group in the layout graph the `LayoutProviders` singleton for the best fitting layouter. This information is stored in the node group itself.

First, the `DiagramLayouters` method searches for a layout provider with the same name as the desired one. If none could be found, then a layout provider capable of the same layout type, which could be *circle*, for example, is searched. If there is still no success, then the user adjustable, globally default layout provider is used. If nothing fits, the above presented `NullLayoutProvider` is returned, avoiding null pointer exceptions, but also doing nothing.

5.3.5. Created Diagram Layouters

Two working diagram layouters, responsible for the two translations, were created in the context of this thesis. A third diagram layouter for Dataflow models is under development (Spönemann, 2009).

The two layouters are the above mentioned `KimlGenericDiagramLayouter`, and the layouter dedicated to the KSSM. In fact there was also another KSSM diagram layouter developed, which can even group parts of a region to be used with one layout provider and not the whole region, but this one was not developed further for this thesis. However, the source code is still existing and may be used and extended to enable a pattern-based layout (Peters, 2008) even within regions. This appears to be an interesting topic, and an example of such a layout can be seen in Section 6.3.

Diagram layouters are independent of the KIML project. They just need to know the `KLayoutGraph`. As most diagram layouters belong to a certain diagram editor, the proposal is to put them into a new Java project, named after their dedicated editor, extended by the suffix *.layouter*.

KimlGenericDiagramLayouter

The `KimlGenericDiagramLayouter` serves, as already mentioned, as the general layouter any editor can fall back to, if no dedicated layouter is available. The generic layouter was first developed for Statecharts, therefore it still does a good job when rendering them. An instance of a simple *Mealy Machine* could also be displayed correctly, as well as UML Statemachines.

When the development of the Dataflow editor started, it became evident that the generic layouter would not be able to perform the task satisfying, due to the above mentioned peculiarities like ports, for example. It turned out that it would be best to have a mechanism as explained above, one generic layouter doing the layout if no dedicated one is present, but the possibility for every editor to ship its own layouter.

KimlSSMDiagramLayouter

A generic layouter can make no assumption on the concrete model it has to lay out. The layouter for the KSSMs was therefore rewritten, now making explicit use of all Statechart specialties known from the model. The only elements, from which the diagram layouter must reconstruct the original model semantics, are node, connection, and label Editparts.

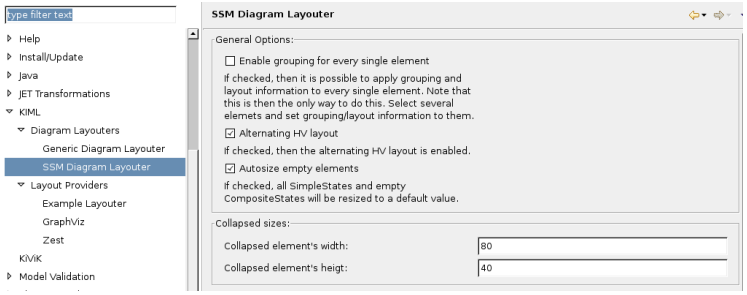


Figure 5.11.: Preference page for the KSSM diagram layouter

Instead, all the newly created layouters can now depend on the concrete editor plug-in and use not only the unspecified Editparts, but the real model elements. With this method, a very straightforward solution is feasible, and a much higher accuracy when translating the model to the `KLayoutGraph` can be achieved, as no guessing is needed.

The implementation walks through the tree structured KSSM model and constructs for every composite state and region new container `KNodeGroups` in the layout graph. Simple states and all the pseudo states are also constituted as `KNodeGroups`, but with no contained elements. Newly constructed nodes are connected by edges as the states were by transitions in the Statechart model. All the other information as labels and priorities is also mapped to their respective elements in the layout graph. The SSM diagram layouter can also take into account whether composite states or

regions have been collapsed in the editor and respect that during the translation into the layout graph and back.

Options, which can be adjusted, are: should an *alternating dot layout* be used, should elements remain at the size as set by the user, and the sizes of collapsed elements.

The re-translation is easy, as mappings are build up during the construction of the layout graph, which map each Statechart element to the corresponding layout graph one.

5.3.6. Created Layout Providers

To compute the actual layout, that are sizes and positions of the objects, layout providers are needed in the presented design, performing the actual task of laying out. Layout providers are typically developed independently from the KIML project and reside in their own project directories. The only thing they need from the KIML environment is the `KLayoutGraph`.

GraphViz layout provider

The (available) layouter of choice for Statecharts is the GraphViz Dot layouter. The framework for using the complete GraphViz suite with its layouters within Java (Kloss, 2005) was used to fit all the GraphViz layouters into the KIML project. This works conveniently under Linux, but problems arise on other platforms, such as Windows.

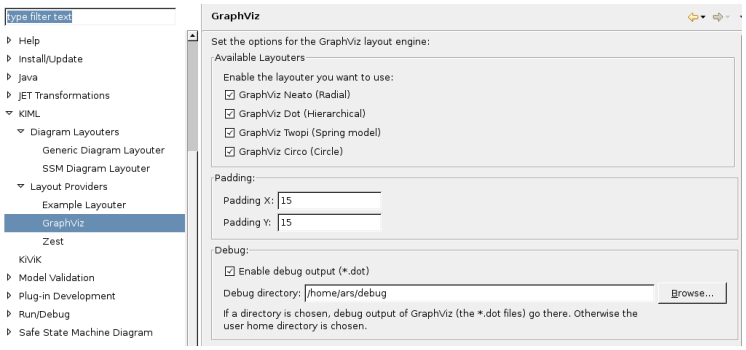


Figure 5.12.: The preference page for the GraphViz layout providers

For every one of the GraphViz layouters, Dot, Circo, Neato, and Twopi, an own class extending `KimlAbstractLayoutProvider` was written, which basically is a wrapper for the calls to `GraphvizLayouter` with the respective layouter name. The class `GraphvizLayouter` does the calls to the GraphViz binary via the Java Native Interface (JNI). The wrapper classes also implement the `getLayouterInfos` method, to provide the environment with the layouter's capabilities, and this function is implemented individually for each of the GraphViz layouters.

With this approach, there were four layout providers available for KIML, being able to generate a hierarchical (Dot), a radial (Neato), a circle (Circo), and a spring embedder (Twopi) layout.

A layout provider has two options to lay out a diagram. The first is to take the layout graph provided and compute the new positions on its own. Second, it can perform another translation, similar to the translation needed when transforming the concrete diagram into the `KLayoutGraph`, namely from the layout graph to the structure the layout provider binary or method understands. After finishing, another translation or annotation of the `KLayoutGraph` with the layout information is needed.

A problem is the graphical representation of edges in GraphViz, as they are drawn as *B-Splines*, making use of *Bézier curves*. The graphical framework used in the editors, GEF, does not understand this representation, but rather works with polygonal chains, called *polylines*. Therefore, a small algorithm in the GraphViz layout provider translates the Bézier curves to a fitting polyline. Within the structure of the `KLayoutGraph`, it can be denoted which representation of a line is used, for example Bézier curves or polylines. But due to GEF's limitations, just polyline is useful at the moment.

Example Layout Provider

This layouter was created to demonstrate the simplicity in creating layout providers for the KIML environment. It is not very powerful, all elements are just laid out in one row, taking no connections into account.

This layout provider operates, in contrast to the previously presented GraphViz ones, directly on the `KLayoutGraph`. No third party library is used. Options like the direction, horizontal or vertical, and padding are adjustable.

Dataflow layout provider

As all existing layout providers did not yield to satisfying results, Spönemann (2009) develops a new layout provider for a Dataflow editor on the basis of the KIML framework and adaptions from different layout algorithms.

5.3.7. User Handling

An important issue about layout is also a convenient interface and handling for the user. Many things are standards when reasoning about input interfaces. That comprises menus, context menus, buttons, and similar facilities. For most of them exist the respective way to implement them in Eclipse, often with extension points, as well as some general Eclipse UI guidelines (The Eclipse Foundation, 2008b).

It has to be noted again that the KIML plug-in is just a subproject of KIELER, therefore has to interact with the overall framework structure KIELER provides. As KIELER itself is also quite at the beginning, and the topic of this thesis does not address KIML solely, the whole UI is created as good as possible, but may seem not

very consistent and will surely change, when the whole project matures. Limitations were for example a missing error handling mechanism for the project, which was introduced just shortly before the end of this thesis.

However, the handling was tried to be implemented as useful as possible. Eclipse has a uniform way to display preference pages, where the user can adjust settings for plug-ins. The strict separation between diagram layouters and layout providers presented above is also reflected in the hierarchy of the respective preference pages. As can be seen in Figure 5.12, under the topic *KIML* in the left tree, there are the nodes *Diagram Layouters* and *Layout Providers*. Every plug-in should place its own preference page into this structure. The figure shown displays the preferences for all four GraphViz layout providers on one page. It is also possible to use a separate one for each GraphViz layout provider, but as there are currently no options that apply to just one layouter, there is no need to do so. To enable a seamless integration of layout providers in the KIML preference pages, an abstract class every layout provider should extend was written, which enables a turning-on and turning-off of the layout providers. This can also be seen it the example figure for the GraphViz layouters. The abstract class takes care of the correct registering of the statuses of the layout providers in the singleton instance `LayoutProviders`.

A preference page for a diagram layouter can be seen in Figure 5.11. Every developer contributing a diagram layouter or layout provider is free to incorporate settings in the respective preference page.

The layout actions can be invoked with a context menu that pops up when right clicking on an element in the editor. An example of it can be seen in Figure 5.13. When firing a new layout request, the old layout of the diagram is transformed into the new one using animation techniques provided by GEF, to help the user understand where and how the elements are moved. Problems with this are addressed in Subsection 5.3.9.

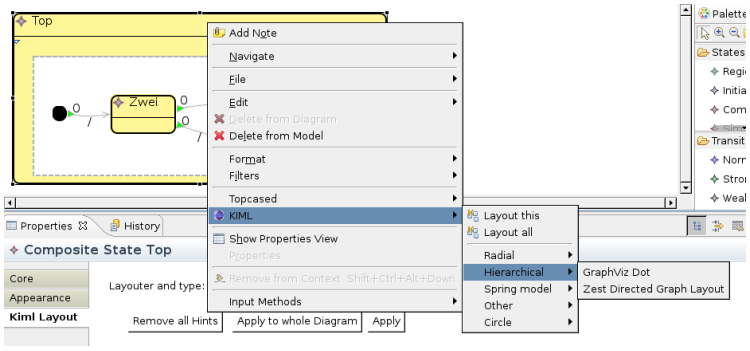


Figure 5.13.: Context menu with some KIML layout providers

The content of the context menu is generated dynamically, depending on the currently available and activated layout providers. The layout providers are grouped by the type of layout they are capable of and the user can thus decide how the selected parts of the diagram should look like after layout.

Another possibility to get information about the current layout of an element and to start a layout process is provided by the properties view. The properties view is a standard facility of Eclipse, displaying all sort of properties of the selected element. The information is grouped together by the type which it belongs to. As the layout of an element is can also be seen as a property, the properties view was chosen to display this information. Additionally, some buttons were implemented to initiate the layout process or to reset all the layout information.

Figure 5.14 shows how the view looks like. The other tabs on the left side, *Core*, *Rulers & Grid* and *Appearance* are standard properties of GMF editor elements. Within the *Core* tab, for example, it is possible to alter attributes of the domain model of the KSSM, the *Appearance* tabs can change settings like the font that is used. The layout tab integrates well in this structure.

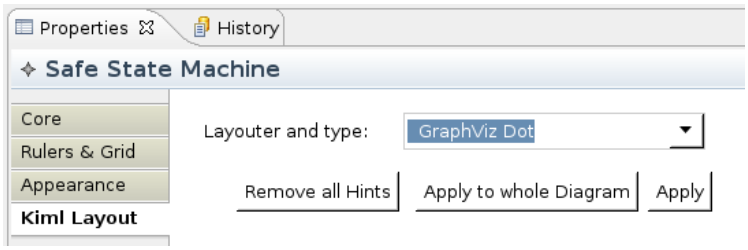


Figure 5.14.: The KIML layout properties view

Another positive aspect is that the layout tab will work with all GMF editors, though no support for GEF is provided. It is not necessary to develop special implementations for every editor, as it was suggested for the diagram layouter, if a sound result is desired. Just an extension point must be used to bind the tab to a diagram editor. However, there are some generic abstract classes provided that each developer can extend, to add special behavior, buttons or options to a tab for a certain editor.

5.3.8. Storing of Layout Information

The ability to use of different layouters for different parts of a diagram, even when the diagram is closed between sessions, requires means for storing the meta layout information. The meta layout information does not specify the concrete positions and sizes of the single elements, they are stored in the notation model of GMF. Meta information relates to the layout type, with which a part should be laid out. More precisely, the meta information consists of a layouter name preferable to use and the layout type.

To store this information, the notation model of GMF is exploited. It allows to store additional information for elements by using an element called `StringValueStyle`. A helper class was created performing exactly the addressed points. Storing and reading out meta layout information from any GMF model through a

wrapper function. Due to the generic approach, these functions can be used by any diagram layouter, as well as any user interface action that wishes to alter the layout.

5.3.9. Problems Encountered

Several problems were faced during implementation, and most of them could be solved. To the problems belonged mainly issues concerning GMF and its specialties, it took a lot of time and investigation to solve them, mainly because of the lack of a proper documentation.

Other things like the Bézier polyline case could be addressed by a simple translation, though it would be nicer to use Bézier curves within GEF. To get a smooth representation of the polyline, which would have just consisted of straight sections otherwise, it is possible to set an option in the preference page for all diagram layouters to *smoothen* the polyline, leading to almost the same shape as the original bézier curve.

A major problem arose from the way GMF handles the resize and move requests. As already mentioned, the single elements are not moved to absolute positions, but by a delta, which has to be computed first. So before moving the elements, all the old positions have to be recorded. Then all the move and resize requests are built during the `applyLayout` method, and at the end the GMF framework issues the concrete moving operations to the viewer of the diagram.

First of all the nodes are moved. Unfortunately, that results in a moving of the attached connections, too. After that, their positions are not the same as the ones recorded before. Nevertheless, their beforehand computed moving delta stays unaltered. Therefore, they are moved to a wrong position. The same applies for the labels of the transitions, which are moved twice, the first time when the nodes force to move the connections, the second time when the connections are moved. This results in a mess.

It is not possible to tell the framework how to move the elements. All commands have to be collected and are executed somewhere deep in the GMF code. A workaround is to force the `applyLayout` method to execute three times programmatically. This is economic, as it is just the applying of layout which is executed multiple times, not the computation thereof. Unfortunately, this results in a bad animation. At the moment, there is an option, to allow the user to decide what he or she prefers. If there are just small changes in a diagram, it is sufficient to use the single layout run.

Summary Layout Plug-in

A modular infrastructure to enable layout for the KIELER framework has been created. With the modular structure of this approach, it is possible to create new layouters for diagrams, as well as layout providers, easily. Clean abstract classes aid new developers when contributing to this concept. The `KLayoutGraph` serves as the link between the editors and the concrete layouters.

The problem of hierarchy was addressed by laying out objects recursively from the leafs to the top of a structure. It turned out that due to specialties of certain diagrams a generic approach is not sufficient, but a generic diagram layouter yields still better results than the integrated ones in Eclipse, so remains available if no dedicated layouter for a diagram type was written.

The meta layout information is stored in the notation model of the diagram. Future versions of KIML and KIELER may use a different approach. A possibility could be to store the `KLayoutGraph` with positional information itself, and omit the notation model, or store several versions of of the layout graph to enable multiple views on one diagram without the need of recomputation.

Around the layout facility several supporting and helper classes were created, and just some of them were presented. Future developers can and should make use of them.

A way for a common representation of preferences and options, as well as actions, was proposed and implemented, leading the way for future contributions.

Several diagram layouters and layout providers were implemented to achieve the final goal, the layout of Statecharts, which can be seen in Section 6.1. But also the claim for to remain generic could be fulfilled, as can be seen in Section 6.2 and through the fact that a Dataflow layouter is on the rise.

Problems arose mainly from peculiarities and inadequacies of GMF. Unfortunately, not all could be solved completely.

One major improvement to existing implementations is the possibility to render parts of a diagram with different layouters.

5.4. The Visual-Diff Plug-in

The other big part of this diploma thesis was to develop a plug-in to enable visual comparison of graphical models. Much preliminary explanations and design considerations have already been given in the previous chapters. Therefore, the constitution of this section will differ slightly from the previous one about the layout.

As exhibited and elaborated in the sections of Chapter 4, the desired representation of the plug-in should work with two panels displaying the diagrams side by side, extended by a third panel presenting a tree view of the structural changes. The diagrams itself should be laid out in their original layout, optionally they could be laid out from scratch, and the changes should be visualized with colors. A handy user interface should aid the user when browsing through the changes.

5.4.1. Utilized Third-Party Projects

To achieve the realization of the visual-diff plug-in, existing third party projects were employed. For diagrams with no initial layout, it is in most cases useful to reorder the objects to get an appealing view. This first goal was obtained by using the newly created layout facilities.

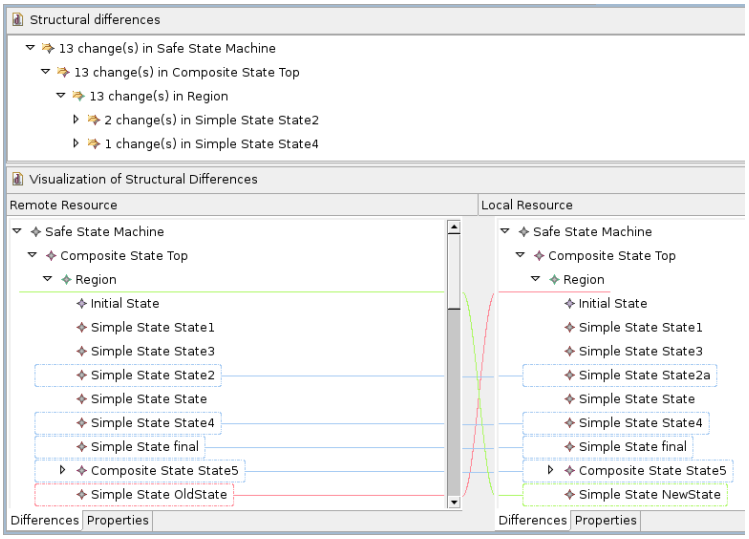


Figure 5.15.: EMF Compare example.

The most important project used was EMF Compare. EMF Compare was not only used to gain the structural differences of the models, but also inspired the visual-diff plug-in in the representation of changes to the user. The concept to display changes of models side by side is very old, and is therefore also used by the standard textual diff of Eclipse. This differences view makes use of special Eclipse classes, `StructureMergeViewer` and `ContentMergeViewer`, which can be integrated seamlessly into the Eclipse framework by using extension points.

EMF Compare takes the same way when displaying the changes to the user. Additionally to the two models, which are presented as structured trees on the left and right, there is another tree at the top, displaying all the changes incurred on the model.

Interesting is the user handling. When clicking on such a change, the lower trees navigate to the model element, which was changed. These elements are marked with special colors, depending on the type of change.

The changes view offers also another tab for each of the models displayed side by side, where it is possible to switch from the tree representation of the model to a view of the properties of the currently selected elements. These tabs can be seen at the bottom of each tree viewer in Figure 5.15.

Emanating from this established concept in Eclipse, it seemed useful not to alter it too much, but to provide means to simply add another tab for the graphical representation of the diagram, also enriched with colors and similar means to navigate through it.

5.4.2. Implementation Concepts

The concept for the implementation was therefore to reuse as much as possible to integrate well into the existing infrastructure. The best concept would have been just to add another tab to the EMF Compare plug-in. Unfortunately, that is not possible.

For this reason, the plug-in had to be created from scratch, but the general architectute was taken from the EMF Compare plug-in. The *Differences* and *Properties* tab are almost the same. The classes building up these tabs have also just been altered slightly to add this new tab.

The *Diagram* tab was developed completely from scratch. During the implementation, some changes had to be done in the previously mentioned classes and other ones reused, as there are some peculiarities of the diagrams that could have not been covered with the existing infrastructure.

When comparing diagrams with EMF Compare, one clicks on the respective files in the *project explorer* on the left and chooses *Compare With* from a context menu. This can be seen in Figure 5.16. As there is a free choice if a generated diagram editor should store the diagram in one file—still maintaining a separation of domain and notation model—or in two files—one file responsible for the domain, the other for the notation model—it is in general possible to compare both of those two files.

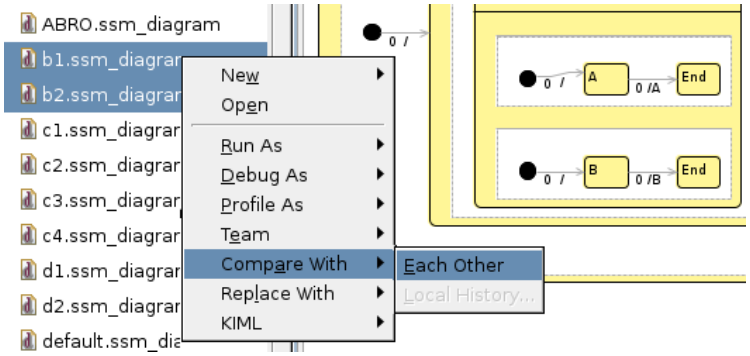


Figure 5.16.: Comparison in Eclipse.

In the one case, the domain model is compared, and there is at first no link to the graphical representation. In the second case, just the layout information is compared, yielding to a completely useless result. To adhere to usability standards, the best way would be to have just one file for domain and notation model and provide the user with a display of strutural as well as graphical changes together. This is how it was done in the implementation. However, it is still possible to compare models that separate their information in two files. Then, the KiViK window is opened when clicking on the files containing the notational information, usually suffixed with *diagram*. The respective domain model needed for the structural comparison is then loaded by KiViK.

A drawback of the approach with two files is that the Eclipse internal mechanism

to fetch the local history of files for separated models does not work. So, when clicking on a local history version of a diagram file, Eclipse is not able to fetch the corresponding domain file of the same age. At the moment, an error message is printed in the console, but this will be changed when making full use of the global error handling. Therefore, editors using just one file work better. To enable this interaction of domain and notation file, another class had to be introduced, as the existing EMF Compare implementation was not designed to handle those special cases.

5.4.3. Packages and Classes in Detail

In this subsection some of the most important classes and their collaboration is presented. Due to the size of the class diagram, it is moved to Appendix B. When implementing the classes, it was tried to use the package and naming convention as already used by EMF Compare.

`ModelContentMergeDiagramTab`

The most relevant contribution was the new tab, which also contains most of the functionality. This class provides the diagram view displayed in the left and right window. To enable the display of the diagrams, this class extends the `DiagramGraphicalViewer`. The interfaces to set the input and what happens when the viewer should display a certain element were adapted to fit the other tabs already provided by EMF Compare.

The input of this viewer originates from the `KivikComparator`. The `ModelContentMergeDiagramTab` takes care of a correct rendering of the diagram and enriches changed elements with appropriate colors. To enable also a coloring of containing elements, and the use of small pop-ups, a further processing of the changes was needed, in contrast to the normal EMF Compare. In EMF Compare those super elements, which possess objects that have been changed, are generally not colored. To achieve a coloring of composite states, for example, whose sub elements have been deleted, the treatment of differences had to be altered slightly. The result is a much clearer representation of changes, as can be seen in Figures 6.4 and 6.6.

The viewer maintains maps, which link the colored elements in the diagram to the elements in the structural tree viewer at the top. Some effort was needed to map all those elements correctly, as the same elements do not have to exist in both viewers, of course, as the can have been deleted or added. In such case, the respective containing element should be used.

Other features the `ModelContentMergeDiagramTab` makes available are the intuitive scrolling and zooming mechanism. If the user wishes, the viewer scrolls automatically to the selected elements in both viewers and in the upper tree viewer, if clicking on the element in any of the three viewers. Was the corresponding element in the other viewer deleted, for example, then it is zoomed to the containing element. This mechanism turned out to be very useful to gain first an overall look of the

changes happened to the diagram, but then to explore the changes in detail.

When clicking on an element in the viewers, a big red box is drawn around it and the corresponding one in the other viewer, to help the user additionally in finding the elements of interest. This can be seen in Figure 6.7, where the region is selected in the structured viewer and the respective graphical elements in the diagrams are marked. This is highly useful in bigger diagrams, when many small states are shown, and it would be hard otherwise to figure out the selected one.

`KivikComparator`

The class `KivikComparator` had to be implemented, as pointed out, to enable the collaboration of the possibly splitted notation and domain models. The main purpose is to fetch the domain information of a diagram when the user clicks on a notational model. The class should use the domain model for comparison, but provide the notational model for the display of the diagram in the viewer, as the user may use the already existing layout information.

On the other hand, if the editor works with one file for both models, this class should also return correctly the needed information. This functionality is wrapped in several functions, so that the calls are transparent for the other classes of the visual-diff plug-in.

The comparison engine of EMF Compare can be extended to serve special needs of certain models. During several tests it turned out that the default implementation is sufficient to compare Statecharts. In Section 6.7 it is pointed out that there are some problems with Dataflow models.

Other Classes

There were several other classes that had to be changed from the EMF Compare project to be used in KiViK. One was for example responsible for exchanging the elements between the three viewers, the two tree viewers at the bottom and the tree viewer at the top displaying the changes. As all elements in these viewers were Java tree items, the data structure exchanging these also used tree items, which did not fit to the diagram elements used in KiViK.

Then there were several content providers for those windows, which made use of the general EMF Compare comparator. As already exposed, the standard comparator was not sufficient and the implementations had to be adapted. Additionally, there were some other minor adjustments, and everything what was not needed here but used in EMF Compare was not incorporated into the code.

Preference Page

To let the user decide what options and help he or she may use, a preference page was created for the KiViK plug-in to enable or disable the settings. As there are also some options which can be adjusted in other plug-ins, due to the reuse of them and

due to a clean concept, there are links in the preference page of the KiViK plug-in enabling the user to directly jump to the desired options in the third-party plug-ins.

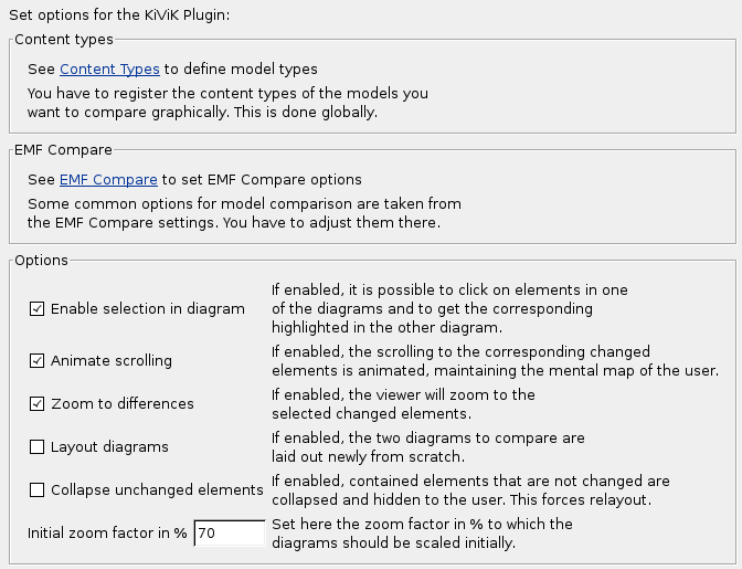


Figure 5.17.: The KiViK preference page

As depicted in Figure 5.17, among those links is one to *Content Types*, where the user defines Eclipse-wide how certain content types are treated, in this case which comparison engine is used. The other link directs to the *EMF Compare* options, where is is possible to adjust the settings of the comparison engine, for example if IDs should be honored, or to change the colors highlighting the differences. The use of the same colors for EMF Compare and KiViK is done on purpose, to gain a consistent look and feel and to help the user to identify the changes the way he or she is already used to from other plug-ins.

Other options cover the ability to select elements in the diagram, if the scrolling to the elements selected should be animated and if a zooming to the selected elements should occur. Furthermore, it is possible to force a new layout of the diagrams before comparison, and it can be chosen if unchanged sub elements should be collapsed, as presented in Figure 6.6. The collapsing forces a new layout of the diagram and the respective diagram layouter, responsible for the translation to the `KLayoutGraph`, should be able to do this transformation of the hidden objects correctly.

The last option, setting the initial zoom factor, is more or less a workaround. It is not possible upon loading of a diagram viewer to set the diagram itself to a zoom factor such that it fits entirely into the window, as at the time of rendering the diagram the viewer window itself has not been created completely. Due to that, one can adjust the initial zoom factor. However, if zooming is enabled, then, after clicking on changed elements in the diagram, the zoom is automatically adjusted.

5.4.4. User Interface

Most of the questions concerning the user interface were addressed by adhering to established Eclipse concepts, like context menus, layout of comparison panels, preference pages, just to mention a few. Comparing diagrams is just simple. Once the diagram extension has been registered to use the KiViK plug-in as the comparison facility through the context menu, selecting the two elements in the project explorer and clicking on compare in the context menu is enough.

Unfortunately, there is currently in bug[6] in the compare framework of Eclipse used by KiViK, which does not honor the order in which the files are selected in the explorer. Currently, the files are lexicographically ordered, which means that the file first in the alphabet is treated as the older file, and the other one as the younger. This is certainly not what is desired, as from this setting it depends how the *addings* and *deletions* are computed.

The comparison of history items is not affected by this, but as already pointed out, this is not working when a model consists of two separate files.

Currently, there are no means implemented to merge the diagrams, which would be very convenient for the user, but this should not be to complicated, as the normal EMF Compare does already quite a good job in this task. However, a sound graphical representation of the merged elements in a diagram would be harder to realize.

Summary Visual-Diff Plug-in

An infrastructure was created, according to the considerations in Chapter 4, which enables the user to compare diagrams in the same notational power like they present the information, graphically. It was tried to adhere to existing projects as much as possible, to get a tight integration into the target platform, Eclipse. The approach is as generic as possible, resulting in a plug-in not only able to compare SSMs, as requested, but to a certain extend any model created with GMF. The benefit of such a method was already pointed out and can be seen in the next chapter.

[6] See: https://bugs.eclipse.org/bugs/show_bug.cgi?id=83970.

"Aufgabe von Kunst heute ist es, Chaos in die Ordnung zu bringen."
Theodor W. Adorno - Minima Moralia, III, 143

6 Case Studies

This chapter presents seven case-studies showing the benefits of the developed plug-ins. The first three examples address the layout plug-in, the latter four deal with visual comparison. The examples are not limited to Statecharts.

6.1. Case study 1: Layout in General

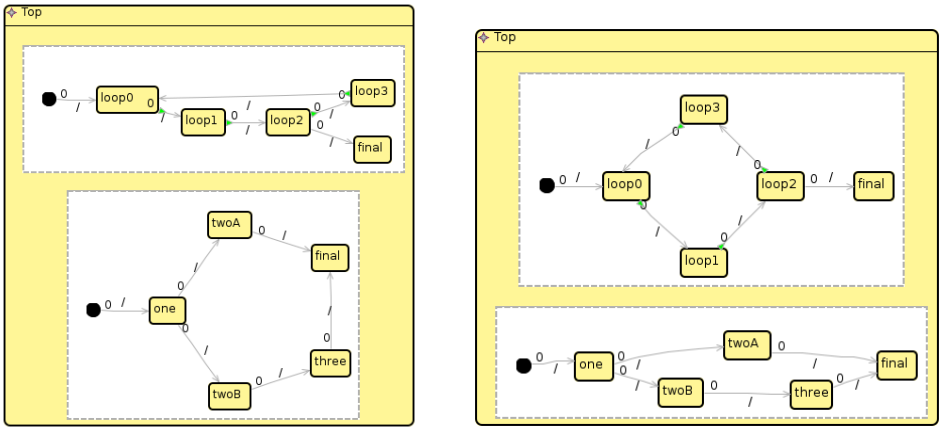
This first example addresses an innovation of the layout plug-in. In KIEL, for example, but also in many other tools, it is just feasible to use one type of layout algorithm for all the parts of a diagram. Because of the modular design of KIML, it is now possible to apply different layout types to parts of a Statechart.

The initial impact were considerations about secondary notation and pattern-based layout (Peters, 2008), as already addressed in the previous chapters. A good choice of the layout can help the user to understand the semantics of an SSM. And sometimes, it is more helpful if not the whole Statechart is rendered by the same algorithm, but if certain parts use a dedicated one.

Figure 6.1 shows the same Statechart, laid out twice with different algorithms. The respective upper and lower regions of the two versions shown correspond to each other. In the upper region a loop is modeled (see also Peters, 2008, Sec. 4.2.2), whereas in the lower region no special behavior is shown.

In the left Statechart the two regions are modeled with algorithms that obfuscate the semantics of the diagram. The loop is drawn horizontally, the lower region seems to describe a loop construct, though that is not the case. Yet the layout itself is appealing, though the application is bad in this special case.

Normally, that means in most current diagram editors, the two regions on the left side would of course have been rendered with the same layouter, not with distinct ones like here, where GraphViz Dot is responsible for the upper, and GraphViz Circo



(a) SSM with bad layout

(b) SSM with good layout

Figure 6.1.: Confrontation of badly and wisely used layouts

for the lower part. That is just done here to save some space instead of introducing a third diagram.

On the right side, where it is actually possible to assign different layouters to the regions, one is free to choose the desired layout algorithm. This has to be done manually in the current version of the KIML plug-in, but one can image an algorithm choosing the appropriate layout for certain parts of a Statechart—or any other diagram—automatically. The layouters chosen for the right region stress with their positioning of the states the meaning of the Statechart.

To develop such an algorithm or class of algorithms—using several strategies for finding a good layout—and implement it in the KIELER and KIML environment seems to be a valuable task.

6.2. Case study 2: Layout of UML Statemachines

Though initially just Statecharts were supposed to be laid out, even more is possible. The concept described in the previous chapter states that every editor needs its own diagram layouter to perform a reasonable layout of the graphical elements. Normally, the editor developer should provide this layouter, as it was done for the Statechart and also for the Dataflow editor. However, for those layouters already available, where nobody has created such a layouter, there exists still the generic diagram layouter.

Though this all purpose layouter is not optimized for the specialties of certain diagrams as the SSM layouter is, for example, it may yield satisfying results. Provided with the modeling edition of Eclipse installations are the UML tools (The Eclipse Foundation, 2008a). They exhibit amongst others a UML Statemachine editor. An example Statemachine was created and laid out with the generic layouter. As the

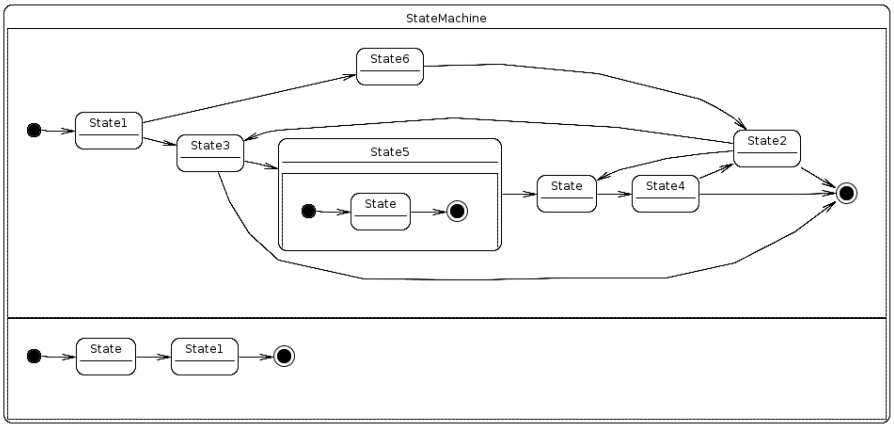


Figure 6.2.: Generic diagram layouter applied to a UML Statemachine, using the GraphViz Dot layout provider.

result in Figure 6.2 shows, the layout is fairly well done. The parameter *inset top* of the generic layouter had to be set to 20 to receive this layout. The alignment of regions in the Statemachine implementation for Eclipse is *list layout*, the reason why every region has the same width.

6.3. Case study 3: Individually Grouped Elements Inside Compartments

Although the approach to group elements individually within one compartment was not further followed in this thesis, the first steps were undertaken to enable it. But during writing of this work, one problem came up, which could not easily be solved: transitions, or in general, when working with models other than Statecharts, connections between grouped elements.

Imagine a connection between elements of two selected groups in the center region. The two groups are laid out with dedicated algorithms, as can be seen in the *Kiml Layout Hints* view on the right in Figure 6.3. The upper part uses the GraphViz Neato layouter, the lower one the Dot layouter. This works well, if the groups are unconnected. But if a link is present between the two clusters, it is very likely that this link will interfere the previously computed layout of the single groups, as it is possible that the center element of the radially laid out, and, for example, a lower element of the bottom group is connected. Several constraints would be violated. The computation of the single layouts must have taken into account the link before. That was beyond the scope of this work. Nevertheless, it is an interesting feature, as already mentioned, and when talking about pattern-based layout a very promising approach.

The result presented in Figure 6.3 was achieved with a special version of the `KimlSSMDiagramLayouter`, the `KimlSSMDiagramGroupingLayouter`, which

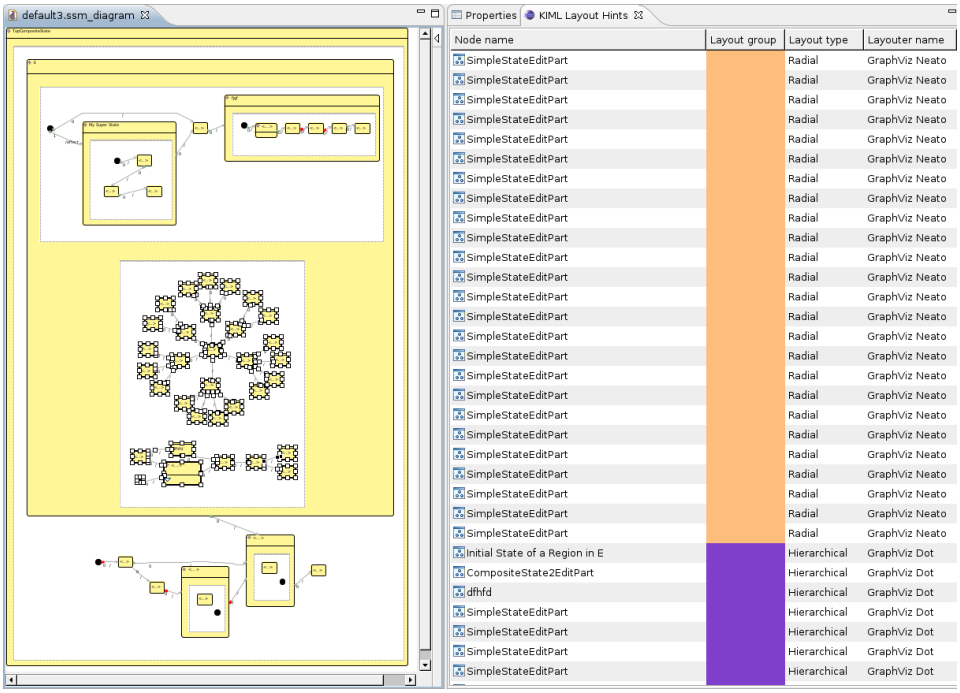


Figure 6.3.: Example of individually grouped elements in a KSSM

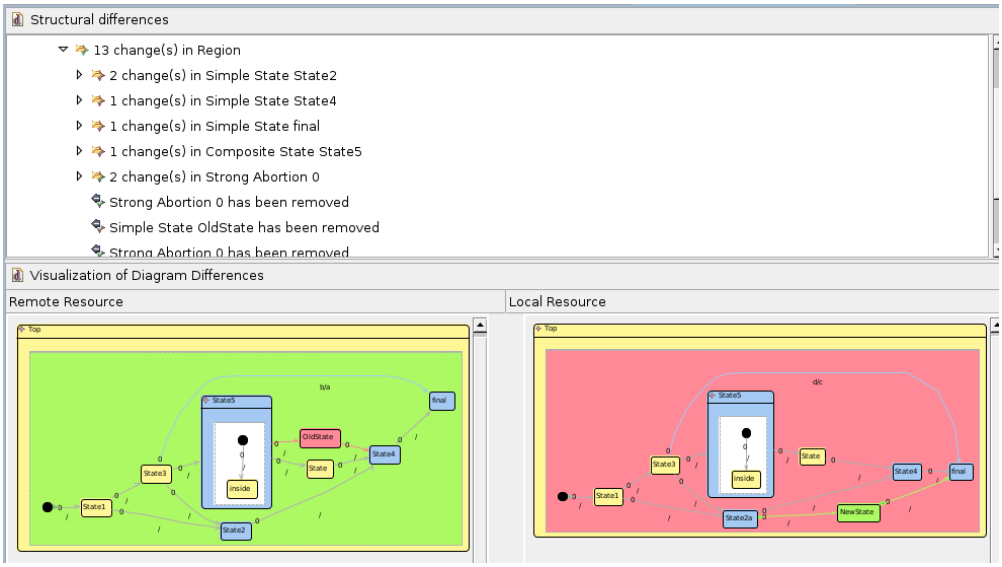
inserts for every group inside of a compartment, region in the SSM case, a new `KNodeGroup` containing the grouped elements. When applying the layout information back to the diagram, the respective offset has to be taken into account, as for example the lower group of the two in the region must be moved below the radial one.

To aid the user when selecting and grouping elements, the *Kiml Layout Hints* view was developed. This view has now been replaced in the default implementation by the generic properties view, which can be seen in Figure 5.14. With the implementation enabling any element to have its own layout information, instead of just the compartments, like it is done in the current design, the user must have instruments to check and set layout information for the elements. This purpose serves the special view. To see at a glance which elements belong to a group, objects sharing a group are colored equally. Clicking on a single state or a group of states enables the same context menu as clicking on the states themselves in the diagram. The actions presented are similar to those described before, as in the actual plug-in. Another feature is to select one state and to let the view highlight all states in the same layout group.

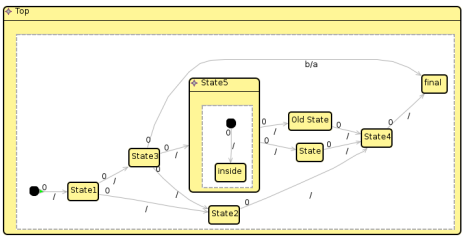
6.4. Case study 4: Comparison of KIELER Statemachines

The first case study presenting the ability of the KiViK plug-in will be performed with the model it is intended for, SSMs. To support the understanding of the example, beside the pure comparison view the two source diagrams are also displayed, see

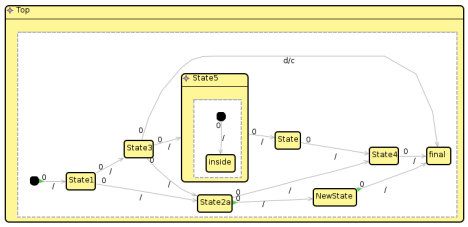
Figure 6.4.



(a) KiViK compare window



(b) SSM version 1



(c) SSM version 2

Figure 6.4.: KiViK comparison window and the two source SSMs, the models are the same as in Figure 5.15

Both diagrams have been auto-laid out with the developed layout plug-in. That is the reason for the placement of *New State* in version 2, which is placed at the original position of a transition. The operations applied to version 1 of the SSM were as follows:

1. Rename simple state *State2* to *State2a*
2. Delete simple state *OldState* and its 2 transitions
3. Add simple state *NewState*
4. Add transition from *State2a* to *NewState*
5. Add transition from *NewState* to *final*

6. Change label of the transition from *State3* to *final* from *b/a* to *d/c*

Though that are exactly six editing operations, the plug-in denotes 13 changes. That is due to the fact that the transitions also belong to the states they are connected to. Therefore, for every existing state to which a transition is added, this is also recorded as a change. The addition of the *NewState* with its two transitions remains just one editing operation. Furthermore, the altering of trigger and effect of the one transition is denoted as two changes, as they are two functional distinct parts of the transition.

This example shows a typical use case when working with SSMs. New elements are added and existing ones altered or deleted. When comparing the two original diagrams without any aid, at least the adding of the *NewState* and deletion of *OldState* catches one's eye. The other changes are too subtle to be recognized immediately. That is where the plug-in can help. The change of the trigger and effect of the transition or the renaming of *State2* are found much faster, than with no aid; if these changes would have been found at all otherwise.

The coloring should help the user to orientate her- or himself. The addings and deletions are quickly found. If there is a deletion and an addition, then the deletion supersedes the addition in the actual, or *local resource*, as denoted in the header of the right part. In the left part, denoted *remote resource*, as the comparison was against the local history of that model, the addition supersedes the deletion. Therefore the region in the left part is colored green, whereas it is colored red in the right part, though there are additions and deletions.

The difference, or the benefit, can be seen when comparing this to the original compare, as displayed in Figure 5.15.

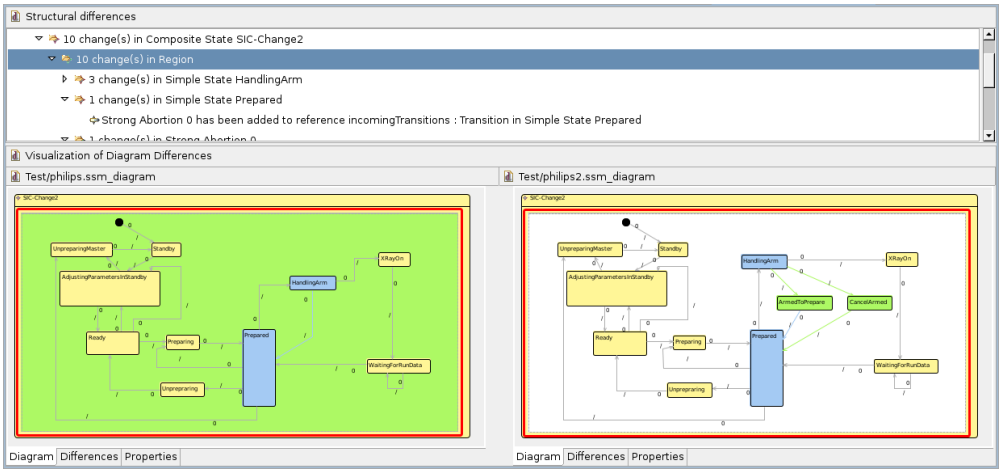
In Figure 6.5 a real model used in the industry is shown. Though semantics and syntax differ considerably—the original diagrams are Rational Rose capsules, the used diagrams in the comparison are similarly hand-modeled SSMs—the advantage of the visual comparison in KiViK can be seen easily.

One has to search at least some time in the two capsules to find the region of interest, where the changes actually happened. In KiViK the user can see this directly, as the colors help him or her. Another aspect is that in Rational Rose the two diagrams have to be opened by hand and positioned side by side, as illustrated in the figure. This is done automatically by KiViK.

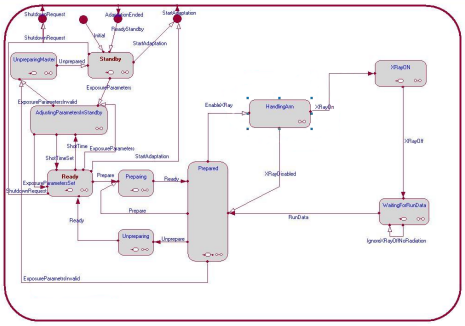
6.5. Case study 5: Comparison of KIELER Statemachines with Collapsing

This example demonstrates the benefit of the collapsing option. When collapsing is enabled, all compartment elements that do not have any changes in lower levels of the hierarchy hide their sub elements, yielding to a more stringent view of the changes. Just the parts of the developer's interest are shown.

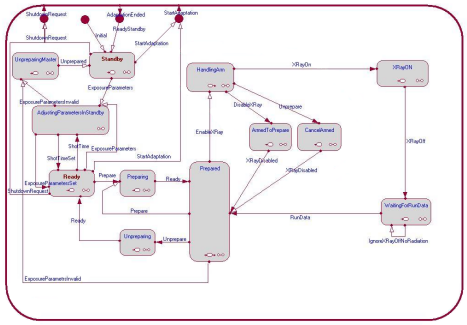
In Figure 6.6 the difference can easily be seen. Even though this is just a small example, the collapsed view is more appealing an points out what is important,



(a) KiViK compare window



(b) Rational Rose capsule, version 1



(c) Rational Rose capsule, version 2

Figure 6.5.: KiViK comparison window and the two original Rational Rose capsules.

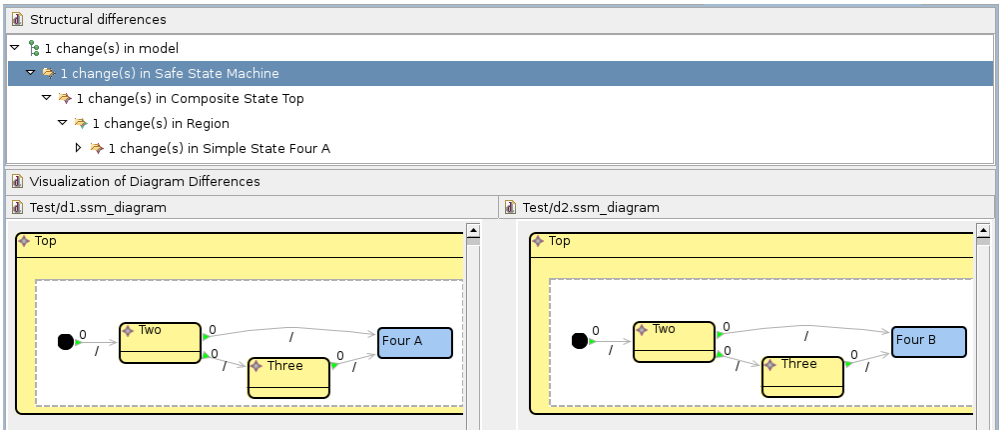
namely the renaming of the state *Four A*. With this mechanism navigating through changes in the diagram also becomes handier, if the changes are spatial and far away from each other.

Collapsing of elements and still providing an appealing view demands a new layout of the Statechart. This was performed automatically by the developed layout plug-in for this example.

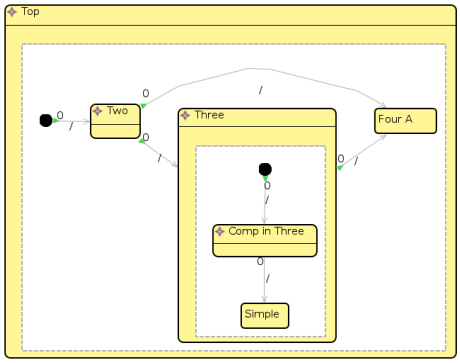
6.6. Case study 6: Comparison of UML Statemachines

What was said for the layout in Section 6.2 also applies to the KiViK plug-in. Initially it was supposed to compare SSMs. But as this plug-in exploits facilities of EMF, GEF and GMF, leading to a uniform method how elements are represented, both in the domain and notation model, comparisons of arbitrary models are possible, as long as the editor is generated with GMF.

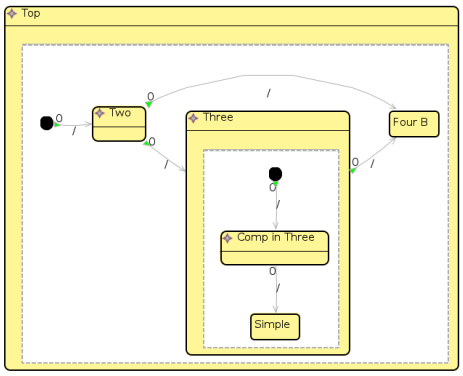
This applies also to the previously mentioned UML Statemachine editor for Eclipse.



(a) KiViK compare window



(b) SSM version 1



(c) SSM version 2

Figure 6.6.: KiViK comparison window and the two source SSMs, demonstrating collapsing

The comparison shown in Figure 6.7 was performed without altering anything in the plug-in. Just the file extension **.umlstm* had to be registered to enable KiViK to perform the comparison, instead of the default comparison engine.

6.7. Case study 7: Comparison of Dataflow Models

To show the power of the comparison plug-in, another model type was used to test it. The alpha version of the Dataflow editor for Eclipse—developed at this research group—was used. As the layouter for this editor is likewise still under development, the boxes and connections were arranged by hand.

Difficulties a layouter for Dataflow models has to face are the ports, which are used to connect the single boxes. The ports also turned out to influence the comparison engine, as they are another semantic element lying between a state and a connection.

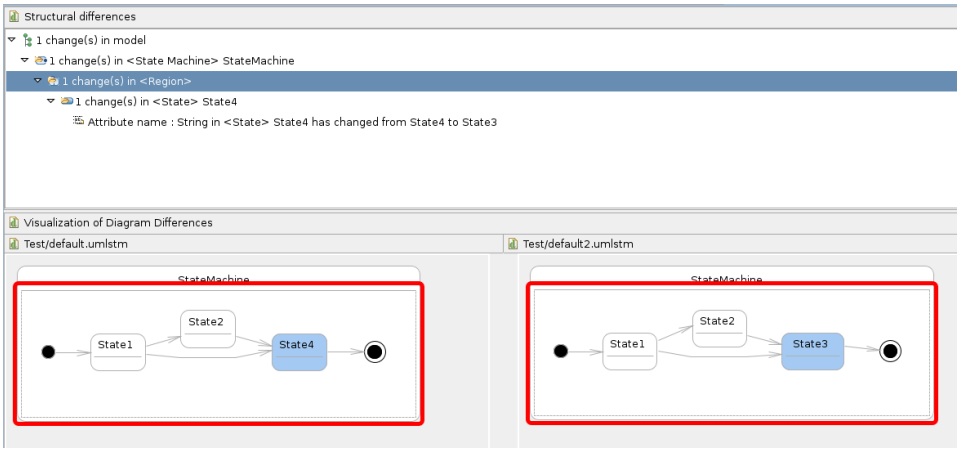


Figure 6.7.: Seamless comparison of UML Statemachines with KiViK

Some effort will have to be made to extent the comparison plug-in to cope with port specialties.

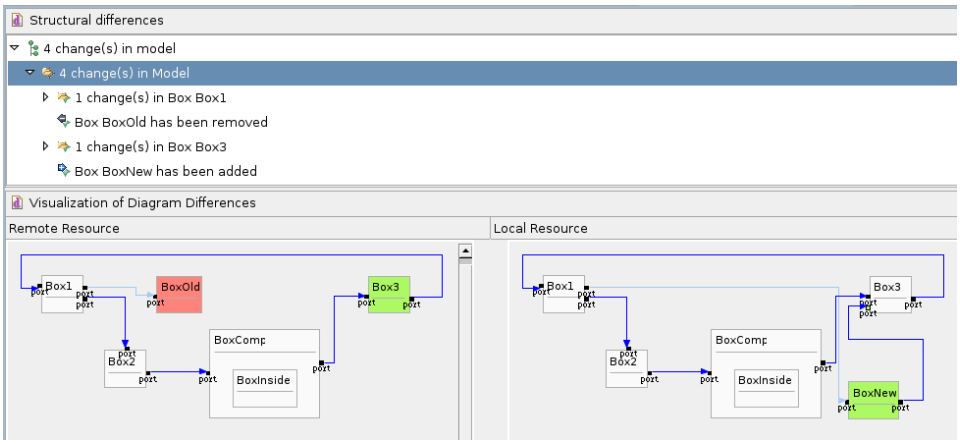


Figure 6.8.: Comparison of Dataflow models with KiViK

For the example in Figure 6.8 a relatively simple model was chosen. The result is overall quite satisfying, but one has to be careful with the interpretation. The removal of *BoxOld* and addition of *BoxNew* is recognized correctly. What looks odd, at least at the first glance, is the green color for *Box3*. Nothing was added to the Box, just a connection. In the SSM case, the respective state would have been marked blue as changed, but not green for added. That follows from the fact that a transition in Statecharts is modeled as a reference of the state. Changing—as well as adding and deleting—of references yields to *changed* in EMF Compare at the end of the comparison process. As ports are elements of the box itself in the Dataflow model, adding a new connection implies adding a port to a box. This port is now

recognized as a new element and the box is marked green. One may argue which of those two behaviors is correct, or better for the user. Maybe marking a box green is adequate, as something, the port, really has been added. On the other hand it could be convenient to have the same characteristics in both worlds, the Dataflow and Statechart one. As comparison of other models than Statecharts was not the topic of this thesis, a further examination of this question is left out, but seems generally worth a broader investigation.

Summary

This chapter presented some examples for application areas of the developed layout and visual comparison plug-ins and demonstrated also their strength with other models than Statecharts. Future enhancement possibilities and other areas of application were pointed out as well as some caveats.

"There is no excellent beauty that hath not some strangeness in the proportion."

Francis Bacon - Of Beauty

7 Conclusion

Results and contributions of this thesis as well as an outlook will be given in this chapter.

7.1. Results and Contribution

The thesis leads to several results. First of all, a working framework enabling (meta) layout facilities for the KIML and KIELER infrastructure was created. This was completed by an SSM editor in GMF, which was needed for this thesis to provide the Statechart models which should have been laid out. The Statecharts model itself was used later for other theses. The usefulness and power of the developed layout plug-in can be seen the Chapter 6, as well as in the further theses centering around KIELER, of which one concentrates on a Dataflow layouter (Spönemann, 2009).

The generated SSM model and editor lead a basis towards a *real* implementation within the KIELER meta-modeling tool. As mentioned, this editor was just developed to that stage that a layout was possible with it. However, the model itself was developed according to the semantics of SSM (André, 2003).

The KIML infrastructure turned out to be very useful when during the writing of this thesis several layouters were added. Among those layouters were some for editors, translating the diagram into the layout graph, as well as layout providers, doing the actual computation.

Another interesting and important point that was discovered relates to diagrams and how they can be translated to a generic structure. Though no *proof* is given, it turned out during the work on the concrete diagram layouters responsbile for the conversion of the Editparts to the `KLayoutGraph` that a generic approach is not feasible. The reason was the different semantics of the respective models, which had to be represented by common entities—Editparts—in GEF. A meaningful re-translation

from the Editparts back to the original semantics into the `KLayoutGraph` was impossible.

The developed plug-in to visualize differences seems to be the first one to use the approach with colored diagrams side by side, supported by means which display the changes structurally in the same comparison window and enable a sound navigating and zooming.

During a presentation of the results for the industry, the meta layout—different algorithms for different parts—and the visual comparison was found to be very useful in the daily work of developers. Further suggestions from this demonstration are presented in the next section.

7.2. Outlook and Future Research

To provide facilities to compare diagrams graphically is just a logical consequence of the increase of models being used in several academic and industrial fields. A stronger attention will be drawn on this area during the next years, as the demand for such methods is high, but yet very little research has been done, and even less has been transferred to academic and commercial tools.

The presented comparison tool is just a first step, the next stage would be to implement in the same manner views that support the users graphically when merging changes into diagrams. Then, there is definitely the need of a well-working incremental layout algorithm.

As it became evident during talks with developers using models, mechanisms to apply patterns to certain parts of a diagram would help considerably during the work with them. This relates to the presented grouping diagram layouter for SSMs, where even parts of a region could be forced to use a dedicated layout algorithm. The demand of the developers was not for a special type of algorithm like spring-embedder or circle, but to apply certain patterns, like *loop*, *if-then-else*, or some *error-handling* to parts of a Statemachine. These groups could be laid out with a small, specially dedicated algorithm with fixed positions for the elements. Still, the problem with transitions between independently laid out groups remains.

As it has not been researched which kind of representation of changes in the visual way is best, the framework could also be used to leverage such a comparison, to get empirical information thereof. It would also be interesting to investigate how the presented approach works out with other diagrams than Statecharts, or if in that case another method would yield better results.

A

Example of a Simple KSSM

A simple example of a KSSM is presented. First the graphical representation is given, then the domain model, finally the notation model.

A.1. Diagram Respresentation

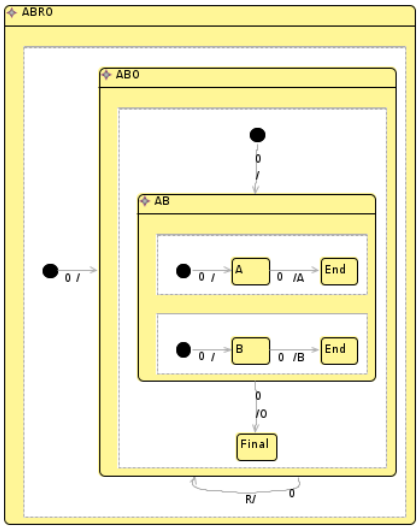


Figure A.1.: Graphical representation of a simple KSSM model

A.2. KSSM Domain Model

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:notation="
       http://www.eclipse.org/gmf/runtime/1.0.1/notation" xmlns:ssm="http://www.
       informatik.uni-kiel.de/rtsys/ssm">
3    <ssm:SafeStateMachine xmi:id="_NkK3gMbCEd28KratHji1uw" name="AB">
4      <top xmi:type="ssm:CompositeState" xmi:id="_ORi64MbCEd28KratHji1uw" name="ABRO
           ">
5        <regions xmi:type="ssm:Region" xmi:id="_QQZ0QMbCEd28KratHji1uw">
6          <states xmi:type="ssm:InitialState" xmi:id="_SCVE4MbCEd28KratHji1uw"
               outgoingTransitions="_TWuE8MbCEd28KratHji1uw"/>
7          <states xmi:type="ssm:CompositeState" xmi:id="_Si40QMbCEd28KratHji1uw"
               incomingTransitions="_TWuE8MbCEd28KratHji1uw _hgrCMMbCEd28KratHji1uw"
               outgoingTransitions="_hgrCMMbCEd28KratHji1uw" name="ABO">
8            <regions xmi:type="ssm:Region" xmi:id="_WAyegMbCEd28KratHji1uw">
9              <states xmi:type="ssm:InitialState" xmi:id="_W0qLcMbCEd28KratHji1uw"
                   outgoingTransitions="_tD500MbCEd28KratHji1uw"/>
10             <states xmi:type="ssm:CompositeState" xmi:id="_sBLzkMbCEd28KratHji1uw"
                    incomingTransitions="_tD500MbCEd28KratHji1uw" outgoingTransitions
                   ="_KTjl4MbDEd28KratHji1uw" name="AB">
11               <regions xmi:type="ssm:Region" xmi:id="_uMB6gMbCEd28KratHji1uw">
12                 <states xmi:type="ssm:InitialState" xmi:id="
                       _wUNvAMbCEd28KratHji1uw" outgoingTransitions="
                       _0P1QAMbCEd28KratHji1uw"/>
13                 <states xmi:type="ssm:SimpleState" xmi:id="_wbbKQMbCEd28KratHji1uw
                       " incomingTransitions="_0P1QAMbCEd28KratHji1uw"
                       outgoingTransitions="_0hIbYMbCEd28KratHji1uw" name="A"/>
14                 <states xmi:type="ssm:SimpleState" xmi:id="_we8_YMbCEd28KratHji1uw
                       " incomingTransitions="_0hIbYMbCEd28KratHji1uw" name="End"/>
15                 <transitions xmi:type="ssm:StrongAbortion" xmi:id="
                       _0P1QAMbCEd28KratHji1uw" target="_wbbKQMbCEd28KratHji1uw"
                       source="_wUNvAMbCEd28KratHji1uw"/>
16                 <transitions xmi:type="ssm:StrongAbortion" xmi:id="
                       _0hIbYMbCEd28KratHji1uw" target="_we8_YMbCEd28KratHji1uw"
                       source="_wbbKQMbCEd28KratHji1uw" effectString="A"/>
17               </regions>
18               <regions xmi:type="ssm:Region" xmi:id="_ubjkQMbCEd28KratHji1uw">
19                 <states xmi:type="ssm:InitialState" xmi:id="
                       _wy8SYMbCEd28KratHji1uw" outgoingTransitions="
                       _0zIKUMbCEd28KratHji1uw"/>
20                 <states xmi:type="ssm:SimpleState" xmi:id="_w54A0MbCEd28KratHji1uw
                       " incomingTransitions="_0zIKUMbCEd28KratHji1uw"
                       outgoingTransitions="_1EQ9oMbCEd28KratHji1uw" name="B"/>
21                 <states xmi:type="ssm:SimpleState" xmi:id="_w-xu8MbCEd28KratHji1uw
                       " incomingTransitions="_1EQ9oMbCEd28KratHji1uw" name="End"/>
22                 <transitions xmi:type="ssm:StrongAbortion" xmi:id="
                       _0zIKUMbCEd28KratHji1uw" target="_w54A0MbCEd28KratHji1uw"
                       source="_wy8SYMbCEd28KratHji1uw"/>
23                 <transitions xmi:type="ssm:StrongAbortion" xmi:id="
                       _1EQ9oMbCEd28KratHji1uw" target="_w-xu8MbCEd28KratHji1uw"
                       source="_w54A0MbCEd28KratHji1uw" effectString="B"/>
24               </regions>
25             </states>
26             <states xmi:type="ssm:SimpleState" xmi:id="_IXfh0MbDEd28KratHji1uw"
                   incomingTransitions="_KTjl4MbDEd28KratHji1uw" name="Final"/>
27             <transitions xmi:type="ssm:StrongAbortion" xmi:id="
                   _tD500MbCEd28KratHji1uw" target="_sBLzkMbCEd28KratHji1uw" source="
                   _W0qLcMbCEd28KratHji1uw"/>
28             <transitions xmi:type="ssm:StrongAbortion" xmi:id="
                   _KTjl4MbDEd28KratHji1uw" target="_IXfh0MbDEd28KratHji1uw" source="
```

```
                _sBLzkMbCEd28KratHji1uw" effectString="0"/>
29            </regions>
30          </states>
31          <transitions xmi:type="ssm:StrongAbortion" xmi:id="_TWuE8MbCEd28KratHji1uw
                " target="_Si40QMbCEd28KratHji1uw" source="_SCVE4MbCEd28KratHji1uw"/>
32          <transitions xmi:type="ssm:StrongAbortion" xmi:id="_hgrCMMbCEd28KratHji1uw
                " target="_Si40QMbCEd28KratHji1uw" source="_Si40QMbCEd28KratHji1uw"
                triggerString="R"/>
33        </regions>
34      </top>
35    </ssm:SafeStateMachine>
36  ...
```

A.3. KSSM Notation Model (Cutout of Line 1 - 50)

```
1   ...
2     <notation:Diagram xmi:id="_Nk6eYMbCEd28KratHji1uw" type="Safe State Machine"
          element="_NkK3gMbCEd28KratHji1uw" name="thesis.ssm_diagram" measurementUnit=
          "Pixel">
3       <children xmi:type="notation:Node" xmi:id="_ORpBgMbCEd28KratHji1uw" type="2001
            " element="_ORi64MbCEd28KratHji1uw">
4         <children xmi:type="notation:Node" xmi:id="_OR0AoMbCEd28KratHji1uw" type="
              5003"/>
5         <children xmi:type="notation:Node" xmi:id="_OR110MbCEd28KratHji1uw" type="
              7001">
6           <children xmi:type="notation:Node" xmi:id="_QQabUMbCEd28KratHji1uw" type="
                3001" element="_QQZ0QMbCEd28KratHji1uw">
7             <children xmi:type="notation:Node" xmi:id="_QQc3kMbCEd28KratHji1uw" type
                  ="7002">
8               <children xmi:type="notation:Node" xmi:id="_SCW6EMbCEd28KratHji1uw"
                    type="3004" element="_SCVE4MbCEd28KratHji1uw">
9                 <styles xmi:type="notation:ShapeStyle" xmi:id="
                      _SCW6EcbCEd28KratHji1uw" fontName="Sans"/>
10                <layoutConstraint xmi:type="notation:Bounds" xmi:id="
                      _SCW6EsbCEd28KratHji1uw" x="16" y="179" width="13"/>
11              </children>
12              <children xmi:type="notation:Node" xmi:id="_Si6CYMbCEd28KratHji1uw"
                    type="3002" element="_Si40QMbCEd28KratHji1uw">
13                <children xmi:type="notation:Node" xmi:id="_Si73kMbCEd28KratHji1uw"
                      type="5001"/>
14                <children xmi:type="notation:Node" xmi:id="_SjDMUMbCEd28KratHji1uw"
                      type="7003">
15                  <children xmi:type="notation:Node" xmi:id="_WAzFkMbCEd28KratHji1uw
                        " type="3001" element="_WAyegMbCEd28KratHji1uw">
16                    <children xmi:type="notation:Node" xmi:id="
                          _WAzsoMbCEd28KratHji1uw" type="7002">
17                      <children xmi:type="notation:Node" xmi:id="
                            _W0rZkMbCEd28KratHji1uw" type="3004" element="
                            _W0qLcMbCEd28KratHji1uw">
18                        <styles xmi:type="notation:ShapeStyle" xmi:id="
                              _W0rZkcbCEd28KratHji1uw" fontName="Sans"/>
19                        <layoutConstraint xmi:type="notation:Bounds" xmi:id="
                              _W0rZksbCEd28KratHji1uw" x="128" y="15" width="13"/>
20                      </children>
21                      <children xmi:type="notation:Node" xmi:id="
                            _sBLzkcbCEd28KratHji1uw" type="3002" element="
                            _sBLzkMbCEd28KratHji1uw">
22                        <children xmi:type="notation:Node" xmi:id="
                              _sBMao8bCEd28KratHji1uw" type="5001"/>
23                        <children xmi:type="notation:Node" xmi:id="
                              _sBMapMbCEd28KratHji1uw" type="7003">
```

```
24                          <children xmi:type="notation:Node" xmi:id="
                                _uMChkMbCEd28KratHji1uw" type="3001" element="
                                _uMB6gMbCEd28KratHji1uw">
25                            <children xmi:type="notation:Node" xmi:id="
                                  _uMChk8bCEd28KratHji1uw" type="7002">
26                              <children xmi:type="notation:Node" xmi:id="
                                    _wUO9IMbCEd28KratHji1uw" type="3004" element="
                                    _wUNvAMbCEd28KratHji1uw">
27                                <styles xmi:type="notation:ShapeStyle" xmi:id="
                                      _wUO9IcbCEd28KratHji1uw" fontName="Sans"/>
28                                <layoutConstraint xmi:type="notation:Bounds" xmi:id=
                                      "_wUO9IsbCEd28KratHji1uw" x="16" y="25" width="
                                      13"/>
29                              </children>
30                              <children xmi:type="notation:Node" xmi:id="
                                    _wbcYYMbCEd28KratHji1uw" type="3003" element="
                                    _wbbKQMbCEd28KratHji1uw">
31                                <children xmi:type="notation:Node" xmi:id="
                                      _wbcYY8bCEd28KratHji1uw" type="5002"/>
32                                <styles xmi:type="notation:ShapeStyle" xmi:id="
                                      _wbcYYcbCEd28KratHji1uw" fontName="Sans"/>
33                                <layoutConstraint xmi:type="notation:Bounds" xmi:id=
                                      "_wbcYYsbCEd28KratHji1uw" x="73" y="18"/>
34                              </children>
35                              <children xmi:type="notation:Node" xmi:id="
                                    _we9mcMbCEd28KratHji1uw" type="3003" element="
                                    _we8_YMbCEd28KratHji1uw">
36                                <children xmi:type="notation:Node" xmi:id="_we-
                                      NgMbCEd28KratHji1uw" type="5002"/>
37                                <styles xmi:type="notation:ShapeStyle" xmi:id="
                                      _we9mccbCEd28KratHji1uw" fontName="Sans"/>
38                                <layoutConstraint xmi:type="notation:Bounds" xmi:id=
                                      "_we9mcsbCEd28KratHji1uw" x="155" y="18"/>
39                              </children>
40                              <styles xmi:type="notation:DrawerStyle" xmi:id="
                                    _uMChlMbCEd28KratHji1uw"/>
41                              <styles xmi:type="notation:SortingStyle" xmi:id="
                                    _uMChlcbCEd28KratHji1uw"/>
42                              <styles xmi:type="notation:FilteringStyle" xmi:id="
                                    _uMChlsbCEd28KratHji1uw"/>
43                            </children>
44                            <styles xmi:type="notation:ShapeStyle" xmi:id="
                                  _uMChkcbCEd28KratHji1uw" fontName="Sans"/>
45                            <layoutConstraint xmi:type="notation:Bounds" xmi:id="
                                  _uMChksbCEd28KratHji1uw" x="15" y="15" width="210"
                                  height="64"/>
46                          </children>
47                          <children xmi:type="notation:Node" xmi:id="
                                _ubjkQcbCEd28KratHji1uw" type="3001" element="
                                _ubjkQMbCEd28KratHji1uw">
48                            <children xmi:type="notation:Node" xmi:id="
                                  _ubkLUMbCEd28KratHji1uw" type="7002">
49                              <children xmi:type="notation:Node" xmi:id="
                                    _wy8SYcbCEd28KratHji1uw" type="3004" element="
                                    _wy8SYMbCEd28KratHji1uw">
50                                <styles xmi:type="notation:ShapeStyle" xmi:id="
                                      _wy8SYsbCEd28KratHji1uw" fontName="Sans"/>
```

B
Class Diagrams

B.1. Layout Plug-in

As the class diagram for the layout plug-in was too huge, it is split into two pieces and is displayed in Figure B.1 and Figure B.2. The diagram shows the connections of the most relevant classes of the layout plug-in. Classes not dealing with the actual layout, such as the preference pages, are left out.

Some layout providers—*GraphViz* and the *Example layouter*—are also listed, as well as the *KimlGenericDiagramLayouter* and the *KimlSSMDiagramLayouter*, to show their dependencies from the respective classes.

B.2. Visual-Diff Plug-in

The most relevant packages of the visual-diff plug-in are shown in the class diagram in Figure B.3. It is easy to recognize the part of the *StructureMergeViewer*—responsible for the tree representation—and part of the *ContentMergeViewer*—responsible for the diagram representation—and their dependencies on certain classes, *KivikComparator* being the most important one.

Figure B.1.: Class diagram of the layout plug-in, left part

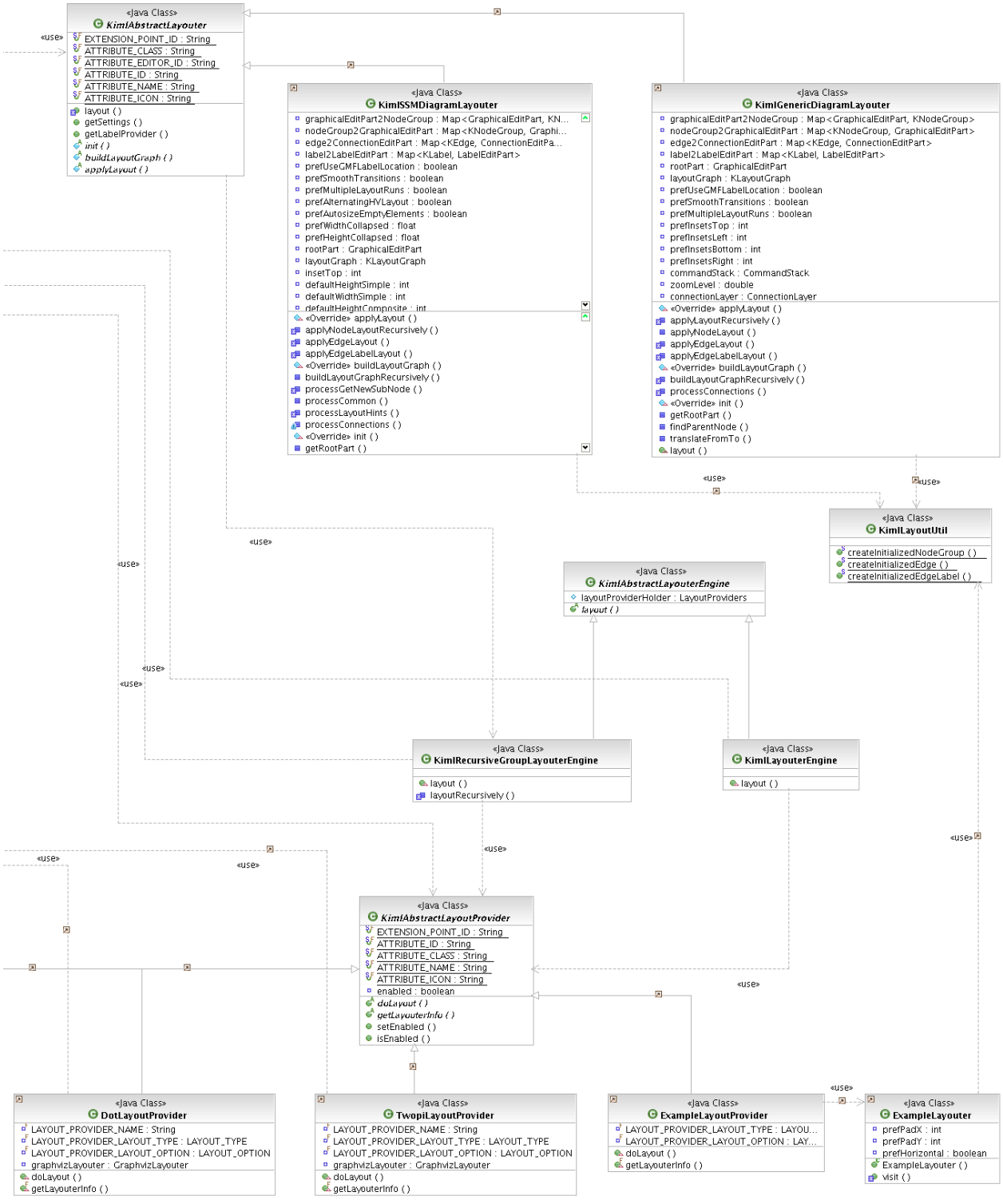


Figure B.2.: Class diagram of the layout plug-in, right part

Figure B.3.: Class diagram of the visual-diff plug-in

C Java Code

C.1. The Code

As several complete Java projects were created, it is not possible to present the source code within this written thesis. Instead, the whole code is available online on the KIELER project homepage:
http://rtsys.informatik.uni-kiel.de/trac/kieler

Relevant sub projects of KIELER this thesis covers are KIML and KiViK, which have their own sub pages on the KIELER site. The KiViK project just covers one package, but the KIML and SSM implementation is distributed over several packages, the following table gives an overview:

Package name	*Functionality*
KIML	
edu.unikiel.rtsys.kieler.kiml.ui	Classes for user interaction
edu.unikiel.rtsys.kieler.kiml.layout	The main layout core
edu.unikiel.rtsys.kieler.kiml.layouter.example	The example layouter
edu.unikiel.rtsys.kieler.kiml.layouter.graphviz	The GraphViz layouters
SSM editor	
edu.unikiel.rtsys.kieler.ssm.emf	The EMF model for the KSSM editor
edu.unikiel.rtsys.kieler.ssm.gmf	The GMF model for the KSSM editor
edu.unikiel.rtsys.kieler.ssm.gmf.diagram	The generated SSM diagram editor
edu.unikiel.rtsys.kieler.ssm.gmf.diagram.layouter	The layout extension for the SSM editor
KiViK	
edu.unikiel.rtsys.kieler.kivik	The whole KiViK project & functionality

Bibliography

Adar, E. (2006). Guess: a language and interface for graph exploration. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 791–800, New York, NY, USA. ACM. 29

Ambler, S. W. (2005). *The Elements of UML 2.0 Style*. Cambridge University Press, New York, NY, USA. 26

André, C. (1996). SyncCharts: A visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France. http://www.i3s.unice.fr/~andre/CAPublis/SYNCCHARTS/SyncCharts.pdf. 9

André, C. (2003). Semantics of S.S.M (Safe State Machine). Technical report, Esterel Technologies, Sophia-Antipolis, France. http://www.esterel-technologies.com. 9, 56, and 91

Barjavel, R. (1943). *Le voyageur imprudent ("The imprudent traveller")*. 12

Bayramoglu, Ö. (2009). A Graphical Dataflow Editor in Eclipse Basing on SCADE. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science. To appear. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/oba-dt.pdf. 56

Beeck, M. v. d. (1994). A comparison of statecharts variants. In Langmaack, H., de Roever, W. P., and Vytopil, J., editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. Springer-Verlag. 11

Bell, K. (2006). Überprüfung der Syntaktischen Robustheit von Statecharts auf der Basis von OCL. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/kbe-dt.pdf. 12

Biedl, T. C. and Kaufman, M. (1997). Area-efficient static and incremental graph drawings. In *Proceedings of the 5th Annual European Symposium on Algorithms*, volume 1284 of *Lecture Notes in Computer Science*, pages 37–52. Springer Verlag. 2

Brandes, U. and Wagner, D. (1997). A bayesian paradigm for dynamic graph layout. In *GD '97: Proceedings of the 5th International Symposium on Graph Drawing*, pages 236–247, London, UK. Springer-Verlag. 24

Branke, J. (2001). Dynamic graph drawing. In Kaufmann, M. and Wagner, D., editors, *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*. Springer Verlag. 24

Brun, C. (2007). EMF Comparison Framework: One year later. In *Eclipse Summit Europe 2007*, Ludwigsburg, Germany. 4 and 17

Brun, C. (2008). Comparing and Merging Models with Eclipse: an Update on EMF Compare. In *EclipseCon 2008*, Santa Clara, California. 4 and 17

Castelló, R., Mili, R., and Tollis, I. G. (2002). A framework for the static and interactive visualization for statecharts. *Journal of Graph Algorithms and Applications*, 6(3):313–351. 12, 21, and 25

Chawathe, S. S. and Garcia-Molina, H. (1997). 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. ACM. 2

Chawathe, S. S., Rajaraman, A., Garcia-Molina, H., and Widom, J. (1996). Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504. 2

Chok, S. S., Marriott, K., and Paton, T. (1999). Constraint-based diagram beautification. In *VL '99: Proceedings of the IEEE Symposium on Visual Languages*, page 12, Washington, DC, USA. IEEE Computer Society. 26

Christian-Albrechts-Universität zu Kiel, Real-Time and Embedded Systems Group. (last visited 05/2008). The KIELER homepage. http://rtsys.informatik.uni-kiel.de/trac/kieler. 5

Collab.Net (2008). Subversion (SVN). http://subversion.tigris.org/. 3

Conradi, R. and Westfechtel, B. (1998). Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282. 3

Crane, M. L. and Dingel, J. (2005). UML vs. Classical vs. Rhapsody Statecharts: Not all models are created equal. In Briand, L. C. and Williams, C., editors, *Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*. Springer. 11

Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM*, 15(10):859–866. 1972 ACM Turing Award Lecture. 8

Dijkstra, E. W. (2002). Go to statement considered harmful. In *Software pioneers: contributions to software engineering*, pages 351–355, New York, NY, USA. Springer-Verlag New York, Inc. 8

Eades, P., Lai, W., Misue, K., and Sigiyama, K. (1991). Preserving the mental map of a diagram. In *Computergraphics*, 99, pages 24–31. 2

Eclipse Foundation, T. (2008a). Equinox homepage. http://www.eclipse.org/equinox/, retrieved 2008-12-10. 52

Eclipse Foundation, T. (2008b). Jface - eclipsepedia. http://wiki.eclipse.org/index.php/JFace, retrieved 2008-12-10. 52

Eclipse Software Foundation (2008). Eclipse homepage. http://www.eclipse.org/. 4, 52, and 53

Ellson, J., Gansner, E., Koutsofios, E., North, S., and Woodhull, G. (2003). Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools. In Junger, M. and Mutzel, P., editors, *Graph Drawing Software*, pages 127–148. Springer-Verlag. 28

EMF Compare Team (2008). EMF Compare Wiki Page. http://wiki.eclipse.org/index.php/EMF_Compare, retrieved 2008-04-15. 19

Esterel Technologies, Inc (2007). *Esterel Studio User Manual*. Esterel Technologies, Inc, 6.0 edition. 9

Esterel Technologies, Inc (last visited 05/2008). SCADE Suite. http://www.esterel-technologies.com/products/scade-suite/. 4, 33, and 34

ExpertControl GmbH (2008). Produkte - ecDIFF. http://www.expertcontrol.com/de/products/ecdiff.php. 33

Fecher, H., Schönborn, J., Kyas, M., and de Roever, W. P. (2005). 29 new unclarities in the semantics of UML 2.0 State Machines. In *ICFEM*, volume 3785 of *Lecture Notes in Computer Science*, pages 52–65. Springer. 11

Frigg, R. and Hartmann, S. (2006). Models in Science. Technical report, Metaphysics Research Lab, CSLI, Stanford University. http://plato.stanford.edu/entries/models-science/, retrieved 2008-04-28. 7

Gansner, E. R., Koutsofios, E., North, S. C., and Vo, K.-P. (1993). A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230. 21, 22, and 28

Gentleware AG. Company homepage. http://www.gentleware.com/. 4 and 30

Girschick, M. (2006). Difference detection and visualization in uml class diagrams. Technical Report TUD-2006-5, TU Darmstadt. http://www.mm.informatik.tu-darmstadt.de/staff/girschick/publications/2006_umldiffcld.pdf. 17, 31, and 32

GNU (2006). Cvs - concurrent versions systems. http://www.gnu.org/software/cvs/. 3

GraphViz (2007). Graphviz—graph drawing tools. http://graphviz.org/. 2 and 28

Green, T. R. G. (1989). 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. Cambridge University Press. 35

Green, T. R. G. and Petre, M. (1996). Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *J. Visual Languages and Computing*, 7(2):131–174. http://www.idealibrary.com/links/doi/10.1006/jvlc.1996.0009/pdf;http://www.ndirect.co.uk/~thomas.green/workStuff/Papers/UsabilityVPs.PDF. 26

Gronback, R. (2008). Graphical modeling framework. http://www.eclipse.org/gmf/. 54

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274. 9

Harel, D. (1988). On visual formalisms. *Communications of the ACM*, 31(5):514–530. http://doi.acm.org/10.1145/42411.42414. 9

Harel, D. and Yashchin, G. (2002). An algorithm for blob hierarchy layout. *The Visual Computer*, 18:164–185. 25

Hudson, R. (2003). Graphical editing framework. http://www.eclipse.org/gef/. 54

Hunt, J. and McIlroy, M. (1976). An algorithm for differential file comparison. Technical Report 41, Bell Laboratories. 2

IBM (2007). Rational Unified Process. http://www-306.ibm.com/software/awdtools/rup/. 37

Jünger, M. and Mutzel, P. (2003). *Graph Drawing Software*. Springer. 13, 28, and 111

Kakoulis, K. G. and Tollis, I. G. (1997). On the edge label placement problem. In *GD '96: Proceedings of the Symposium on Graph Drawing*, pages 241–256, London, UK. Springer-Verlag. 25

Kaufmann, M. and Wagner, D., editors (2001). *Drawing Graphs: Methods and Models*. Number 2025 in Lecture Notes in Computer Science (LNCS). Springer-Verlag, Berlin, Germany. http://link.springer.de/link/service/series/0558/tocs/t2025.htm. 22 and 28

Kaufmann, M. and Wagner, D., editors (2007). *Graph Drawing, 14th International Symposium, GD 2006, Karlsruhe, Germany, September 18-20, 2006. Revised Papers*, volume 4372 of *Lecture Notes in Computer Science*. Springer. 13

Kaufmann, M. and Wiese, R. (2002). 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. Springer-Verlag. 39

Kelter, U. (2007). Dokumentdifferenzen. http://pi.informatik.uni-siegen.de/kelter/lehre/lm/dif. 2 and 3

Kernighan, B. W. and Plauger, P. J. (1982). *The Elements of Programming Style*. McGraw-Hill, Inc., New York, NY, USA. 12

Kloss, T. (2005). Automatisches Layout von Statecharts unter Verwendung von GraphViz. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/tkl-dt.pdf. 12, 21, 22, 25, 28, 39, 59, and 69

Kolovos, D. S., Paige, R. F., and Polack, F. A. (2008). Second international workshop on layout of (software) engineering diagrams (led'08). In *IEEE Symposium on Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008.*, pages 3–3. IEEE. 28

Kruchten, P. (2003). *The Rational Unified Process: An Introduction*. Addison Wesley. 8

Lee, Y.-Y., Lin, C.-C., and Yen, H.-C. (2006). Mental Map Preserving Graph Drawing Using Simulated Annealing. In *APVIS '06: Proceedings of the Asia Pacific symposium on Information visualisation*, pages 179–188, Darlinghurst, Australia, Australia. Australian Computer Society, Inc. 39

Maier, S. and Minas, M. (2007a). A Pattern-Based Layout Algorithm for Diagram Editors. In *Layout of (Software) Engineering Diagrams 2007*, volume 7 of *Electronic Communications of the EASST*, Berlin, Germany. 30

Maier, S. and Minas, M. (2007b). A generic layout algorithm for meta-model based editors. In *AGTIVE*, pages 66–81. 30

Maier, S. and Minas, M. (2008). 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. 30 and 31

Maraninchi, F. (1991). The Argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*. 9

Mathworks Inc. (2006). *Simulink – Simulation and Model-Based Design*. The Mathworks, Inc., Natick, MA, 6.5r2006b edition. http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sl_using.pdf. 10

Mehra, A., Grundy, J., and Hosking, J. (2005). 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. ACM. 4, 20, 21, and 35

Merks, E. (2008). Eclipse modeling framework. http://www.eclipse.org/modeling/emf/. 4 and 53

Misue, K., Eades, P., Lai, W., and Sugiyama, K. (1995). Layout adjustment and the mental map. *Journal of Visual Languages & Computing*, 6(2):183–210. 2 and 39

Myers, R. H. (1986). *Classical and Modern Regression with Applications*. Duxbury Press, Boston. 2

North, S. C., Gansner, E., Ellson, J., Koutsofios, E., and Woodhull, G. (2003). Graph Drawing Software. In *Workshop on Statistical Inference, Computing and Visualization for Graphs*, Stanford University. 21

Ohst, D., Welle, M., and Kelter, U. (2003a). Difference tools for analysis and design documents. *icsm*, 00:13. 3 and 31

Ohst, D., Welle, M., and Kelter, U. (2003b). Differences between versions of uml diagrams. *SIGSOFT Softw. Eng. Notes*, 28(5):227–236. 3, 31, 35, 44, and 45

OSGi Alliance (2008). Homepage. http://www.osgi.org/, retrieved 2008-12-10. 52

Peters, A.-K. (2008). Musterbasiertes Layout von Statecharts. Masters thesis, Universität Hamburg, Falkultät für Mathematik, Informatik und Naturwissenschaften, Department Informatik. 12, 23, 26, 67, and 81

Petre, M. (1995). Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44. 12, 25, and 27

Piccolo (2005). Piccolo homepage. http://www.cs.umd.edu/hcil/piccolo. 41

Pilgrim, J. v. (2007). Mental Map and Model Driven Development. In *Layout of (Software) Engineering Diagrams 2007*, volume 7 of *Electronic Communications of the EASST*, Berlin, Germany. 48

Pilgrim, J. v. (2008). Gef3D. http://www.fernuni-hagen.de/se/personen/pilgrim/gef3d/. 49

Prochnow, S. (2008). *Efficient Development of Complex Statecharts*. PhD thesis, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, Kiel, Germany. 12, 22, 27, 39, and 59

Prochnow, S. and von Hanxleden, R. (2004). Visualisierung komplexer reaktiver Systeme – Annotierte Bibliographie. Technical Report 0406, Christian-Albrechts-Universität Kiel, Department of Computer Science, Kiel, Germany. http://www.informatik.uni-kiel.de/uploads/tx_publication/2004_tr06.pdf. 13 and 26

Prochnow, S. and von Hanxleden, R. (2006). Comfortable modeling of complex reactive systems. In *Proceedings of Design, Automation and Test in Europe (DATE'06)*, Munich, Germany. 12, 26, and 48

Prochnow, S. and von Hanxleden, R. (2007). Statechart development beyond WYSIWYG. In *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, Nashville, TN, USA. 27, 45, and 48

Purchase, H. C. (2002). Metrics for graph drawing aesthetics. *Journal of Visual Languages and Computing*, 13(5):501–516. 27

Purchase, H. C., Hoggan, E. E., and Görg, C. (2006). How important is the "mental map"? — an empirical investigation of a dynamic graph layout algorithm. In Kaufmann, M. and Wagner, D., editors, *Graph Drawing*, volume 4372 of *Lecture Notes in Computer Science*, pages 184–195. Springer. 39

Rational Rose Realtime. IBM. http://www.rational.com/rosert. 10 and 16

Real-Time and Embedded Systems Group (2008). Seminar Eingebettete Echtzeitsysteme: Technologien rund um Eclipse. http://www.informatik.uni-kiel.de/rtsys/teaching/ws08-09/s-eclipse/, retrieved 2008-12-10. 54

Schaefer, G. (2006). Statechart style checking – automated semantic robustness analysis of Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science. 12

Schmidt, M. (2007). SiDiff: generische, auf Ähnlichkeiten basierende Berechnung von Modelldifferenzen. In *Workshop, Vergleich und Versionierung von UML-Modellen (VVUM07)*, Hamburg, Germany. 4 and 19

Schmidt, M. and Glötzner, T. (2008). Constructing Difference Tools for Models Using the SiDiff Framework (Informal Research Demonstration). In *ICSE 2008 Companion Proceedings, 30th International Conference on Software Engineering*, Leipzig. 4 and 19

Schneider, C. (2003). CASE Tool Unterstützung für die Delta-basierte Replikation und Versionierung komplexer Objektstrukturen. Diploma thesis, Corolo Wilhelmina zu Braunschweig, Germany. 3

Schneider, C., Zündorf, A., and Niere, J. (2004). 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. http://www.se.eecs.uni-kassel.de/fileadmin/se/publications/SZN04.pdf. 3 and 20

Shu, N. C. (1989). Visual programming: perspectives and approaches. *IBM Syst. J.*, 28(4):525–547. reprinted 1999. 25

Spark Systems (2008). The Compare Utility (Diff). http://www.sparxsystems.com.au/resources/diff/. 31

Spönemann, M. (2009). On the Automatic Layout of Dataflow Diagrams. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science. To appear. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/msp-dt.pdf. 60, 66, 67, 70, and 91

Starke, F. (2009). Executing Statecharts with the Kiel Esterel Processor. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science. To appear. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/fast-dt.pdf. 56

Stylus Studio (2008). XML Differencing tool. http://www.stylusstudio.com/xml_differencing.html. 31

Sugiyama, K., Tagawa, S., and Toda, M. (1981). Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125. 22

SWT Community, T. (2008). SWT: The Standard Widget Toolkit. http://www.eclipse.org/swt/, retrieved 2008-12-10. 52

Tamassia, R., Battista, G. D., and Batini, C. (1988). Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern.*, 18(1):61–79. http://dx.doi.org/10.1109/21.87055. 22

The Eclipse Foundation (2008a). Eclipse Modeling - MDT - UML2 Tools. http://www.eclipse.org/modeling/mdt/downloads/?project=uml2tools, retrieved 2008-12-10. 82

The Eclipse Foundation (2008b). Eclipse UI Guidelines. http://wiki.eclipse.org/User_Interface_Guidelines. 70

The KIEL Project, (Kiel Integrated Environment for Layout). (2006). Project homepage. http://www.informatik.uni-kiel.de/rtsys/kiel/. 5 and 51

The Object Management Group. UML Homepage. http://www.uml.org/. 1 and 8

The Object Management Group (2006). Meta Object Facility (MOF) Core Specification, v2.0. http://www.omg.org/spec/MOF/2.0/PDF/. 53

The Object Management Group (2008). OMG trademarks. http://www.omg.org/legal/tm_list.htm, retrieved 2008-04-29.

Tollis, I. G., Battista, G. D., Eades, P., and Tamassia, R. (1999). *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall. 13 and 21

Toulmé, A. (2006). Presentation of EMF Compare Utility. In *Eclipse Modeling Symposium at Eclipse Summit Europe 2006*, Esslingen, Germany. 4 and 17

Toulmé, A. (2007). Model Comparison Panel. In *EclipseCon 2007*, Santa Clara, California. 4 and 17

Treude, C., Berlik, S., Wenzel, S., and Kelter, U. (2007). Difference computation of large models. In *ESEC-FSE '07: Proceedings of the 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. ACM. 4 and 19

Universität Bremen (2005). uDraw(Graph) Homepage. http://www.yworks.com/de/index.html, retrieved 2008-04-17. 2 and 3

University of Paderborn, Software Engineering Group. (2006). Fujaba. http://www.fujaba.de/. 4

Völcker, J. (2008). A quantitative analysis of Statechart aesthetics and Statechart development methods. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science. 12, 22, 26, 39, and 59

Wenzel, S. and Kelter, U. (2006). Model-driven design pattern detection using difference calculation. In *Proc. of the 1st International Workshop on Pattern Detection For Reverse Engineering (DPD4RE), co-located with 13th Working Conference on Reverse Engineering (WCRE'06)*, Benevento, Italy. 4 and 19

White, J., Schmidt, D. C., and Gokhale, A. (2008). Generic eclipse modeling system. http://www.eclipse.org/gmt/gems/. 56

Wiese, R., Eiglsperger, M., and Kaufmann, M. (2003). yfiles - visualization and automatic layout of graphs. In Jünger and Mutzel (2003). 30

Wischer, M. (2006). Textuelle Darstellung und strukturbasiertes Editieren von Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science. 10

Xing, Z. and Stroulia, E. (2005). UMLDiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65, New York, NY, USA. ACM. 4 and 17

Xing, Z. and Stroulia, E. (2007). Differencing logical UML models. *Automated Software Engg.*, 14(2):215–259. 4 and 17

yWorks (2005). yFiles homepage. http://www.yworks.com/ger/products_yfiles_about.htm. 4 and 30

Zhu, N., Grundy, J. C., Hosking, J. G., Liu, N., Cao, S., and Mehra, A. (2007). Pounamu: A meta-tool for exploratory domain-specific visual language tool development. *Journal of Systems and Software*, 80(8):1390–1407. 20

Index