STEFFEN PROCHNOW

EFFICIENT DEVELOPMENT OF COMPLEX STATECHARTS

EFFICIENT DEVELOPMENT OF COMPLEX STATECHARTS

STEFFEN PROCHNOW

Dissertation

Zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(Dr. rer. nat.)

der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Kiel, Mai 2008

GUTACHTER:

1. Reinhard von Hanxleden (e-mail: rvh@informatik.uni-kiel.de)
2. Susanne Graf (e-mail: susanne.graf@imag.fr)

DATUM DER MÜNDLICHEN PRÜFUNG:

3. Juli 2008

ORT:

Kiel

The point is that analytical designs are not to be decided
on their convenience to the user or necessarily their readability
or what psychologists or decorators think about them;
rather, design architectures should be decided
on how the architecture assists analytical thinking about evidence.

— Edward Tufte

ABSTRACT

Modeling systems based on semi-formal graphical formalisms, such as Statecharts, has become standard practice in the design of reactive embedded devices. Using paradigms established so far often results in complex models that are difficult to comprehend and maintain. Moreover, potential errors can be very subtle and hard to locate in complex systems for the human beholder. This severely compromises the practical use of the Statechart formalism.

To overcome this, we present a methodology to support the easy development and understanding of complex Statecharts. Central to our approach is the definition of a *Statechart Normal Form (SNF)*, which makes systematic use of secondary notations to aid readability. The approach employs an efficient automated layout mechanism that transforms any given Statechart to this SNF. For the understanding of complex Statecharts during simulation we provide a *dynamic variant of the SNF* which provides a dynamic focus-and-context view based on the semantics of Statecharts. This technique reduces the displayed Statechart complexity using context abstraction of the locus of control. As an alternative to the standard WYSIWYG editing paradigm, which has some weaknesses for complex Statecharts, we present a graphical approach that is rather *oriented towards the underlying structure* of the System Under Development (SUD), and another approach based on a textual, dialect-independent *Statechart description language*. We also present an approach to *transform textual Esterel programs* into equivalent graphical Safe State Machines—a synchronous Statechart variant. These proposals permit a design flow, where the designer efficiently develops the structure of a system, but uses a graphical browser and simulator to inspect and validate the SUD. Furthermore, we focus on an approach analyzing Statecharts that limits or even prevents the use of error-prone expressions.

The Kiel Integrated Environment for Layout (KIEL) is a prototypical modeling tool to explore our editing, browsing and simulation paradigms in the design of complex reactive systems. An empirical study on the usability and practicability of our Statechart editing techniques, including a Statechart layout comparison, indicates significant performance improvements in terms of editing speed and model comprehension compared to traditional modeling approaches. Experimental measurements on KIEL indicate the practicality and effectiveness of our methodologies.

PUBLICATIONS

Some of the concepts and figures presented in this thesis have appeared previously in the following publications. For a current on-line bibliography on the Kiel Integrated Environment for Layout (KIEL) project, visit the KIEL homepage: http://www.informatik.uni-kiel.de/rtsys/kiel.

[1] Steffen Prochnow and Reinhard von Hanxleden. Statechart Development Beyond WYSIWYG. In *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, Nashville, TN, USA, October 2007.

[2] Steffen Prochnow and Reinhard von Hanxleden. The Use of Complex Stateflow-Charts with KIEL—An Automotive Case Study. In *Proceedings of 5th GI-Workshop Automotive Software Engineering (ASE'07)*, Bremen, Germany, September 2007.

[3] Steffen Prochnow and Reinhard von Hanxleden. Enhancements of Statechart-Modeling—The KIEL Environment. In *Proceedings of the ARTIST 2007 International Workshop on Tool Platforms for Modeling, Analysis and Validation of Embedded Systems, held in conjunction with the 19th International Conference on Computer Aided Verification (CAV 2007)*, Berlin, Germany, July 2007.

[4] Steffen Prochnow and Reinhard von Hanxleden. Enhancements of Statechart-Modeling—The KIEL Environment. In *Proceedings of the Design, Automation and Test in Europe University Booth (DATE'07)*, Nice, France, April 2007.

[5] Steffen Prochnow, Gunnar Schaefer, Ken Bell, and Reinhard von Hanxleden. Analyzing Robustness of UML State Machines. In *Proceedings of the Workshop on Modeling and Analysis of Real-Time and Embedded Systems (MARTES'06), held in conjunction with the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML 2006)*, Genua, Italy, October 2006.

[6] Stephan Höhrmann, Hauke Fuhrmann, Steffen Prochnow, and Reinhard von Hanxleden. A Versatile Demonstrator for Distributed Real-Time Systems: Using a Model-Railway in Education. In Amund Skavhaug and Erwin Schoitsch, editors, *Proceedings of the Second ERCIM/DECOS Workshop on Dependable Embedded Systems: Dependability Issues of Networked Embedded Systems: Research, Industrial Experience and Education*, Cavtat, Croatia, August 2006.

[7] Steffen Prochnow, Claus Traulsen, and Reinhard von Hanxleden. Synthesizing Safe State Machines from Esterel. In *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, Ottawa, Canada, June 2006.

[8] Steffen Prochnow and Reinhard von Hanxleden. Comfortable Modeling of Complex Reactive Systems. In *Proceedings of Design, Automation and Test in Europe (DATE'06)*, Munich, Germany, March 2006.

[9] Steffen Prochnow and Claus Traulsen. KIEL—Textual and Graphical Representations of Statecharts (Presentation). In *12th Synchronous Workshop (SYNCHRON'05)*, Malta, November 2005. URL: http://www.cs.um.edu.mt/~synchrone05/Presentations/15-SteffenProchnow.pdf.

[10] Steffen Prochnow and Reinhard von Hanxleden. Visualisierung komplexer reaktiver Systeme – Annotierte Bibliographie. Technical Report 0406, Christian-Albrechts-Universität Kiel, Department of Computer Science, Kiel, Germany, June 2004. URL: http://www.informatik.uni-kiel.de/uploads/tx_publication/2004_tr06.pdf.

ACKNOWLEDGMENTS

First of all, I owe my sincere gratitude to my supervisor Reinhard von Hanxleden, who inspired my interest enhancing the usability of visual languages and their modeling process. Thank you very much for all the interesting discussions, the large freedom you allowed me for my research, and for a personal and friendly collaboration. I would like to thank Susanne Graf (VERIMAG, Grenoble) for her willingness becoming my second assessor.

I owe many thanks to the members of the development team with whom I collaborated as part of my dissertation project: Ken Bell, Karsten Heymann, Tobias Kloss, Lars Kühl, Florian Lüpke, André Ohlhoff, Adrian Posor, Gunnar Schaefer, Jan Täubrich, Jonas Völcker, and Mirko Wischer.

I would like to thank all Real-Time and Embedded Systems Group members for creating a great atmosphere for doing research, for many fruitful discussions, and for all the enjoyable lunches. In particular, I would like to thank not only Claus Traulsen and Jan Lukoschus for interesting discussions on Esterel problems, and Hauke Fuhrmann for valuable hints on the KIEL modeling tool. I thank Tim Grebien who was always willing to help in technical problems, and Gesa Walsdorf and Maren Lutz for their anticipatory assistance and their cordiality.

Jürgen Golz (Department of Psychology, University of Kiel) and Christiane Gross (Department of Sociology, University of Kiel) have been an invaluable help in the design and analysis of the validation experiment. I also thank the students who participated in the experimental study for their support and interest. I would also like to thank Matthias Grochtmann and Heiko Dörr (DaimlerChrysler AG) for providing us an interesting Stateflow application. Charles André (Université Nice Sophia Antipolis) has suggested improvements for the SSM synthesis from Esterel. Special thanks go to Ken Bell, Christiane Gross, and all the anonymous reviewers for their helpful comments.

This work would not have been possible without my parents and my sister. Thank you for your support and for being always there, when I need you.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ABS	Anti-Lock Braking System
ADBL	Alternating Dot Backwards Layout
ADL	Alternating Dot Layout
AL	Arbitrary Layout
ALL	Alternating Linear Layout
API	Application Programming Interface
AST	Abstract Syntax Tree
ASCII	American Standard Code for Information Interchange
AToM^3	A Tool for Multi-Formalism, Meta-Modelling
ATP	Automated Theorem Proving
CASE	Computer-Aided Software Engineering
CEC	Columbia Esterel Compiler
CLOVER	Cluster-Oriented Visualization Environment
CVC Lite	Cooperating Validity Checker Lite
DiaGen	Generator for Diagram Editors
DSNF	Dynamic Statechart Normal Form
EMF	Eclipse Modeling Framework
ESC	Electronic Stability Control
FSM	Finite State Machine
GEF	Graphical Editor Framework
GenGEd	Generator of Visual Language Editors
GraphViz	Graph Visualization Software

HTML	HyperText Markup Language
HUTN	Human-Usable Textual Notation
JNI	Java Native Interface
KIEL	Kiel Integrated Environment for Layout
KIELER	KIEL for Eclipse Rich Client Plattform
KIT	KIEL Statechart Extension of Dot
KOCL	KIEL wrapped OCL
LabVIEW	Laboratory Virtual Instrumentation Engineering Workbench
LLL	Linear Layer Layout
MAAB	MathWorks Automotive Advisory Board
MISRA	Motor Industry Software Reliability Association
MVC	Model View Controler
NASA	National Aeronautics and Space Administration
OCL	Object Constraint Language
OMG	Object Management Group
PCB	Printed Circuit Board
PGML	Precision Graphics Markup Language
RSML	Requirements State Machine Language
SCR	Software Cost Reduction
SCXML	Statechart XML
SMT	Satisfiability Modulo Theories
SNF	Statechart Normal Form
SSM	Safe State Machine
SUD	System Under Development

SVG	Scalable Vector Graphics
SVM	Statechart Virtual Machine
SWIG	Simplified Wrapper and Interface Generator
UMC	UML on the fly Model Checker
UML	Unified Modeling Language
VLSI	Very-Large-Scale Integration
WYSIWYG	What You See Is What You Get
XMI	XML Metadata Interchange
XML	Extensible Markup Language

1 INTRODUCTION AND MOTIVATION

The current change in developing control systems, especially in the automotive industry, is the replacement of electrical, mechanical, and hydraulic solutions by software-based electronic controllers, *i. e.*, *embedded systems*. These systems are designed to perform highly specific tasks and are completely encapsulated by the devices they control. Due to economical modification, extension, and duplication the software-based design of such controllers is much more flexible. Typical application areas of embedded systems are consumer electronics (*e. g.*, mobile phones), home automation (*e. g.*, climate control units), and safety-critical environments (*e. g.*, Anti-Lock Braking System (ABS)). The extremely growing demands for functions of embedded systems call for even more efficient development solutions.

The change in developing control systems

Reactive systems are systems that have permanent interaction with their environment, as is typical for embedded systems. The execution of these systems is determined by their internal state and external stimuli. As a reaction, new stimuli and/or a new internal state are generated. To describe the behavior of reactive systems, the family of synchronous languages has been developed, including Esterel [20], Lustre [68] and Signal [66]. These languages offer numerous control flow primitives (such as concurrency and preemption) which are pertinent to reactive systems; the *synchrony hypothesis* [20] gives a sound semantical basis to these languages.

Specification languages of reactive systems behavior

As an alternative to the aforementioned textual languages, one may also develop reactive systems using graphical notations, such as Statecharts [71]. Statecharts extend the classical formalism of the Finite State Machine (FSM) and the state transition diagram by incorporating the notions of hierarchy, orthogonality, compound events, and a broadcast mechanism for communication between concurrent components. Statecharts provide an effective graphical notation, not only for the specification and design of reactive systems, but also for the simulation of the modeled system behavior. Moreover, it is possible to automatically generate optimized program code that implements a Statechart [21, 22].

Statecharts as specification language of reactive systems behavior

Since the original Statecharts proposal by Harel, numerous dialects of Statecharts have been developed and Statecharts have also been incorporated into the Unified Modeling Language (UML) [114]. Today, Statecharts are supported by several commercial tools, *e.g.*, Matlab Simulink/Stateflow [154], Statemate [73], or Rational Rose [134]. In this thesis, the main work deals with the Safe State Machines (SSMs) [3] dialect of Statecharts, which is a graphical variant of Esterel.

Statecharts make systems comprehensible and provide a good system overview.

A commonly touted advantage of graphical formalisms, such as Statecharts, is their intuitive usage and the good level of overview they provide—according to the phrase "one picture is worth ten thousand words." Especially for visualizing complex structures, Statecharts have advantages compared to textual programming languages. Here, the one-dimensional flow of text provides poor overview over the whole system, while Statecharts provide clustering elements, such as states, hierarchy and orthogonality to structure Statechart components.

An advantage of Statecharts is their intuitiveness, but complex Statecharts get unmanageable.

However, the graphical modeling of realistic applications often results in very large and unmanageable graphics, severely compromising their readability and practical use. The problem becomes even more dramatic when starting to simulate the system, as modular designs typically instantiate Statecharts several times, and each instance may have its own simulation status. In one Statechart example provided to us, describing an airbag system, this multiple instantiation resulted in a total of 288 Statecharts. The classical paradigm to animate a simulated Statechart is to highlight active states, *e.g.*, by marking them in a particular color; however, this is only of use if the active states are visible to the user in the first place. When the total number of Statecharts of a system goes significantly beyond what can be visible on the screen simultaneously, as was the case in the airbag example, keeping track of the active states of a system quickly becomes a rather frustrating exercise.

Summarizing, complexity in Statecharts can originate from different problems of the development process:

LARGE SYSTEMS: The increasing functionality of developed systems results in a high number of components. This leads to intricate interactions and inter-dependencies between specified system components. As a result the modeler of realistic systems can easily lose track of the entire System Under Development (SUD) and the model intention.

DYNAMIC BEHAVIOR: During the simulation phase of realistic Statecharts one is not only confronted with the complexity of the system itself but also with the complexity of its dynamic behavior. For example, simulations steps may result in several states. Consequently, many windows are necessary in order to focus their occasionally distributed locations in a static view simulation. But beyond a certain number of windows it is not possible to observe all of them simultaneously.

2D EDITING: Entering textual programming code to specify a system is very efficient. Due to the linear text flow, the insertion and deletion of symbols and words is very simple. Regarding graphical models, the two-dimensional nature brings some editing complications. Before inserting elements, the modeler must often "make room" first. The deletion of elements is also tedious, because it often leaves distracting "holes." To handle this and to produce nice and readable graphics, adjacent elements have to be moved, while respecting the relationships of interacting elements (*e. g.*, transitions, state hierarchy, concurrency).

A problem is that existing modeling tools do not offer good mechanisms for abstracting or condensing Statechart representations; each Statechart appears the way a designer has modeled it using some graphical editor. In our experience, even a large screen rarely permits to view more than four or five Statecharts at once.

Existing modeling tools limit the efficiency of the Statechart development process.

Since the introduction of Statecharts some twenty years ago, significant progress has been made concerning their semantics, formal analysis and efficient implementation. Concerning the practical handling of Statecharts, however, it appears that comparatively little progress has been made since the very first Statechart modeling toolset [73]. Specifically, the *construction*, *modification*, and *revision management* of Statecharts tend to become increasingly burdensome for larger models, and we feel that, in this respect, Statecharts are still at a disadvantage relative to other development activities, such as classical textual programming. This observation, corroborated by numerous discussions with practitioners and modeling experiences ranging from small, academic models to industrial projects, has motivated the work presented in this thesis.

We view visual formalisms such as Statecharts as a very powerful concept for the development of reactive systems because graphical

models may be convenient to browse. However, compared to textual entry, they are rather cumbersome to construct and maintain, as designers spend a significant fraction of their time with tedious drawing and layout chores.

We also feel that today's paradigms for editing, visualizing and simulating Statecharts have not progressed significantly since the first perception of Statecharts, and that they do not scale well to develop complex systems. We are convinced that the usefulness of Statecharts depends to a large extent on their readability, that is the capability of the drawing to convey its meaning quickly and clearly. Thus motivated, we have developed a methodology that seeks to support a system developer in modeling, simulating and comprehending complex Statecharts.

The intention of this work is to improve the efficiency of the Statechart development process.

1.1 PURPOSE AND CONTRIBUTION

This thesis summarizes the results of our research regarding the efficiency of the Statechart editing process, the readability of resulting Statecharts, the improvements during Statechart simulation, and the maintainability of Statechart models.

EDITING: Statecharts are commonly created using some What You See Is What You Get (WYSIWYG) editor, where the modeler is responsible for the graphical layout, and subsequently a Statechart appears the way a designer has modeled it. We believe that the WYSIWYG construction paradigm, which leaves the task of graphical layout to the human designer, has been a limiting factor in the practical usability of Statecharts, or graphical modeling in general so far.

The premise of this thesis is that this paradigm may have been justified at some point, but current advances in layout algorithms and processing power make it feasible to free the designer from this burden. Based on this premise, this thesis presents two alternative Statechart construction paradigms—a text-based and a macro-based technique—that let the modeler focus on the modification of the Statechart structure, *i. e.*, the Statechart element topology. These techniques apply an automatic layout mechanism that frees the modeler from the burden of manually rearranging Statechart elements. The layout method transforms any given Statechart to a standard-

ized layout Statechart Normal Form (SNF) that is compact and makes systematic use of secondary notations in order to aid readability.

The automatic layout method also allows our approach to synthesize SSMs from (textual) Esterel programs. This permits a design flow where the designer develops a system at the Esterel level, but uses a graphical browser and simulator to inspect and validate the system under development.

SIMULATION: A common problem when simulating complex systems with many concurrent activities is that designers easily loose track of the current overall system state. This thesis addresses this by a "dynamic semantic focus-and-context representation," which provides different views of the system depending on the system state.

ERROR PREVENTION: In this thesis an automated checking framework is described, which checks compliance with robustness rules. The framework provides a wide range of predefined dialect-dependent rules, and also allows to express new design rules in the Object Constraint Language (OCL) or in Java. The latter is also used to incorporate a theorem prover for more advanced checks, such as determinism.

1.2 CHAPTER OVERVIEW

This section gives an outline of the chapters in this thesis.

CHAPTER 2: An overview of existing approaches that address the described problems in editing and reading Statecharts is presented on in this chapter. The chapter points out how they relate to our work as well as their advantages and disadvantages.

CHAPTER 3: This chapter presents the analysis of WYSIWYG editing patterns and the identification of generic Statechart editing patterns. An SNF is presented that gives a clear, compact, consistent view of a Statechart solely based on its topology. Furthermore, the macro-based and text-based Statechart construction approaches and the KIEL Statechart Extension of Dot (KIT) language are discussed. A further contribution of

this chapter is a synthesis mechanism that derives a graphical Statechart model, specifically the SSM dialect, from a textual imperative programming language, Esterel. Therefore, a set of optimization rules is given that transform a Statechart (SSM) into an equivalent, but more compact Statechart—these rules can also be useful when handling Statecharts developed in the traditional WYSIWYG way. Afterwards, a discussion follows of how to assert the correctness of such a transformation and optimization.

CHAPTER 4: For simulations, we extend the concept of SNF to *Dynamic Statecharts*, which continuously adapt their appearance to the current system configuration, providing a focus-and-context representation based on the active regions of each Statechart.

CHAPTER 5: This chapter introduces a fundamental inspection and classification of error prevention in software in general and focuses on style checking in Statecharts. We perform a comparison of existing style guides and application for textual programming languages and Statecharts. Furthermore, a set of *robustness rules* for less error-prone Statecharts following the advice of Buck and Rau [24] is provided.

CHAPTER 6: The Kiel Integrated Environment for Layout (KIEL) tool is a prototypical modeling framework that has been developed for the exploration of complex reactive system designs. A central capability of KIEL is the automatic layout of graphical models. This not only reduces the designer's drawing effort significantly, but also permits novel construction and simulation paradigms and design flows. KIEL also incorporates a checking framework, which automatically proves predefined OCL-based rules.

CHAPTER 7: KIEL serves as an evaluation platform for our proposals. In this chapter we show the usability of the KIEL framework. Therefore, we characterize the runtime performance of KIEL and present the results of an empirical study on the usability and practicability of our Statechart editing techniques, including a Statechart layout comparison. This study indicates significant performance improvements in terms of

editing speed and model comprehension compared to traditional modeling approaches.

CHAPTER 8: In this chapter conclusions are drawn, reflections on our work including discussing the limitations of our work are made, and an outlook to opportunities for future work is given.

2

BACKGROUND AND RELATED WORK

The field of enhancing readability and comprehension of complex reactive systems appears to have been rather neglected in the past. The work presented in this thesis cuts across several related areas that have been studied. In the following section we give a short introduction into Statecharts and describe characteristics of SSMs. Approaches, techniques and experiences on Statechart modeling tools, which had an influence on KIEL, are introduced in Section 2.2. KIEL was developed to support a system developer in modeling, simulating and comprehending complex Statecharts. Therefore, Section 2.3 gives an overview of available layout methods, focus-and-context methods, and methods to keep track of dynamically changing models. Section 2.4 collects works, which influenced the synthesis of graphical Statecharts from textual Esterel and the development of the Statechart description language KIT. An inspection of existing robustness checking techniques and frameworks in software engineering, that influenced the development of the style guide and the automated checking framework described in this thesis, is provided in Section 2.5. Finally, Section 2.6 shortly describes the works that contributed to the KIEL project.

2.1 AN INTRODUCTION TO STATECHARTS

The graphical language Statecharts is one of the most popular approaches for the development of real-time and reactive systems. The Statechart formalism was introduced by Harel [71] and constitutes an extension of the classical formalism of the state transition table and the FSM [133] by incorporating the notions of hierarchy, orthogonality, compound events, and a broadcast mechanism for communication between concurrent components. Statecharts provide an effective graphical notation, not only for the specification and design of reactive systems, but also for the simulation of the modeled system behavior.

Statecharts constitute an extension of Finite State Machines.

Since the original Statecharts proposal by Harel, numerous dialects of Statecharts have been developed—each with different

syntax and semantics. Beeck [14] compares 21 of these Statechart dialects. As an informal and implementation-independent definition, the Object Management Group (OMG) has incorporated Statecharts into the UML specification [114]. KIEL can display the Statechart variants of *SyncCharts* [2, 5] (which is the predecessor of SSM), Stateflow Statecharts [101], and UML Statecharts as implemented in ArgoUML [6]. KIEL can also simulate Statecharts according to the behavior specification of André [2] and uses the Stateflow Application Programming Interface (API) [154] to simulate Stateflow charts. Furthermore, KIEL synthesizes SSMs from textual Esterel specifications. Figure 1a shows an example of the Statechart dialect SSM, as it is implemented in Esterel Studio [39] and Figure 1b shows an example of a Stateflow Statechart. The graphical objects of the Statechart representations have been labeled for a short survey.

Characteristics of Safe State Machines

SSMs are a Statechart dialect with synchronous semantics that strictly conform to the Esterel semantics. A procedural definition of SSMs is given by André [3].

States in SSMs

A *macro-state* consists of one or more state-transition graphs. Additionally, SSMs can contain *textual macro states*, which consist of plain Esterel code. States can also have *internal actions*: *on entry*, *on exit* and *during*.

Signals in SSMs

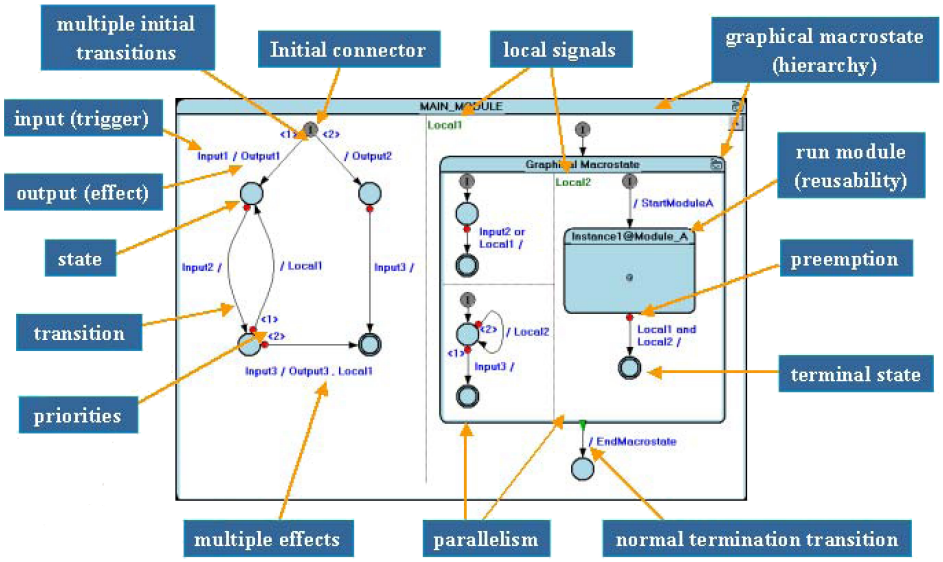
SSMs inherit the concept of signals and valued signals from Esterel. Hence, a transition trigger can consist of an event, which tests for presence and absence of values, and a conditional, which may compare numerical values.

Preemption in SSMs

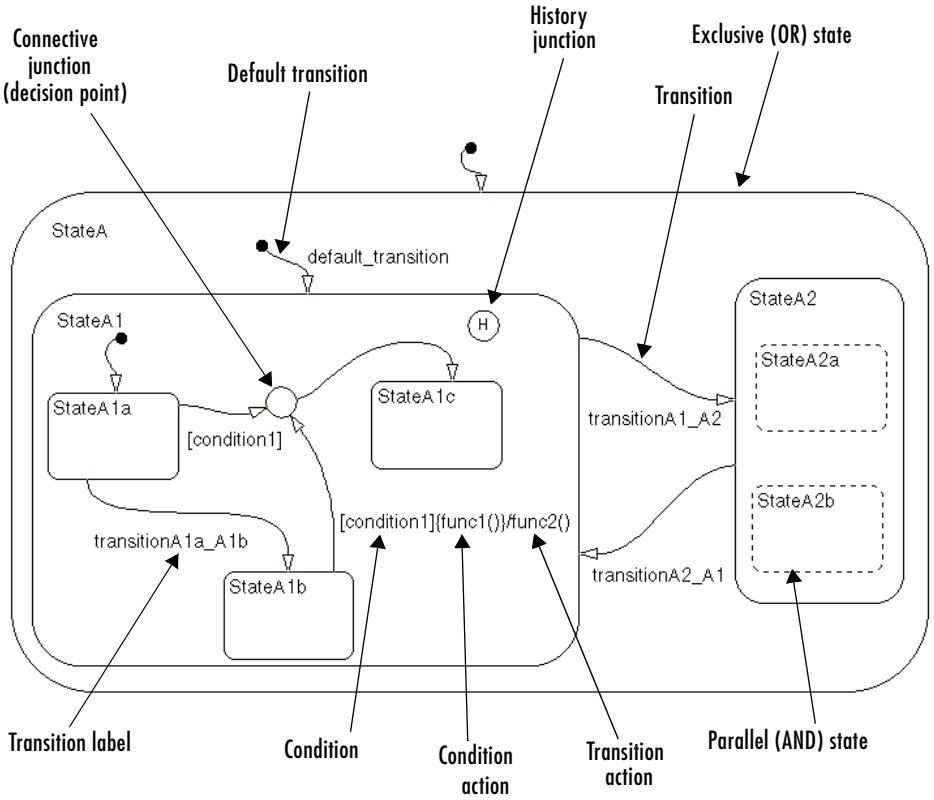
Characteristics of SSMs are the different forms of preemption, expressed by different state transition types. Weak and strong abortion transitions, as well as suspension, can be applied to macro states. A macro state can either be left by an abortion, which has an explicit trigger, or by a *normal termination*, which is taken if the macro state enters a terminal state.

Characteristics of transitions in SSMs

Analogously to Esterel, all transitions can either be immediate or delayed, where a delayed transition is only taken if the source state was already active at the start of an instant. In contrast, immediate transitions may be taken as soon as the state becomes active; this makes it possible that one state is activated and deactivated multiple times within one instant. Delayed transitions can also be



(a) Example of an SSM (source [38])



(b) Example of a Stateflow Statechart (source [101])

Figure 1: Examples of different Statechart dialects

count delayed, *i. e.*, the trigger must have been evaluated to true for a specific number of times, before the transitions is enabled. When a state has more than one outgoing transition, a unique *priority* is assigned to each of them, where lower numbers have higher priority. Weak abortions must have lower priority than strong abortions, and if a normal termination exists, it always has the lowest priority.

2.2 MODELING ENVIRONMENTS FOR STATECHARTS

There are a large number of modeling tools which provide a modeling environment for Statecharts. They generally include tools for creation and simulation. These modeling tools are often used for the modeling of complex software or hardware systems and provide in general a specific semantic for the simulation behavior of the modeled system. Modern also tools support code synthesis, reverse engineering, and version control features. These features automatize burdensome modeling tasks, *e. g.*, synthesizing source code from a graphical model. However, as discussed in Chapter 1, today's modeling tools do not offer editing techniques for an efficient modeling process. Also Robbins and Redmiles state:

Existing Statechart modeling tools provide basic techniques to develop Statechart models and to synthesize running code.

> *"[. . .] current CASE tools fail to address the essential cognitive challenges facing software designers."* [138]

An exemplary UML Modeling Tool

An outstanding example of a modeling tool which focuses cognitive support of the modeling process for graphical languages is ArgoUML [6]. It provides design tool features intended to support design tasks. Each of these features are motivated by the developers' experience in designing software systems. Robbins and Redmiles [138] describe the following features which focus the user oriented support of model development and which were inspired by theories of cognition in design:

ArgoUML was developed with consideration of cognitive aspects in developing software models.

CRITICS AND CRITICISM CONTROL MECHANISMS: Design critics are agents that check the design for potential problems. In ArgoUML, the modeler is continuously be informed about potential design problems. The modeler does need not need to request for critique; it is automatically generated. In ArgoUML, over sixty design critics have been implemented.

"TO-DO" LIST: A "to-do" list in ArgoUML is a text list of critique messages automatically generated by the system; it can also be supplemented by the modeler. It acts as a track list for design problems which are still unresolved.

NON-MODAL WIZARDS: ArgoUML provides a mechanism with non-modal wizards in order to propose designer solutions for a design problem. The sequential resolving of multiple design problems frees the modeled systems from defects. Trivial problems can be solved automatically.

CHECKLISTS: In ArgoUML, there is a distinction between *checklists* and *critics*. Critics mainly serve to remind designers of common design problems. In contrast, ArgoUML checklist items are more concrete, as *e. g.*, "Could gradePointAvrg be moved from Undergraduate to Student?" The ideas of ArgoUML's *critics* and *checklists* are realized in KIEL as the *style checker* (cf. Section 6.1).

OPPORTUNISTIC TABLE VIEWS: *Table views* can be requested by the modeler; they represent information pertaining to design elements (*e. g.*, states with outgoing respectively incoming transitions) in a dense tabular format.

NAVIGATIONAL PERSPECTIVES: As an enhancement of the classical "static" tree structure view on a hierarchical system, ArgoUML provides a selection of different tree structures. *E. g.*, class structures can be listed combined with their contained states and transitions; states can be listed with their successor states, *etc.*

BROOM ALIGNMENT TOOL: Robbins and Redmiles [138] point out that "alignment is one important form of secondary notation." ArgoUML provides the *broom alignment tool* for the alignment of graphical elements. The application of ArgoUML's broom tool is depicted in Figure 2 . The broom tool pushes objects that come in contact with it. Thus, graphical objects are aligned along the broom's face. An experimental study performed by Robbins et al. [139] showed that the usage of the alignment tool significantly reduces the number of mouse actions during the modeling process.

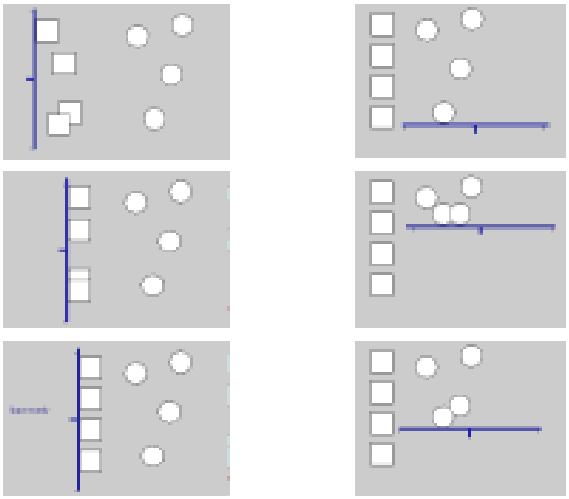


Figure 2: Vertical and horizontal application of ArgoUML's broom alignment tool (source: Robbins and Redmiles [138])

However, ArgoUML does not support the developer in simulation of a developed Statechart model. Furthermore, it does not provide a layout method for arranging graphical elements automatically.

Other Modeling Tool Approaches

Generator frameworks for domain specific languages can simplify the development effort of graphical editors.

Besides existing graphical WYSIWYG editors for modeling Statecharts, a lot of frameworks for the generation of domain specific graphical editors exist. Approaches for generating graphical editors rely on meta-modeling concepts, grammars, or some kind of logic. A very flexible approach of a graph grammar driven framework for the synthesis of graphical editors is the Generator for Diagram Editors (DiaGen). With DiaGen, an editor for a certain kind of diagram can be generated from an Extensible Markup Language (XML) specification. It includes a hypergraph grammar, which describes the structure of diagrams to edit. In DiaGen layout is realized using certain graphical constraints that influence the position between graphical objects. In contrast to this method, KIEL provides a layout for Statecharts from scratch. Section 6.2 describes the functionality of the layout method implemented in KIEL.

DiaGen editors generally support two main properties of powerful graphical editors:

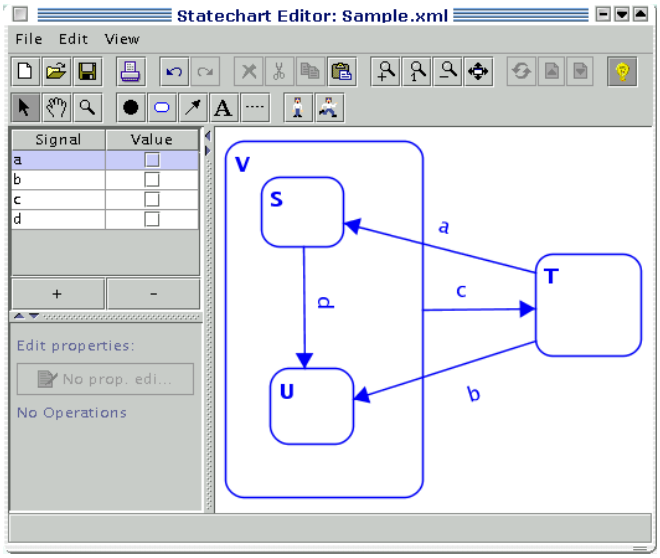


Figure 3: Screen shot of a Statechart modeling tool generated using DiaGen (source: Minas [104])

SYNTAX-DIRECTED EDITING: Using this technique "[...] the user is [...] restricted to a collection of predefined editing operations" [105].

FREE-HAND EDITING: The user of a graphical editor "[...] is not restricted at all, but misses the convenience of [...] complex editing operations" [105].

A prototypical Statechart editor was realized for demonstration using the DiaGen framework [104]. It does not only support Statechart editing, but also animation of the Statechart behavior. Figure 3 shows the Statechart editor, which was developed with the DiaGen. Besides the Statechart editor, a graphical editor for class diagrams was also devised by Köth and Minas [88]. This editor supports the focus-and-context technique for "structure-based" abstraction. It supports in detail (quoting Köth and Minas):

- "the abstraction of classes by 'hiding' their attributes and operations [...], and
- the abstraction of packages by 'hiding' all contained elements." [88]

As the DiaGen diagram editor for class diagrams, KIEL also uses the hierarchy abstraction paradigm during Statechart simulation.

Besides the DiaGen the Generator of Visual Language Editors (GenGEd) [8, 59] and A Tool for Multi-Formalism, Meta-Modelling (AToM^3) [29] also employ graph grammars to generate graphical editors for visual languages. The manipulation of the visual language using such a generated editor is restricted due to the underlying graph grammar. GenGEd uses graph grammars to modify visual languages using graph productions. The visual language is produced by *a priori* specified production sequences. KIEL's macro-based editing approach instead proposes interactive manipulations of the model.

The graph grammar based tools use graphical constraints for placing graphical elements. Graphical constraints only influence the placement of graphical objects relative to another. Especially for complex systems this technique results in unreadable graphics. A layout from scratch rearranges all graphical objects independently from another. Hence it can find optimal positions for all graphical objects. Using KIEL we perform an automatic layout from scratch.

A tool which provides different views on the same SUD is, *e. g.*, *Ptolemy II* [36]. Brooks et al. [23] describe the combination of synchronous/reactive models as *AND* states and finite state machines as *OR* states in *Ptolemy II*.

Experimental Studies on Statechart Modeling

Results of empirical studies can improve the future software modeling process.

Several experimental studies address the comprehensibility of textual and visual programs. *E. g.*, Green and Petre [64] performed an experimental study to evaluate the usability of textual and graphical notations using the Laboratory Virtual Instrumentation Engineering Workbench (LabVIEW) [112]. They determined that visual programs can be harder to read than textual ones. Purchase et al. [130] have evaluated the aesthetics and comprehension of UML class diagrams. We are not aware of any experimental studies on the effectiveness of *editing* visual languages. In this thesis an empirical study on the effectiveness and practicability of different Statechart editing techniques is presented. The experiments also compare the aesthetics and the comprehensibility of manually and automatically created Statechart layouts.

2.3 VISUALIZATION OF COMPLEX STATE-BASED SYSTEMS

From Graph Layout to Statechart Layout

Graph drawing algorithms

Generally, graph drawing algorithms can be applied to the drawing of Statecharts. This is practicable because of the similar basic node-transition principle. As one of the bases in graph drawing, Di Battista et al. [30] published a recapitulatory work concerning all topics of graph drawing. Programming frameworks that implement some of the numerated algorithms are collected in Jünger and Mutzel [81]. A typical problem in drawing graphs is the placement of nodes in conjunction with well-tuned placement of edges. Here, Batini et al. [11] introduce algorithms using an orthogonal approach, a tree structured layout is provided by Reingold and Tilford [135], a force-directed approach by Eades [33], Kamada and Kawai [83], and a milestone in graph drawing—the layer-based algorithm—is realized by Sugiyama et al. [150]. The exigence of automatic layout, especially of generated graphs, is pointed out in Fleischer and Hirsch [43].

Aesthetic criteria in drawing graphs

As a degree of observer-friendly layout of placed objects, Tamassia et al. [152] spot and define aesthetic criteria concerning graph drawing. Further works extend the aesthetic criteria to the drawing of Statecharts [28, 72]. Such aesthetic criteria are affected by the perception of the observer of graphs, respectively, Statecharts. Other publications, such as [64, 130, 138, 141], focus on cognitive perception analysis of graphical notations especially for software specification.

Specific aspects in drawing Statecharts

Several works deal with specific aspects of Statechart representations. Harel and Yashchin [72] developed techniques for the layout of *blobs*, which are edge-less hierarchical structures that correspond to Statecharts without transitions. Castelló et al. [27] extend the label placement problem for Statecharts by an approach to handle flexible geometries. Both works also consider aesthetic criteria. To make Statechart simulations more understandable, Efroni et al. [35] introduce an approach to animate the simulation using meaningful pictures.

Tools for laying out Statecharts

Castelló et al. [27] have developed the framework *Vista* for the automatic generation of Statechart layouts. Their work treats hierarchical drawing, labeling and floor planning for Statecharts. Their focus is the preservation of the mental map while editing. Thus,

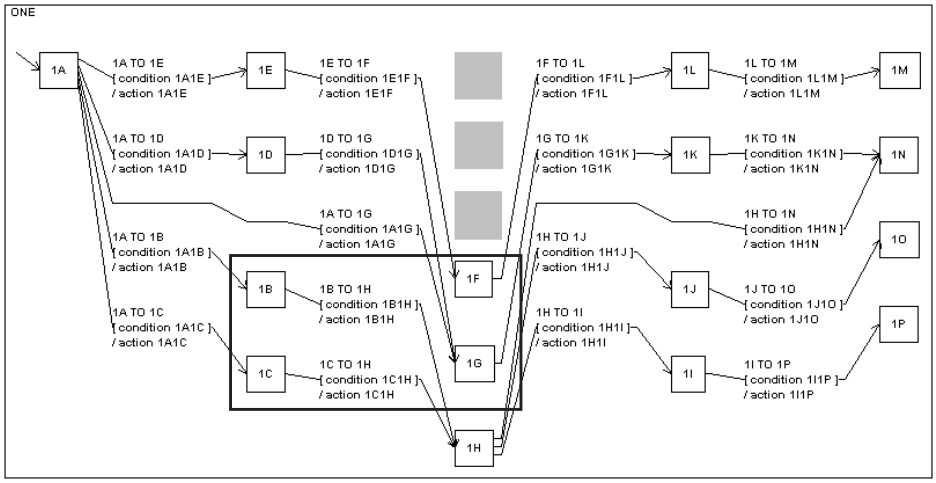


Figure 4: Statechart laid out using Vista (source: Castelló et al. [27])

the resulting Statecharts also indicate the editing history, which may be desirable in certain cases, but conflicts with the concept of an SNF. Furthermore, they do not seem to address backward transitions, and the resulting drawings are not very readable due to long and intersected transitions (cf. Figure 4).

As one of the few modeling tools based on Statecharts, Rational Rose [134] supports the modeler by automatically rearranging the manually constructed Statechart. The underlying layout algorithm uses horizontal layers to place states of upper hierarchy level (inner states will not be touched) and performs a middle affine placement of polygon or spline curve transition. However, the resulting layouts very satisfying, as the state placement still seems unsystematic, and transitions may cut into states. We even experienced that transitions are superimposed, making it impossible to discern their labels. Figure 5 shows an automatically laid out Statechart using Rational Rose. KIEL offers several layout mechanisms, some employ the Graph Visualization Software (GraphViz) [57] layout framework and others were developed from scratch. GraphViz [56, 63] is a collection of tools implementing several graph layout algorithms. It uses heuristics for the computation of graph layouts and finds very fast optimal coordinates for nodes, also for huge graphs. The *dot* tool from the GraphViz framework draws graphs according to the style of Sugiyama et al. [150]. Figure 6 shows a graph drawn using the dot tool. The computation time of the shown compiler dependency graph was 0,98 sec [55].

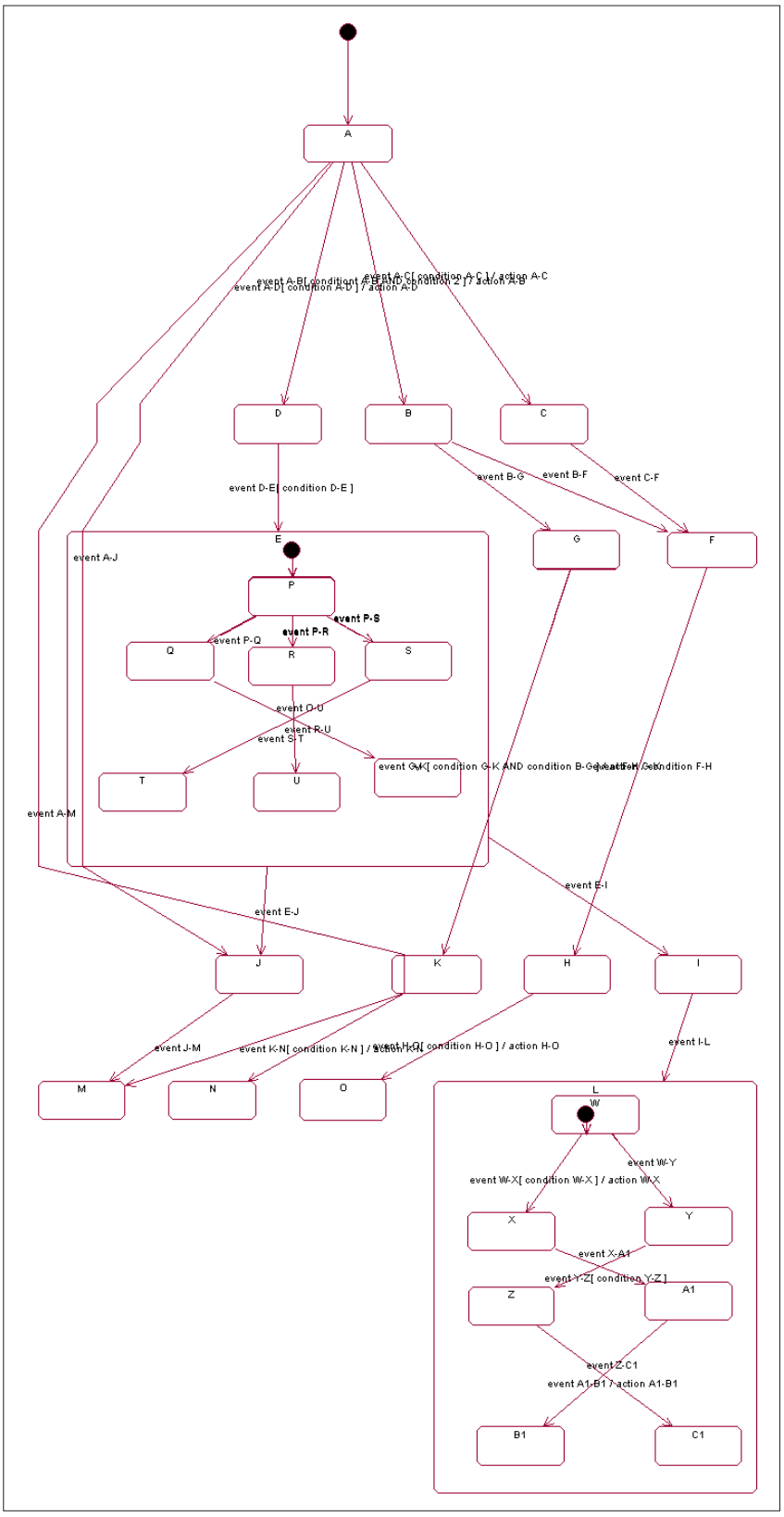


Figure 5: Statechart Example in Rational Rose after automatic layout (source: Castelló et al. [27])

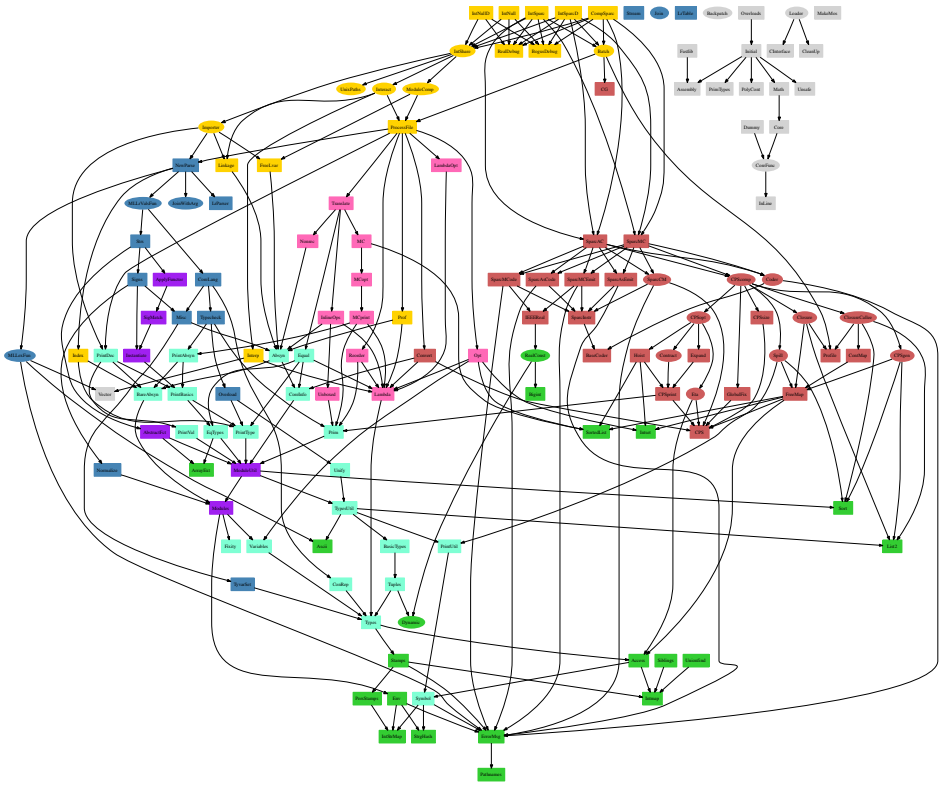
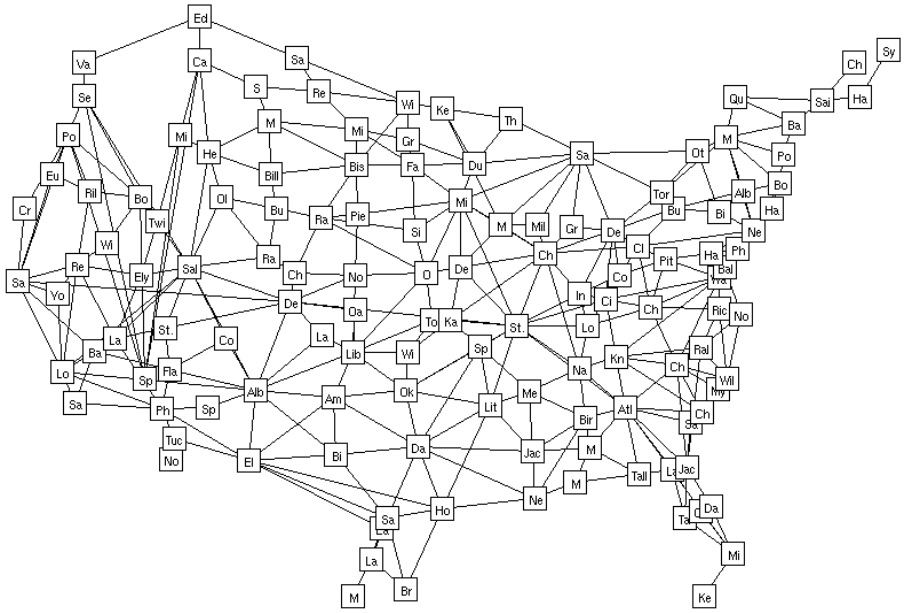


Figure 6: A complex graph drawn by the dot tool (source: Gansner et al. [55])

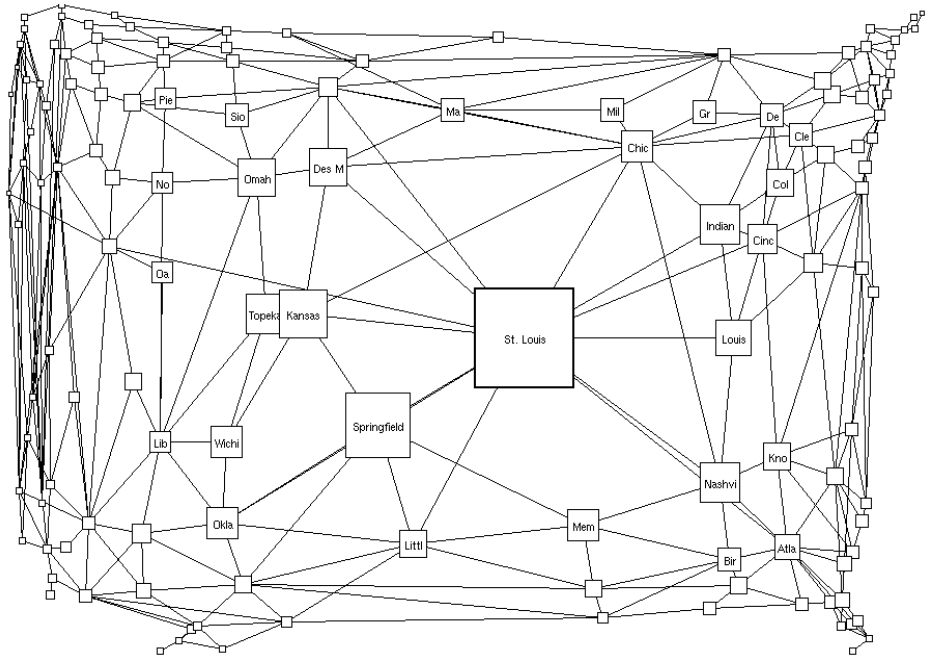
Focus-and-Context Methods

Focus-and-context methods allow to inspect detail of large systems while its context is always present.

One of the important challenges in visualization systems is how to present as much important information as possible given a finite display area. Focus-and-context (or also *fisheye*) methods have been developed to address this problem by attempting to smoothly integrate detail views with as much surrounding context as possible, so that users can see all relevant information in a single view. Much work has been done on focus-and-context approaches to graph representations. Research works in the fields of information visualization, graph drawing, software analysis, *etc.* have contributed to this topic. Observations and solutions concerning zoomable user interfaces are introduced by Pook et al. [124]. Leung and Apperley [94] provide an overview of focus-and-context techniques. Furnas [51] gives a description of the original focus-and-context view. Sarkar and Brown [142] developed a variant of focus-and-context views for graphs. Figure 7 illustrates the graphical focus-and-context



(a) Original graph representation



(b) Fisheye view of the graph in Figure 7a

Figure 7: Different graph representations of major cities of the United States (source: Sarkar and Brown [142])

method depicting the major cities in the United States as a graph. In Bederson and Hollan [13] an interactive graphical user interface was developed which uses suppression of detailed information and displays a system in a simplified form.

Focus-and-context methods for hierarchical systems

Berner et al. [16] modify this approach with the concept of graphical abstraction of hierarchical structures. Köth and Minas [88] have applied semantic focus-and-context representations to visualize complex UML class diagrams with the DiaGen. Minas [104] has also presented a Statechart editor based on DiaGen that supports syntax-directed editing, offers a layout method that enforces correctness constraints, and a simple simulator. However, there are no dynamically changing views. KIEL's approach of Dynamic Statecharts is an extension of this concept, which provides dynamically changing views on a SUD, according to the simulation state.

3D Methods

3D methods are efficient for the visualization of complex systems, but they make high demands on computer hardware and cognitive abilities.

For large state-transition systems, Groote and van Ham [65] visualize state spaces in 3D. This technique uses a clustering method to obtain a simplified representation in form of a tree. Their method allows to obtain information about complexity estimation of state systems and effectiveness of different testing approaches [65]. Figure 8 visualizes the state space of a communication protocol. Sheelagh et al. [147] extend the focus-and-context method from 2D to 3D. Further 3D methods for visualizing large graphs are the cone trees of Robertson et al. [140] and the fractal approach for visualizing huge hierarchies of Koike and Yoshihara [86]. Gil and Kent [60] describe an approach of 3D software modeling.

Maintenance of the Mental Map

The mental map in perception of graphical models

While a human modeler interacts with a Statechart, he or she will gather and maintain an image of the Statechart structure. This concerns the general location and relations between states and pseudo states of a Statechart. This phenomenon—the *mental map*—was observed for graphs by Misue et al. [106]. When the Statechart is modified (*e. g.*, by applying an automated layout method or by supplementing an existing Statechart with new state elements) or when a focus-and-context method changes the viewpoint heavily, special care must be taken to prevent breaking the modelers current

Figure 8: 3D tree visualization of a large state space system (source: Groote and van Ham [65])

mental map. For maintaining the mental map, Misue et al. [106] describe three models based on the position of the nodes in the diagram:

1. the *orthogonal ordering model*, which attempts to capture relative directions between nodes in a diagram,
2. the *proximity relations model*, which attempts to capture the notion due to node distances, and
3. the *topology model*, which attempts to capture the notion using regional node relations.

They also introduce layout adjustment methods for the preservation of the mental map. Freire and Rodríguez [48] identified the following general factors that contribute to mental map preservation:

PREDICTABILITY: A user of a visual system must be aware of the importance of what changes are to be expected. This can be achieved by (1) visual feedback of actions, (2) educating user

about methods of operation, and (3) keeping actions simple to understand them.

DEGREE OF CHANGE: If, *e. g.*, a new layout is applied to a graph, then the graph should be laid out according to a similar paradigm.

TRACEABILITY: For changes that are applied to a graph, it should be possible to track and integrate them into the user's mental map. This can be achieved by using animation of a newly placed element.

Freire and Rodríguez [48] also provide the Cluster-Oriented Visualization Environment (CLOVER) [47], which demonstrates algorithms for preventing the mental map. It uses hierarchical clustering and incremental layout methods for focus-and-context representations.

For the animation Diehl et al. [31] provide an approach, which is based on sequences of evolving graphs. Their approach to preserve the mental map uses a foresight layout algorithm for dynamically drawing animated graphs.

2.4 Statechart synthesis

In general, there is only very limited work on synthesizing Statecharts from textual descriptions, and the approaches that we are aware of already assume that the textual input directly expresses the Statechart topology as an *AND-OR* tree (*e. g.*, [27, 79]). This already achieves the advantages of textual entry, but does not offer the rich, concise control flow constructs available in synchronous programming languages, such as *e. g.*, Esterel.

Statechart Synthesis from Esterel

SyncCharts [2, 5], the predecessor of SSMs, have been defined via a translation to Esterel [4], and the commercial tool Esterel Studio generates Esterel programs from SSMs as part of its code generation process. A non-trivial problem arising from this is the "linearization" of arbitrary state transitions (which are comparable to a *goto*) into an imperative control flow. To our knowledge, there also have been no attempts yet to explore the other direction, as we do here.

As discussed in Section 3.5, one complication arising in the presented transformation is to express Esterel's *trap* mechanism in SSMs. This situation also arises in the Esterel derivate Quartz [146], which does not support traps directly (see also Section 3.5.3). Notwithstanding, Quartz allows to test the current configuration of a system. As we do not have this option in SSMs, we present an alternative mechanism based on explicit trap signals in Section 3.5.

The problem of Esterel's trap mechanism

Regarding Statechart optimizations, there have been already proposals for design rules that define "good" Statecharts. However, they typically focus on checking consistency [44] or graphical appearance, and do not attempt to systematically reduce the complexity of a Statechart, as we do here.

Textual Statechart Notations

There exist several Statechart description languages, with different objectives. *E. g.*, the Statechart XML (SCXML) [171], a variant of XML, has a comprehensible structure; but the required *tags* and their hierarchical dependencies call for specific XML editors. Alternative Statechart descriptions, such as Statechart Virtual Machine (SVM) [42] and the UML on the fly Model Checker (UMC) [102] use explicit declarations of Statechart objects; such expressions reduce the readability, especially, for large Statecharts. The technique of explicit declarations provides the advantages of textual entry, but does not offer the Statechart dialect-independent, concise constructs, as available in KIT described in Section 3.4.2. The KIT language structure was developed with respect to the Human-Usable Textual Notation (HUTN) [116, 157], where HUTN languages are required purely for the display of information. In contrast, McIver and Conway [103] and Richard and Ledgard [136] suggest several design principles of human usable programming languages. Having these in mind, several design ideas for KIT were borrowed from similar description languages: from the *dot* notation [57] the implicit node declaration, from Argos [98, 99] the hierarchy construction, and from the Esterel language the parallel construction.

Explicit object declarations

These Statechart description languages generally serve as an intermediate format synthesized from manually edited Statecharts. To our knowledge, none of these languages has been used so far for Statechart synthesis, as we propose to do here. An exception is the Requirements State Machine Language (RSML) approach [75],

which synthesizes a graphical view of the topology using a very simple, but surprisingly effective layout scheme, which inspired KIEL's alternating linear layout. However, RSML still keeps much information that is normally part of the graphical model in textual *AND-OR* tables.

2.5 PREVENTING STATECHART MODELING ERRORS

To lower the defect rate in developing software, a number of tools and style guides for classical textual programming languages have been developed, dealing not only with code layout but also with robustness aspects. *E. g.*, C language style is addressed by the Motor Industry Software Reliability Association (MISRA) [107] and Java style is addressed by Sun [151] and National Aeronautics and Space Administration (NASA) [32]. Style guides have been published also for the Statechart's modeling paradigm as well, *e. g.*, by the MathWorks Automotive Advisory Board (MAAB) [100] and by the Ford Motor Company [45], both exclusively for Stateflow. Scaife et al. [143] propose the development of a *safe subset* of the Stateflow language, which is considered to be less error-prone. Huuck [78] points out several error-prone Stateflow concepts, such as condition actions and the order of execution. He proposes the development of an appropriate set of automated *sanity checks* leading to a safer subset of Stateflow. However, neither the rules nor the automated checks are specified in detail. Furthermore, Kreppold [89] has presented a style guide for the modeling tool Statemate.

Style guides for textual programming languages

Style guides for Statecharts

When it comes to UML State Machines, the well-formedness rules defined within the UML specification (cf. OMG [113, 115]) clarify the semantics of Statechart elements. Besides the well-formedness rules, other rules for UML State Machines have been formulated in the literature, *e. g.*, by Mutz [109]. Some of his advised rules can also be applied to other Statechart dialects.

Automatically checking robustness (or soundness) of UML State Machines is an active field of research. First of all, applicable techniques had to be clarified, as done by Pap et al. [119]. They presented different techniques, which (1) are utilizing checks based on the OCL, (2) are applying graph transformation, (3) special programs, and (4) finally reachability analysis driven tests. Implementations have been conducted in several areas. Richters [137]

Automated robustness checking of UML Statecharts

has investigated different frameworks that can be used when it comes to working with OCL. Approaches to checking based on graph transformation on UML State Machines are presented by, *e. g.*, Gogolla and Parisi-Presicce [62] and Varró et al. [161].

Furthermore, Mutz and Huhn [110] have developed a customized tool (the *Rule Checker*) for the automated analysis of user-defined design rules for UML State Machines (see also [109]). They pursue an interpreter-based analysis; they use the OCL to formulate rules and to analyze the underlying data structure with help of some of the specified rules. Moreover, checking is also performed by a Java program. In contrast, our checking framework transforms OCL to Java, because a transformative approach is faster and more flexible.

Automated robustness checking of Stateflow Statecharts

Another approach in order to check the style guide conformance of Statecharts is the tool *Mint* [156] which is focused on the MathWorks Automotive Advisory Board (MAAB) style guide. The checker primarily aims at achieving a consistent look-and-feel, enhancing readability, and avoiding common modeling errors. The *Guideline-Checker* [108], a no-cost/academic alternative to *Mint*, is based on the MAAB rules and coded in Matlab. The range of the *Guideline-Checker* is currently constricted to the most trivial checks, *e. g.*, "A [state] name does not include a blank," or "A [state] name consists of [at least] 3 characters" [108, page 26]. As mentioned, both tools focus solely on the MAAB rules. Therefore, they are limited to Stateflow and are not applicable to other dialects, *e. g.*, UML State Machines.

Automated robustness checking of Statemate Statecharts

Moreover, special programs for the validation of crucial problems have been developed. Here, the *State Analyzer*, provided by DaimlerChrysler R&D, is a prototypical software tool to check the "determinism" of Statemate Statecharts [145]. Performing an automated robustness analysis of requirements specifications, the tool verifies for every state that the predicates (trigger and condition) of multiple outgoing transitions are pairwise disjoint. The approach for detecting non-determinism employs automated theorem proving (cf. Section 6.7), *i. e.*, proving the satisfiability of a formula consisting of the conjunction of each pair of transition predicates. Approaches analyzing requirements specifications are introduced by, *e. g.*, Heitmeyer et al. [76]; their approach is based on Software Cost Reduction (SCR). Heimdahl and Leveson [75] therefore utilize the RSML.

2.6 THE KIEL ENVIRONMENT

Even though a wide range of methods and applications for graph and Statechart visualization already exist, none fulfills all of our needs. They are either highly specialized, or are not extendable, or they focus different purposes. Hence, we decided to develop the KIEL tool, which embraces different mentioned methods and applications. *E. g.*, KIEL employs the GraphViz [57] layout framework for the layout of Statecharts, it borrows the concept of the semantic focus-and-context method as realized in DiaGen editors, and it uses OCL for checking rule specification.

Contributions to the KIEL project

Numerous people have contributed to the KIEL environment. The central module of KIEL—the layout mechanism—was developed by Kloss [84, 85]. The KIEL macro editor and the KIT editor were developed by Wischer [169]; he also developed the KIEL Statechart browser [168] and an Esterel Studio file import module [167]. A prototype of a KIEL WYSIWYG editor was contributed by Lüpke [97]. Ohlhoff [118] implemented KIEL's SSM Statechart simulator, which follows the semantics described in André [3]; Posor [125] implemented a Statechart simulator using the Matlab Simulink/Stateflow API. The synthesis of SSM from Esterel was developed by Kühl [90]. Heymann [77] contributed a LaTeX frontend, which makes use of KIEL's command line tool. Schaefer [144] implemented KIEL's semantic robustness analysis using Java code and Bell [15] implemented KIEL's syntactic robustness analysis using OCL. Völcker [162] investigated the causes of Statechart modeler preferences for certain Statechart layouts and editing methods. For an on-line listing of these works see http://www.informatik.uni-kiel.de/rtsys/theses/completed.

3

EDITING GRAPHICAL MODELS

Statecharts are commonly created using some What You See Is What You Get (WYSIWYG) editor. Even for novices WYSIWYG editors are very easy to use due to their intuitiveness. Section 3.1 characterizes conventional Statechart modeling techniques. Therefore, we analyze the common modeling process using WYSIWYG Statechart editors.

However, WYSIWYG editors can also be a limiting factor in the practical usability as stated for WYSIWYG word processors, *e. g.*, by Tsai [160] and also by Taylor [153]. Quoting Tsai:

> "*Since a WYSIWYG interface has to provide commands for formatting and layout in its menus and toolbars, there is no way it can be as optimized for text processing as a text editor is. This is unfortunate, since probably only 10% of the time it takes to compose a document is spent on formatting—maybe less.*" [160]

This problem becomes even harder when moving from WYSIWYG word processing to graphical WYSIWYG Statechart editing. Especially the editing of *complex Statecharts* raises further problems. Here, one is quickly confronted with large and unmanageable graphics originating from a high number of components or from intricate interactions and inter-dependencies. Due to this, we present—as an alternative to the WYSIWYG approach—our ideas of fast and effective modeling of graphical models in Section 3.2 to Section 3.5. Especially the editing of complex graphical systems can benefit from this techniques.

3.1 THE WYSIWYG STATECHART EDITING PROCESS

The WYSIWYG paradigm is very popular and allows fast graphical development from scratch.

The available Statechart modeling tools such as Esterel Studio, Stateflow or Rational Rose provide highly specialized Statechart editors that follow the WYSIWYG paradigm. A WYSIWYG Statechart editor allows to view graphical objects of the modeled Statechart very similar to the end result and implies the ability to change the layout

during the modeling process. One of the advantages of WYSIWYG editors is their intuitive handling; due to the widespread use of the WYSIWYG paradigm, also novices can operate using Statechart editors (cf. Section 7.1).

In general, Statechart modelers using WYSIWYG editors typically value the ability of fast "painting" and the full flexibility of arranging Statechart elements. WYSIWYG Statechart editors limit free-form editing to develop proper Statechart elements. This simplifies editing Statecharts by enabling the modeler to quickly enter a complete System Under Development (SUD). Nevertheless, the composition of Statechart elements using WYSIWYG Statechart editors is not innately syntactically correct. *E. g.*, when connecting transitions are deleted, unassociated state objects can remain. This necessitates checking the syntax and correcting the resulting Statechart manually. Therefore, the editing process of especially complex Statecharts gets time consuming, due to often occurring inter-dependencies of modified Statechart elements.

Graphical models are nice to browse, but hard to write.

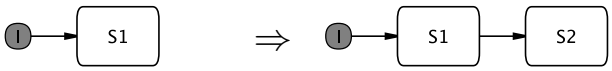
To analyze and educe improvements in developing Statecharts, we inspected the common WYSIWYG editing process. We identified six main *editing schemata*, which can be grouped into three categories: Statechart *creation*, *modification* of Statechart elements, and *deletion* of elements. These are presented in the following section with the illustration in Figure 9.

3.1.1 *Editing Schemata*

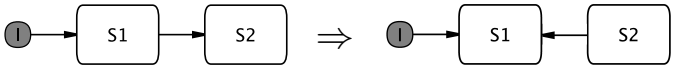
1. *Statechart creation:* For a new Statechart, the modeler has to create at least three Statechart elements: (1) one state which can be entered, (2) one initial connector for identification of the first entered state, and (3) a connecting transition.

2. *Add Statechart elements:* A frequently used Statechart modification is adding states and pseudo states. Typically, states and pseudo states (*e. g.*, initial connector) are not single elements, but connected with other states. Hence, the modeler additionally has to insert a new transition connecting a new Statechart object (see Figure 9a).

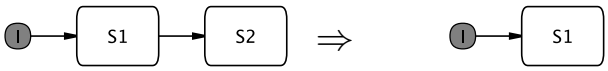
3. *Add hierarchical states and parallel regions:* A further, often observable operation is supplementing a Statechart with new hierarchical (*OR*-) states and existing (*AND*-) states with a



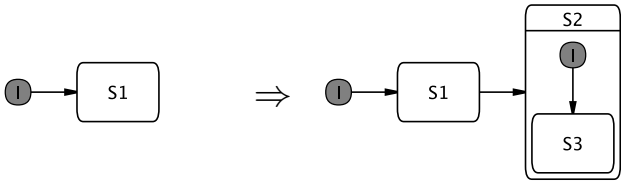
(a) Insertion of a simple successor state.



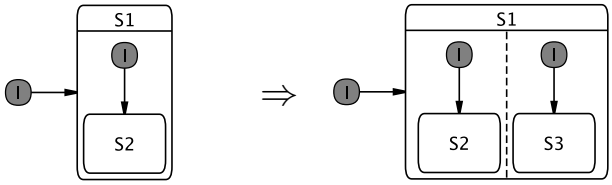
(b) Modification of transition direction.



(c) Deletion of a Statechart element.



(d) Insertion of hierarchical successor state.



(e) Insertion of a parallel region.

Figure 9: Generic editing schemata derived from a typical editing process using WYSIWYG editors

new parallel region. For adding a new *OR*-state, in the simplest case the modeler adds (1) the new hierarchical state, (2) the inner initial connector, (3) the initially indicated state, and (4) a connecting transition (see Figure 9d). Similar editing actions are necessary if a new parallel region should be entered (see Figure 9e).

4. *Change complexity of states:* When entering a new level of abstraction, it is necessary to upgrade states to hierarchical *OR*- respectively *AND*-states. In doing so, the modeled state attributes (*e.g.*, state name, activities, incoming/outgoing transitions) should be preserved (see Figure 9e). Some Statechart modeling tools (*e.g.*, Esterel Studio) provide such a

mechanism, but in most of the tools, the modeler often has to delete the existing state and re-create another state object with all its attributes.

5. *Switch transition source and target:* The inversion of a transition direction is one of the most annoying exercises modifying Statecharts in conventional Statechart editors; the modeler has to delete and re-create the transition with all its attributes (see Figure 9b).

6. *Delete Statechart elements:* Almost all Statechart elements are connected by transitions or they are related to hierarchical associations. If any Statechart element is deleted, miscellaneous elements of the deleted element's context have to be deleted to achieve a syntactically correct Statechart. *E. g.*, after deletion of a state connected to other elements, the remaining transition cannot be left without any source respectively target (see Figure 9c); hence, the modeler has to delete it. This also applies to the deletion of inner states of a hierarchic state (or parallel region), *etc.*

Using a WYSIWYG editor each of these editing schemata involves a number of steps. For example, to apply the schema "add hierarchical successor state" (Figure 9d), the modeler has to perform the following steps:

1. select the state to supplement,
2. add a new hierarchical state,
3. insert an inner initial connector,
4. insert an inner state, and
5. insert connecting transitions.

3.1.2 *Action Sequences*

Each WYSIWYG editing schema requires the modeler to perform a sequence of low-level editing steps.

When using conventional Statechart editors, none of the editing schemata can be realized as a single action. Generally, each editing schema using WYSIWYG editors passes the following action sequence:

1. If needed, create free space (*e. g.*, expand hierarchical states for new sub-elements, move existing elements for placing new elements).

2. Focus on a Statechart element for modification respectively supplementation. Initially the location of a Statechart element, which should be modified or where elements should be added, has to be determined. Within WYSIWYG editors, this is done with pointing by mouse.

3. Apply an editing schema (cf. Section 3.1.1).

4. If needed, rearrange the modified chart to improve readability.

It is a common experience that the modeler spends much time with the layout-related activities of steps 1 and 4. For Statecharts developed from scratch, this effort may be small. In contrast, if an existing chart has to be modified, the work for arranging the elements increases roughly with the number of Statechart elements and Statechart complexity. Quoting a practitioner:

> "*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.*" [122]

Furthermore, there is a very direct impact on productivity as modelers spend less time with manual layout. Each editing schema requires multiple mouse clicks selecting Statechart elements, and after that, appropriate editing actions either by mouse or by keyboard are necessary. Thereby, it is often difficult to point the mouse to an intended object; often closely placed elements avoid an exact element selection.

The analysis of the Statechart editing process using WYSIWYG editors indicates possible areas of improvement. In particular, the modification and augmentation of existing complex charts is laborious and time-consuming. Due to this, we address more practical and efficient modeling techniques in the next sections.

Graphical elements should be automatically rearranged by the computer.

The basic idea of our approach is to automate the editing process as far as possible. Specifically, we propose to reduce the effort of rearranging Statechart elements by applying automatic Statechart layout mechanisms. This produces Statecharts laid out according to a Statechart Normal Form (SNF), which is compact and makes systematic use of secondary notations to aid readability. Our approach

of automatic layout and the notion of the SNF are introduced in the next section.

3.2 LAYOUT OF GRAPHICAL MODELS

When coding in textual programming languages, it is common practice to structure the text according to some formatting conventions, for example regarding indentation [26, 151]). These conventions—referred to as *secondary notation* [121]—are irrelevant for the meaning of the program but aid the understanding of the human programmer. The same is true for graphical programming—or rather, it should be true. Quoting Gurr:

> "*Pragmatics [...] helps to bridge the gap between truth conditions and 'real' meaning—that is, between what is said and what is meant. [...] the correct use of pragmatic features, such as layout in node and link based notations, is a significant contributory factor in the comprehensibility, and hence usability, of these representations.*" [67]

However, as already observed for example by Petre [122], the use of secondary notations in graphical modeling is still underdeveloped. This not only slows down the development process, but may also lead to overlooking design faults. As Petre notes:

> "*It is time to recognize the impact of 'bad graphics'—of haphazard use of perceptual cues and secondary notations—miscueing, misleading, misreading, and misunderstanding.*"

Furthermore, making good use of secondary notations does not appear trivial for graphical programming. Quoting Petre again:

> "*It appears that graphical notations can have a greater capacity to 'go wrong' than textual notations.*"

Aesthetic criteria for drawing Statecharts

Based on aesthetic criteria [27, 72] and on the meaning behind the graphical Statechart elements, we have devised rules for the layout of these elements, thus enforcing a standardized use of secondary notations. These rules effectively define an SNF, which, for a given Statechart, defines how the elements of that Statechart should be placed. The rules we advocate include the following:

1. *Initial states are placed at the left or top area, final states are placed at the bottom/right.* If the developer of a Statechart is familiar with a drawing, he or she will rapidly notice that a common order of elements increases readability. Hence, this allows to quickly locate where behaviors start and where they finish, analogous to the sequencing in textual programs. According to the reading order, the initial state should be placed in front and the final state (see next point) is placed in rear. Placed in this way, they follow the set up of texts in most scripts.

2. *Final states are placed at the opposite of initials.* As the final component is the object the user looks at following the Statechart semantics, it is placed on the right or bottom respectively.

3. *Successive states are placed adjacently, the number of back transitions is minimal.* This further simplifies the tracking of the sequence of states in order to comprehend the modeled control flow.

4. *The number of transition bends are minimal.* This simplifies tracking a transition from source to sink, and is achieved by state placement and the use of splines.

5. *The number of transition crossings are minimal.* If two transitions intersect, it cannot be constituted which direction the transition takes. Especially in case of small angle crossings and in collaboration with polygonal transitions it, is barely feasible to follow a transition.

6. *Transitions are oriented consistently.* For example, in Kiel Integrated Environment for Layout (KIEL), transitions are oriented clockwise for horizontal layouts and also for vertical layouts. This allows to infer the orientation of a transition just from its location, obviating the need to locate the tip (arrow head) of a transition. This is particularly useful for dense representations.

7. *Labels have a consistent placement*, *e. g.*, this simplifies the mapping between labels and transitions, in particular, in case of adjacent transitions.

8. *The level-wise placement direction alternates.* The direction of state sequences, which are laid out in a linear or treelike

> fashion, alternates at different hierarchy levels between left-to-right and top-down. This causes a good aspect ratio and minimizes the placement area.

As an example, consider the Statechart for a simple traffic light controller presented in Figure 10. There are macro states for *normal* and *error* operation modes, each of which concurrently controls a light for pedestrians (*e. g.*, "*Pred*" means red light for pedestrians) and cars. The figure compares the Statechart as drawn by a modeler, using Esterel Studio with the same chart brought into SNF (using KIEL, see Chapter 6). In the original layout, states are placed irregularly, shapes of transitions vary, and space is wasted; the SNF variant is more compact and lets the viewer focus on the functionality of the Statechart.

Note that what constitutes a "good" or "bad" use of secondary notation is often a matter of opinion, just as experienced programmers may have—sometimes strong—opinions on what constitutes a good formatting style (cf. Section 7.1). Certainly, there are other alternatives to the SNF presented here. Nevertheless we do not deem it critical *which* SNF is used—as long as some SNF is used, which all members of a development team should agree upon. The price to pay for this is that modelers may not be able to draw a Statechart the exact way they would prefer. However, one obtains a uniform appearance to Statecharts, with the corresponding gain in portability and maintainability.

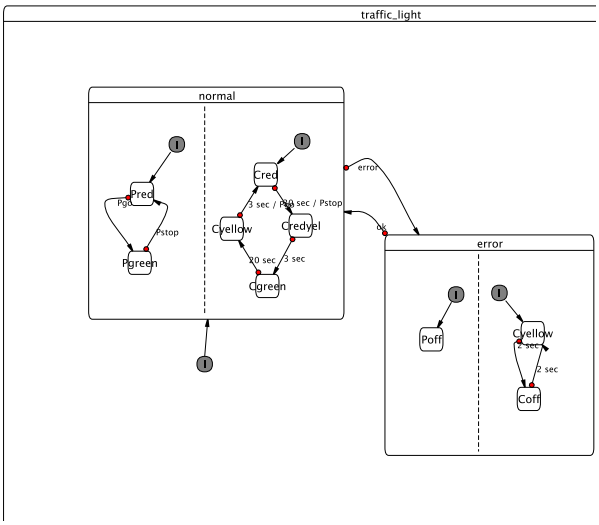
Automatic layout reduces the necessary editing actions.

Due to the application of an automatic layout mechanism, the editing action sequence of Section 3.1.2 can be reduced to:

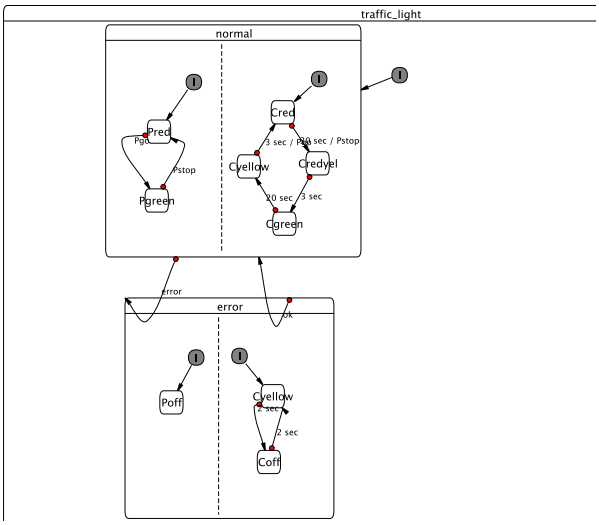
1. Focusing on a Statechart element for modification respectively supplementation;
2. Applying an editing schema.

Both editing actions remain under control of the modeler and will be treated by the following editing proposals for Statechart editing—the macro-based, the textual, and the transformational Statechart creation approaches. All three paradigms focus on the development of the Statechart structure. Quoting Laurel:

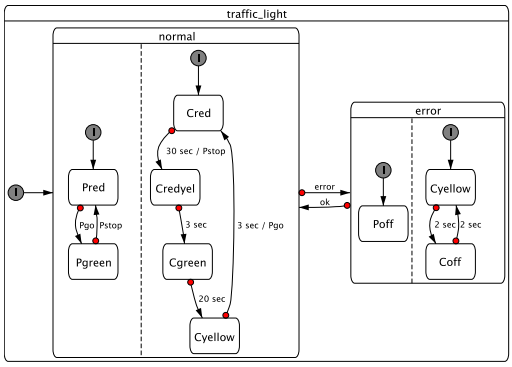
> "*Focus on designing the action. The design of objects, environments, and characters is all subsidiary to this central goal.*" [92, p. 134]



(a) Original, manually laid out Statechart.



(b) Another manually laid out representation of the Statechart shown in Figure 10a.



(c) Layout conforming to the SNF

Figure 10: A traffic light example, illustrating the SNF

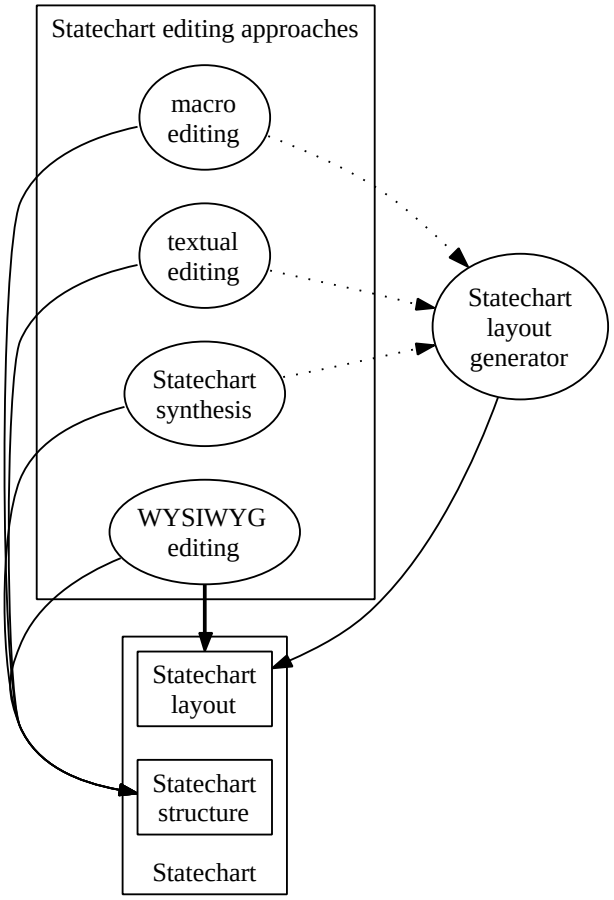


Figure 11: Different ways of Statechart creation. All mentioned editing approaches provide the Statechart structure developed by the modeler. Using the WYSIWYG paradigm the Statechart modeler additionally must take care of the Statechart layout.

And also Tidwell states:

> *"Sometimes it is easier to let the tool do certain mechanical jobs, such as precise layout, repetitive tasks, complex geometric shapes, image or sound processing, etc. Don't make the user do these by hand unless they choose to; provide easy-to-use automation instead, in a way which is smoothly integrated into the WYSIWYG Editor itself, and which doesn't require a jarring context shift in the user's mind."* [158]

We consider the graphical layouting of Statechart a repetitive task, which can be done automatically by the computer. Figure 11 characterizes the development process of WYSIWYG editing and our editing proposals. In this sense, our Statechart development proposals

follow the paradigm of markup languages, such as LaTeX [91, 129], HyperText Markup Language (HTML) [164], Extensible Markup Language (XML) [172], Scalable Vector Graphics (SVG) [165], and Precision Graphics Markup Language (PGML) [111]. This paradigm frees the modeler from time-consuming rearrangement of graphical elements and they have also proven their power when maintaining complex document layouts. Beyond this, a certain consistent style of layout which contributes to the readability of Statecharts follows.

3.3 MACRO-BASED EDITING

Using WYSIWYG editors, a simple editing action (*e. g.*, placement of a state) scarcely needs time; but applying a complete editing schema (cf. Section 3.1.1) requires multiple mouse and keyboard actions. Our proposal to optimize this is to directly manipulate the Statechart *structure*, uncoupled from its graphical representation.

Interactive application of Statechart production rules

The schemata described in Section 3.1.1 can be interpreted as *Statechart productions*. Before applying a production (a schema), the modeler selects the location for the modification (the focus), which corresponds to the left-hand side of the production. If the production pattern matches, the application of the schema replaces the focus with the right-hand side of the production. Hence, as a result of applying the production, Statecharts are produced which conform to the syntax of the visual language. The set of productions constitutes a Statechart *grammar*, which has the characteristic that every application of a production results in a syntactically correct Statechart. Hence, a design does not go through meaningless intermediate editing stages, which (ideally) frees the modeler from time-consuming syntax-checking. An exception to this are productions that delete model elements, which may result in isolated states. However, KIEL provides syntax checks that detect these (see Section 6.7).

Concerning editing step 1 of the reduced action sequence (the setting of the focus, see page 36), we propose not only to provide the traditional mouse-oriented mechanism, but also to allow a structure-oriented navigation similar to text editors. *E. g.*, in the KIEL macro editor (see Section 6.5),

1. the right/left key navigates through state sequences,

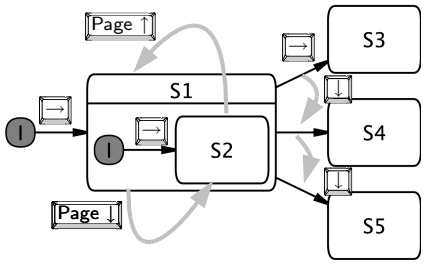


Figure 12: Navigation with key strokes using the macro-based modeling approach

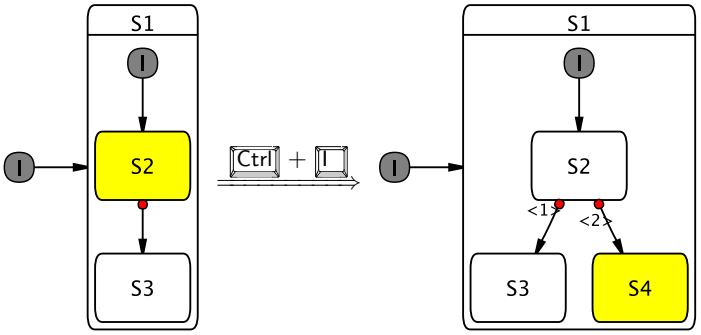


Figure 13: An editing action using the macro-based modeling approach

2. the up/down key navigates among sibling elements (*e. g.*, multiple outgoing transitions from a state object), and

3. the page up/down keys navigate up respectively down in state hierarchies.

Figure 12 illustrates some navigation examples.

Concerning editing step 2, the selection of an editing schema, the designer may select a schema from a pull-down menu or by pressing a keyboard shortcut. *E. g.*, in KIEL *Ctrl+I* generates a new successor state with a connecting transition and adjusts, if necessary, the priorities associated with other transitions originating from the selected state. Figure 13 shows an example of applying the "insert simple successor state" schema. Before applying the schema, state *S2* is selected afterwards the inserted state *S4* remains selected for further editing operations. After the application of the editing schema, a rearrangement of the Statechart elements will be performed automatically, in accordance with the SNF.

3.4 TEXT-BASED EDITING

As an alternative to the graphical modeling, one can also develop reactive systems using textual notations. As described in Section 2.4 a couple of languages exist that either describe Statecharts directly (*e. g.*, SCXML) or indirectly (*e. g.*, Esterel). Consequently the developer of reactive systems may choose between the textual and the graphical approach to specify systems. They offer the same expressiveness and the same level of abstraction. However, there are notable differences in terms of practical use, and both approaches have their benefits.

3.4.1 *Comparing Textual and Graphical Editing*

Graphical models are typically created with WYSIWYG editors. They benefit from intuitiveness and are good for higher level context. Statechart modelers often emphasize the usefulness of a deeper insight into a Statechart model. Quoting a practitioner:

> *"[. . .] tool developers should improve the integration of visual and textual languages. Reviewing the behavior of a detailed Statechart requires the reader to understand not only the visual representation, but also the action code in transitions, choice points, operations, states and classes. The reader should be able to easily browse this code to avoid missing or misunderstanding it."* Heidenberg et al. [74]

Textual languages can represent precise details very well; furthermore, they permit powerful macro capabilities, *e. g.*, using generic scripting or preprocessing languages such as Perl [155] or M4 [128] and allow a detailed revision management.

Editing Effort

Entering textual programming code to specify a system is very efficient. Due to the linear text flow, the insertion and deletion of symbols and words is very simple. Regarding graphical models, the two-dimensional nature brings some editing complications. Before inserting elements, the modeler must often "make room" first. The deletion of elements is also tedious, because it often leaves distracting "holes." To handle this, and to produce good

Graphical modeling tools do not provide sufficient support for rearranging graphical objects.

and readable graphics, adjacent elements have to be moved, while respecting the relationships of interacting elements (*e. g.*, transitions, state hierarchy, concurrency).

To illustrate this point, consider the canonical example *ABRO* [18], which is specified as follows: The system concurrently waits for two input signals, *A* and *B*. When both signals have occurred, the output *O* is emitted. This behavior is reset by the input signal *R*. An Esterel program expressing this behavior is shown in Figure 14a, and an equivalent Safe State Machine (SSM) representation is shown in Figure 14c. Now, suppose we want to extend this example with a further accepting signal *C* parallel to the signals *A* and *B*. To extend the Esterel version accordingly into the program shown in Figure 14b, we just use our favorite text editor, move the cursor to the "*]*", and type "|| *await C*." Performing the same operation on the corresponding SSM, to obtain the SSM seen in Figure 14d, is rather more involved: we have to enlarge the top-level state and the states *ABO* and *AB*, we have to move the state *Program_Terminated*, we have to draw a new horizontal line, we have to draw the new state *C* and its predecessor and successor, we must draw two new transitions and label them—an operation likely to take an order of magnitude longer than the corresponding textual edit. Furthermore, the result is unlikely to be as precisely laid out as the one shown here, unless one applies further alignment operations—provided that such alignment tools exist in the modeling tool in the first place.

Revision Management

Graphical modeling tools do not support graphical version management.

When large repositories of textual code are developed, evolution is well traceable. At each milestone of a project, one can obtain revealing information about the increments of the programming work (*e. g.*, applying the Unix *diff* utility to compare different versions). Only few modeling tools (*e. g.*, the *SCADE Suite* [40]) provide the feature to compare different model versions. If a modeling tool provides such a revision management, then only the model structure is compared but not the accompanying changes of graphical information.

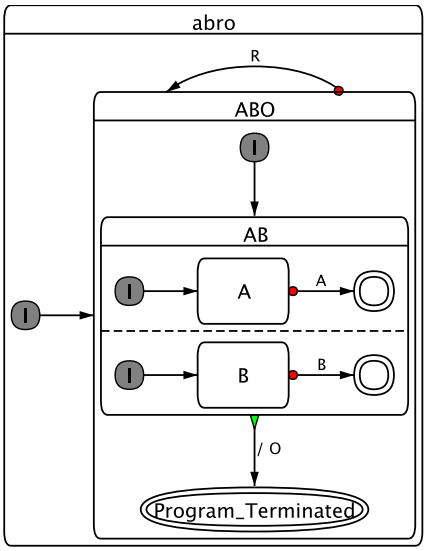
The modeling tools store graphical models as ASCII. These representations do not only include the relevant structural information that is of interest to us, but also information pertaining to time of creation, positions of elements, a multiplicity of used fonts and

```
module ABRO:
input A, B, R;
output O;
loop
  [
    await A
  ||
    await B
  ]
  emit O;
each R
end module
```

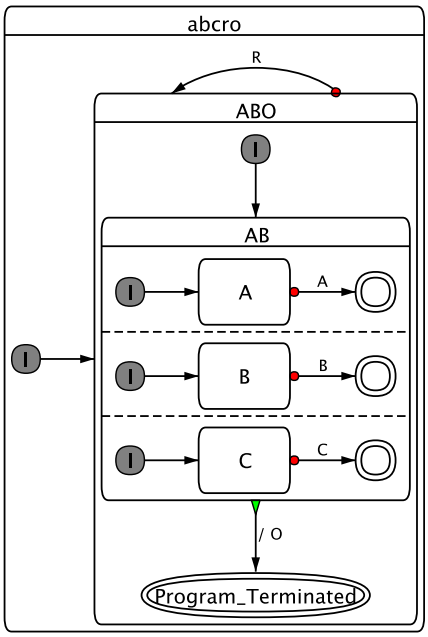
(a) Original Esterel fragment

```
module ABRO:
input A, B, R;
output O;
loop
  [
    await A
  ||
    await B
  ||
    await C
  ]
  emit O;
each R
end module
```

(b) Extended Esterel program



(c) Original SSM



(d) Extended SSM

Figure 14: Extending *ABRO* to *ABCRO*, Esterel and SSM versions

```
9,10d8
<   ||
<     await C
```

(a) *diff* applied to Esterel files

```
1c1
< #     Model of type
     Document saved by /
     home/esterel/
     EsterelStudio
  -5.2/bin/estudio.exe
       [11/18/2005
       10:39:01]
---
> #     Model of type
     Document saved by /
     home/esterel/
     EsterelStudio
  -5.2/bin/estudio.exe
       [11/18/2005
       10:40:03]
161c161
< {115
---
> {295
227c227
< AT    107 145
---
> AT    197 145
243c243
< [E]
---
> [V105 E E]
344a345,519
> NODE  init.2  init
> ATTRIB
> {[]
> ""
>
> {0
> 0
> }
> {
> {"helvetica"
> 12
> 1
> "Blue"
> }
> {"helvetica"
> 12
> 1
> "Blue"
> }}<false>
> }
> AT    170 35
> END # of init.2
>
> NODE  state.4 state
> ATTRIB
> {""
> <halt>
> <no>
> ""
>
> {30
> 30
> 0
> 0
> 1
> "1 0 0 1 0 0"
> 0
> 0
> 0
> 0
> 0
```

(b) *diff* applied to the editing results of SSM files (*scg* format, as used by Esterel Studio)—only first 60 of 289 lines shown

Figure 15: Results of applying *diff* to compare *ABRO* with *ABCRO*, alternatively using the Esterel representation and the SSM representation

color specifications. This extra information is apparently necessary to represent a SSM, but is not intended to be read by humans. As a result, the revision management for graphical models is not very practicable. To illustrate this, Figure 15 shows the differences (produced by *diff*) between *ABRO* and *ABCRO*. For the Esterel representations, the difference file is three lines long, and we can immediately deduce how the programs differ. For the SSMs, the difference file is orders of magnitude larger and so cluttered that it is practically worthless to the human examiner.

Meta-Modeling

One would often like to express models in a generic fashion at an abstraction level that is higher than what is directly supported in the (textual or graphical) modeling language. A trivial example is to extend *ABRO* from two to n signals that are awaited. For another example, one might want to model a distributed system communicating via some protocol with n stations. An example is the Token Ring Arbiter used in the benchmarks in Lukoschus [95], with up to 1000 stations. It is generally rather easy to achieve this with textual languages that have standard macro capabilities, using generic scripting or preprocessing languages such as Perl or M4, or just a powerful text editor. Graphical languages may also have macro capabilities, but creating a top-level system with n stations still requires manual work for each instance.

Graphical modeling tools to not support "scripting".

To summarize, both textual and graphical languages have their specific domains and advantages. The traditional model-based design flow starts with the graphical entry of a system model from which textual programs are synthesized; however, as we argue here, it would actually be advantageous to allow the designer to work in the opposite direction as well. The following section describes our idea of a Statechart description language.

The combination of both—the textual and the graphical language—would benefit the Statechart modeling process.

3.4.2 *A Statechart Desciption Language*

The macro-based Statechart editing (cf. Section 3.3) works directly on the Statechart topology, combining several simple editing actions. Another alternative structure-based Statechart editing technique that we propose is the textual Statechart structure description. The modeler describes only the Statechart structure using a textual Statechart description. We show in Chapter 7 that this modeling approach can produce graphical Statecharts applying our layout method proposal (see Section 3.2) very efficiently.

Several works in the area of programming languages deal with principles of readability and maintainability of programming languages in general (see *e. g.*, Richard and Ledgard [136] and McIver and Conway [103]). Furthermore, the Object Management Group (OMG) points out that Human-Usable Textual Notation (HUTN) underlies certain usability, syntax and aesthetic criteria and makes use of symbols and punctuation, reserved words, and user ex-

Criteria for an human readable and editable Statechart description language

pectations [116]. Having this in mind, in our opinion, a human readable and editable Statechart description language should meet the following demands:

IDENTIFIABLE STATECHART STRUCTURES: In the initial phase of a Statechart modeling process, the modeler often takes resort to paper and pencil. One can observe that the typical Statechart structure (*i. e.*, states and connecting transitions, state hierarchy, *etc.*), is initially developed in the mind. It should be the intention of a Statechart description language to aid the modeler expressing Statechart structures (*e. g.*, by making use of secondary notation, such as indentation.)

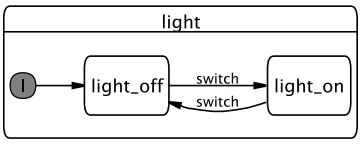
CONCISENESS: An easy-to-use Statechart description language should prevent the modeler from using many keyboard inputs; it should describe Statechart elements with short and expressive phrases.

COMPREHENSIBILITY: A Statechart description language should permit a modeler to describe the Statechart topology in an intuitive way. The consultation of a language manual should be avoided as far as possible. This can be achieved by a plausible language structure.

CROSS REFERENCES: Central elements of Statecharts are states and transitions. Some Statechart description languages subsume states as also transitions in separate sections (cf. Figure 16c). However, *implicit* declarations—as *e. g.*, realized by Argos (cf. Figure 16e)—should be used for an efficient modeling process.

Characteristics of XML notations

In general, much work is done on textual Statechart languages which are easy to understand by computers, but only limited work has been carried out on describing Statecharts in a human comprehensible manner. One important example of a structured Statechart notation is the Statechart XML (SCXML) [171]. This notation was developed based on the XML. The XML's regular structure allows the automatic validation as to whether an XML document conforms to constraints expressed by a definable schema or not. Central to XML notations is the content structuring using *tags* and hierarchical dependencies (cf. Figure 16b). The SCXML is a typical example of a good readable, but laborious to write notation especially when the



(a) A simple graphical Statechart example of a light switch. The system waits for the signal *switch* to enter the state *light_on* or *light_off*.

```
<?xml version="1.0" encoding="us-ascii
     "?>
<scxml version="1.0" xmlns="http://www
     .w3.org/2005/07/scxml">
  <initial>
    <transition>
      <target next="light_off"/>
    </transition>
  </initial>

  <state id="light_off">
    <transition event="switch">
      <target next="light_on"/>
    </transition>
  </state>

  <state id="light_on">
    <transition event="switch">
      <target next="light_off"/>
    </transition>
  </state>
</scxml>
```

(b) SCXML notation of the Statechart in Figure 16a

```
STATECHART:
  I [DS]
  light_on
  light_off

TRANSITION:
  S: I
  N: light_off

TRANSITION:
  S: light_off
  N: light_on
  E: switch

TRANSITION:
  S: light_on
  N: light_off
  E: switch
```

(c) SVM notation of the Statechart in Figure 16a

Figure 16: Different notations of a simple light switch Statechart

Statechart becomes complex. Instead of manually writing XML like notations, specific XML editors are used. These support the editing process using XML schemata.

Characteristics of the SVM notation

The Statechart Virtual Machine (SVM) notation [42] was developed as a notation for a Statechart simulator. The "transition centric" view (cf. Figure 16c) of the Statechart structure makes it easy to specify an execution model. But we see the transition-centric view as a difficulty to understand a state-based system. Like the SVM the UML on the fly Model Checker (UMC) notation [61] primarily describes state transitions The UMC notation provides definition sections for states, signals, and transitions (cf. Figure 16d). In contrast to the SVM notation, *e. g.*, transitions are subsumed in *one* transition section. Transitions are expressed by an arrow (->), which connects state expressions. However, the distribution of the specified State-

Characteristics of the UMC notation

```
Class light is
Signals: switch
State top = I,  light_on,  light_off
Tronsitions:
 I -> light_on
 light_on -(switch)-> light_off
 light_off -(switch)-> light_on
end
Object my_light : light
```

(d) UMC notation of the Statechart in Figure 16a

```
targos
main light (switch) ()

process light (iswitch) () {
  controller{
    init i

    light_on {}
    -> light_off with iswitch;

    light_off {}
    -> light_on with iswitch;
  }
}
endtargos
```

(e) Textual Argos notation of the Statechart in Figure 16a

```
statechart light[model="Esterel Studio";version="5.0";]{
 input switch;
 {
  ->light_off;
  light_off->light_on[label="switch"];
  light_on->light_off[label="switch"];
 }
}
```

(f) KIT notation of the Statechart in Figure 16a

Figure 16: Different notations of a simple light switch Statechart

chart elements in dividing sections limits the comprehensibility of the notation.

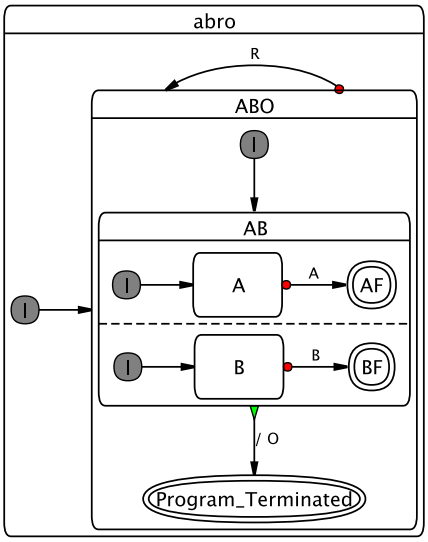
Characteristics of the Argos notation

The Argos notation developed by Maraninchi and Rémond [98] constitutes the most efficient notation of the above introduced approaches. It omits the usage of dividing sections and provides a "state centric" structure (cf. Figure 16e). Curly braces identify state hierarchy and outgoing transitions are dedicated to their source state. These language characteristics make the described Statechart intuitively comprehensible. Despite this, unnecessary keywords like *e. g.*, *targos*, *process* and *controller*, and the construction of the initial state, slightly increase the effort of writing a Statechart description.

A very simple but efficient language for specifying simple directed graphs is the *dot* language [55]. However, the hierarchy

```
1  statechart abro[model="Esterel Studio
       ";version="5.0"]{
2   input A;
3   input B;
4   input R;
5   output O;
6   {
7    ->ABO;
8    ABO{
9     AB{
10      ->A;
11      A->AF[type=sa;label="A"];
12      AF[type=final];
13     ||
14      ->B;
15      B->BF[type=sa;label="B"];
16      BF[type=final];
17    };
18    ->AB;
19    AB->Program_Terminated[type=nt;
          label="/ O"];
20    Program_Terminated[type=final];
21   };
22   ABO->ABO[type=sa;label="R"];
23  };
24 };
```

(a) KIT description representation.



(b) SSM representation.

Figure 17: Textual and graphical representations of the *ABRO* example [17].

mechanism of *dot* is not applicable to the requirements of Statecharts.

We devised the simple and efficient Statechart description language KIEL Statechart Extension of Dot (KIT) (see Figure 16f for a simple example). It combines implicit declarations as used in *dot*, the hierarchy construction as used in textual Argos, and the orthogonal construction as used in Esterel [20] with the ability to describe different dialects of Statecharts. Wischer [169] provides further details on the KIT language.

The KIT language is a very efficient notation for the specification of Statechart structures.

Figure 17 presents a KIT example with the equivalent graphical model of an SSM, the Statechart-dialect implemented in Esterel Studio. Figure 17a lists the KIT code, which is shortly described in the following. The Statechart preamble is listed in Line 1, containing the Statechart name and the model type and version, which determine the Statechart dialect and the accompanying graphical Statechart representation of the targeted modeling tool. Lines 2–5 declare the signal events. Ensuing, Lines 7–22 declare Statechart

elements and their relations. State objects are implicitly identified by their state names (cf. Line 8), curly braces define the scope of hierarchical relations (*e. g.*, state *AB*, cf. Line 9–17), transitions are written as "->" (cf. Line 11), and the || operator denotes parallel regions (cf. Line 13). KIT includes a couple of shorthand notations; *e. g.*, a transition without a source node determines an initial connector (cf. Line 7), a transition of type *sa* abbreviates the SSM's *strong abortion* (cf. Line 11).

Comparison of different Statechart description languages

Figure 18 compares the characteristics of the KIT language and the above mentioned languages according to the demands on Statechart description languages. The textual notations in Figure 16b to Figure 16d demonstrate different usability properties depending on the notations application area. As one can observe, the KIT language fulfills our demands based on the need of a Statechart description language to the greatest possible extent. We have already mentioned the problem of concise language structures in KIT. Therefore, we provide short versions of declarations consisting of only two characters, *e. g.*, "*sa*" instead of "*strong abortion*" (see above). However, some structures—*e. g.*, the property names—could be expressed in shorter manner (*e. g.*, "*t*" instead of "*type*" and "l" instead of "label"). However, these constructions are at the expense of KIT's intuitiveness.

3.5 SYNTHESIZING GRAPHICAL MODELS

As mentioned in Chapter 1, for the development of reactive systems, one can choose from a number of visual and textual languages. Esterel is a language for the specification of reactive systems behavior, which is—in contrast to the above mentioned textual languages—not state based. In this section we present an approach to transform Esterel v5 programs into equivalent SSMs. This permits a design flow where the designer develops a system at the Esterel level, but uses a graphical browser and simulator to inspect and validate the system under development. We synthesize SSMs in two phases. The first phase transforms an Esterel program into an equivalent SSM, using a structural translation that results in correct, but typically not very compact SSMs. The second phase iteratively applies optimization rules that aim to reduce the number of states, transitions and hierarchy levels to enhance readability of the SSM. As it turned out, this optimization is also useful for the traditional, manual

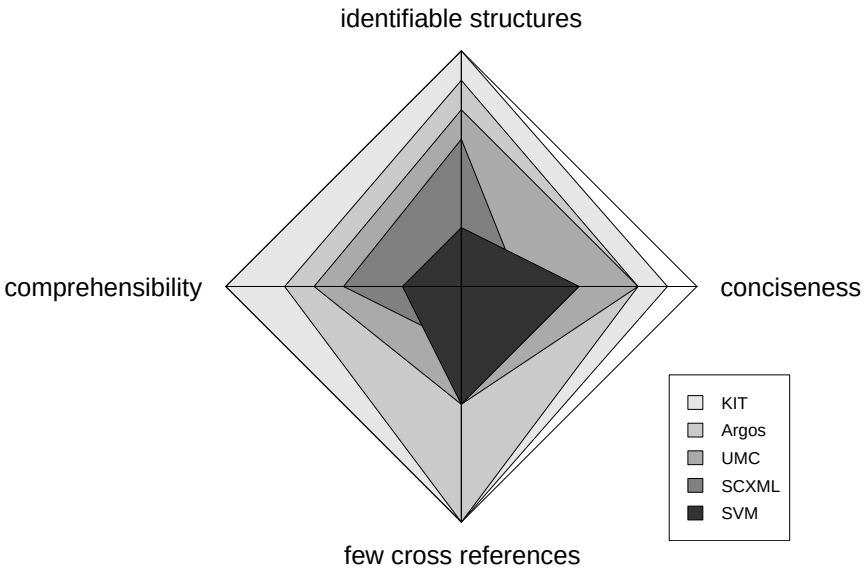


Figure 18: Comparing characteristics of textual Statechart description languages. A larger surface area denotes better conformance to the principles of language ergonomics.

design of SSMs. The whole set of transformation and optimization rules is described by Kühl [90].

Overview of Esterel

Esterel is an imperative synchronous language and consists of a set of kernel statements from which other statements are derived [19]. In contrast to other transformations and analyzes on Esterel, we should not confine ourselves to these kernel statements, because the derived statements give us useful information about the structure of the program.

A short introduction into Esterel

For a better understanding of the transformation, let us briefly review some Esterel basics; for a more detailed reference, see Berry [18]. Different parts of an Esterel program communicate via *signals*, which have a boolean status. *Valued signals* can also carry an additional value, like an *integer* or *real* number, or a value of user-defined type. The execution of Esterel programs is based on *instants*, and the execution of all statements is considered to take zero time and to take place within an instant, except for the *pause* statement, which explicitly waits for the next instant. As a consequence, no signal can change its status during one instant. This is even true in the case of valued signals; multiple emissions of

one valued signal in an instant can be combined by an associative and commutative function. The set of active input and output signals in one instant is called *event*. Important features of Esterel are the direct support of concurrency and multiple forms of preemption. Statements can either be *strongly aborted*, *weakly aborted*, *i.e.*, they are still executed during the abortion instant, or *suspended*, *i.e.*, they are frozen in this instant, but may resume later on. The preemption can either be *immediate* or *delayed*. In the immediate version, a statement can be preempted in the same instant it becomes active, while in the delayed version, it must be active for at least one instant before it can be preempted. Another important feature is the possibility to explicitly wait until a signal becomes present (or absent); this can also be either immediate or delayed. In addition to weak and strong abortion, Esterel provides *traps* as an exception mechanism. Traps are declared by the *trap* statement, which can optionally be augmented with one or more exception handlers. Traps are *raised* by the *exit* statement. We will discuss the behavior of traps in more detail in Section 3.5.3.

3.5.1 *From Esterel to SSMs*

The transformation starts with a textual macro state.

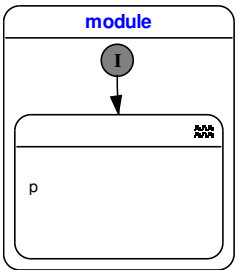
Our transformation from Esterel to SSMs is defined with a graph grammar, where non-terminal symbols are textual macro states. These are transformed into graphical macro states, which themselves may contain further textual macro states. For each Esterel statement, one rule exists to transform it into one macro state, where each sub-statement becomes a substate; thus the hierarchy of the SSM corresponds exactly with the nesting of statements in the Esterel program. A graphical macro state enters a terminal state if and only if the corresponding Esterel statement terminates. Since Esterel and SSMs are closely related, the transformation of most language constructs is fairly straightforward. Care has to be taken to preserve the semantics of traps, which do not have a direct counterpart in SSMs.

We now illustrate the transformation scheme by first presenting the transformation rules that are needed for the *ABRO* example and then applying theses rules to the Esterel version of *ABRO*. In the following, p and q stand for arbitrary Esterel statements; *de* is an arbitrary delay expression, *i.e.*, an expression over signals plus the additional information whether this expression shall be

evaluated immediately or in the next instant; *s* and *t* are arbitrary signal and trap names, respectively; *exp* is a signal value expression; *ehe* is an exception handler expression, *i. e.*, a boolean expression over trap signals. For details on the grammar of Esterel, see the Esterel manual [18].

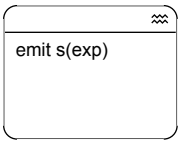
Transformation Rule 1 (*module*)

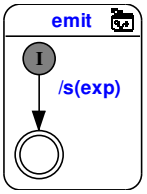
```
module mod_name:
  input I1,...,In;
  output O1,...,Om;
  p
end module
```

An Esterel program (a "module") starts with an interface declaration. Thus, the first step of the transformation is to generate an SSM with the same name and interface, which simply enters a textual macro state that contains the program body *p*.

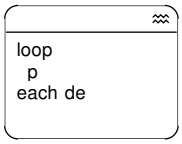
Transformation Rule 2 (*emit*)

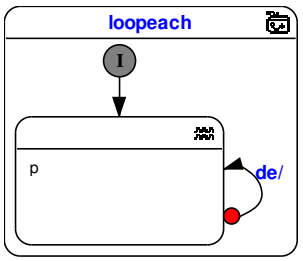




The *emit* statement broadcasts a signal *s*, with an optional value *exp*, and terminates instantaneously.

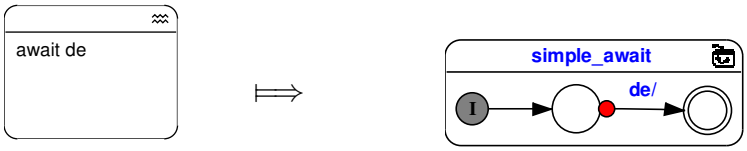
Transformation Rule 3 (*loop each*)





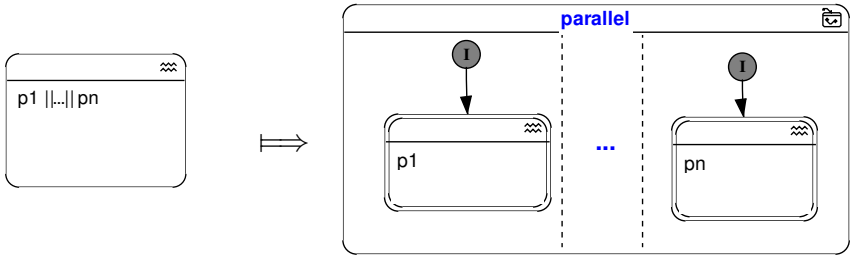
The *loop p each de* statement executes its body *p* and restarts it whenever the delay expression *de* is true.

Transformation Rule 4 (*simple await*)



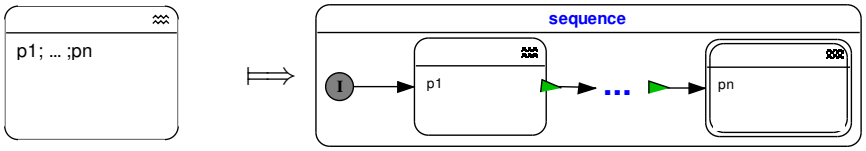
In its simple form, *await de* waits until the expression *de* evaluates to true and then terminates. The general case (see Kühl [90]) allows to wait for different events and executes a specific code depending on which event occurred first.

Transformation Rule 5 (*parallel*)



The parallel operators in Esterel and in SSMs work exactly in the same manner. The parallel branches are executed synchronously. Note that the parallel terminates if all its sub-statements terminate; therefore, all contained macro states are final.

Transformation Rule 6 (*sequence*)



The sequence waits for termination of one statement before it starts executing the next statement. Note that a sequence terminates when its last sub-statement terminates.

Transforming ABRO

A demonstration example

The rules introduced so far suffice to generate an SSM for *ABRO*. The first step is to apply the (*module*) rule to the Esterel program that is to be transformed (Figure 14a, page 43), which results in an SSM with the same interface declaration as the Esterel program

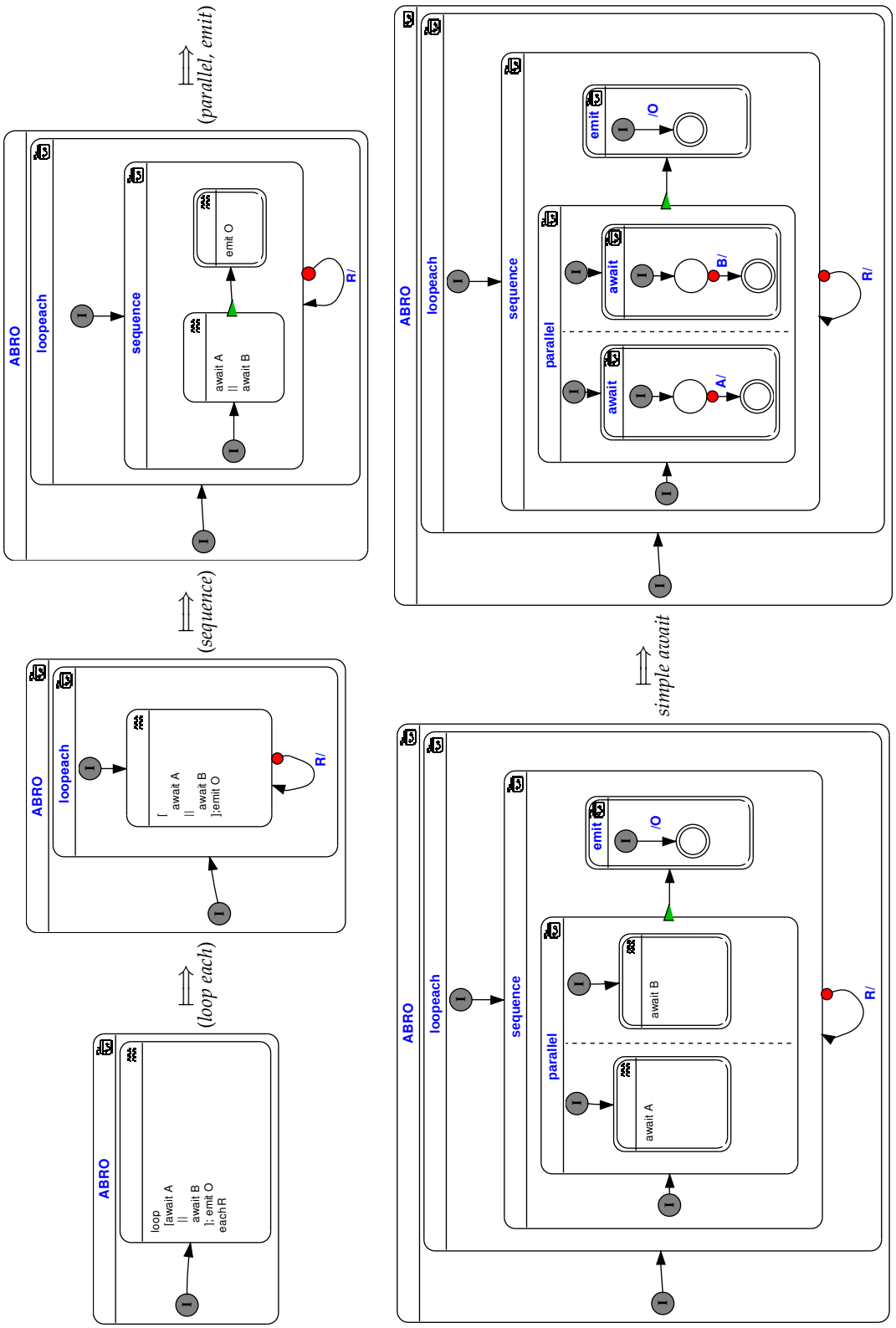


Figure 19: Stepwise transformation of the Esterel program *ABRO* (Figure 14a) into an equivalent SSM

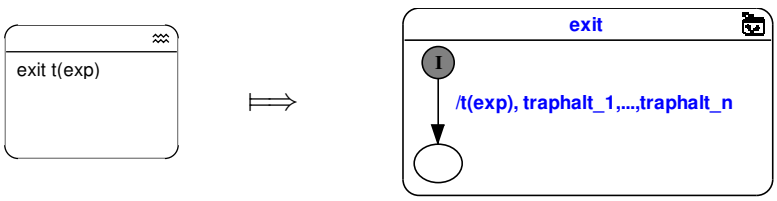
and a textual macro state that contains the body of the given program. Figure 19 shows this SSM and the subsequent stepwise transformation into an SSM that contains no textual macro states anymore, hence no more transformation rules are applicable. The behavior of the original Esterel program is completely preserved in the resulting SSM; but we also see that the generated SSM contains unnecessary hierarchy nestings and is relatively hard to read. However, before considering optimizations in Section 3.5.2, we first consider some of the remaining transformation rules. For a complete list see the diploma theses of Kühl [90].

Handling Traps

Handling of Esterel specialties

Traps are a part of Esterel and their translation to SSMs is not straightforward. Their behavior must be simulated using local signals and weak abortions. For each *trap* statement we introduce local signals for all declared exceptions, plus one new local signal *traphalt*. This signal is emitted, if another trap with higher priority is activated from inside the trap scope. This is needed to assure that no exception handler is executed when an exception with higher priority is active at the same time; our translation handles the arbitrary nesting of traps. Whenever an exception is raised, all other exceptions with its scope that are not running parallel to the statement that raised the exception are deactivated. We will discuss this in detail in Section 3.5.3.

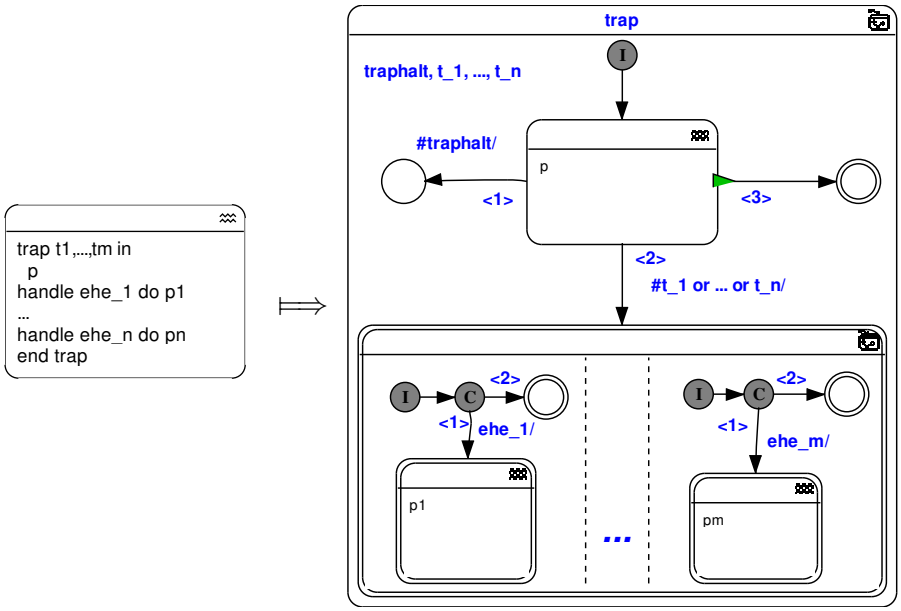
Transformation Rule 7 (*exit*)



The *exit* statement raises an exception, optionally annotated with the value of an expression, which can be used in an exception handler. The SSM also includes signals to prevent the execution of exception handlers with lower priority (see also Section 3.5.3). Note that the *exit* state does not terminate normally, hence the simple state that is entered is not marked as final state. The body of a *trap* statement is weakly aborted when one of its exceptions is raised. When no other exception with higher priority is active, its active

handlers are executed in parallel. When a trap with higher priority is active at the same time, no handler is executed. A trap terminates when its body terminates.

Transformation Rule 8 (*trap*)



The *trap* catches exceptions which are raised inside its body. Depending on which exception was raised, different handlers are executed—possibly in parallel, when multiple exceptions were raised.

A requirement placed on Esterel programs is that they must not contain *instantaneous loops*; *i. e.*, the body of a loop is not allowed to terminate instantaneously. This requirement also transfers to SSMs; they must not contain instantaneous cycles. This causes a complication in our transformation of traps, since replacing traps by weak abortions can compromise the ability of a compiler (or simulator) to establish that there are no instantaneous loops in a given model. A weak immediate abortion is always assumed to be potentially instantaneous, whereas for a trap the Esterel compiler analyzes whether it may be raised immediately or not. A justification for this is that the scope of trap signals (*exceptions*) is confined to just the trap, whereas weak aborts may be triggered by signals with arbitrary scope. Consider the example shown in Figure 20a; here, the Esterel compiler determines correctly that within the loop, a *pause* statement must be executed before the exception T is raised, and therefore, the loop is not instantaneous. However, the trap

with a weak abort construction would be replaced by an immediate trigger, the compiler would claim that the loop is potentially instantaneous. Hereupon it would reject the program—even though the program would be behaviorally equivalent.

There are several possibilities to resolve this dilemma. One approach would be to apply an analysis to weak abort blocks with immediate triggers to determine whether the trigger signals can possibly be emitted in the instant when the abort block is entered; if not, we can safely replace the immediate trigger with a delayed trigger, which would solve the problem. This, however, would make the handling of the *trap_halt* signals and of traps with multiple handlers significantly more complicated. Another approach is based on the observation that for loops that are not instantaneous, we can safely add a parallel thread to each loop body that pauses for one instant (and does nothing else). This is illustrated in the Esterel module shown in Figure 20b, which is equivalent to the example in Figure 20a. The second thread within the loop, which just pauses for one instant and then terminates, does not change the behavior of the program, but allows the compiler to establish that the loop is not instantaneous, since parallel statements terminate only (normally) if all concurrent threads have terminated.

The first of these approaches has the advantage that it does not enlarge the program; however, we opted for the second approach, as it is conceptually simpler. Figure 20c shows the SSM synthesized from the Esterel example in Figure 20a, and we can see that the macro state corresponding to the loop contains an extra thread that just pauses for one instant. However, to avoid excessive additions of such parallel threads, we limit this to loop bodies. Hereupon it depends on an enclosed trap whether the loop is instantaneous or not, as is the case in this example. These cases seem to be rare in practice, in our benchmarks only *ABCD* [37] contained such loops.

Handling Abortions

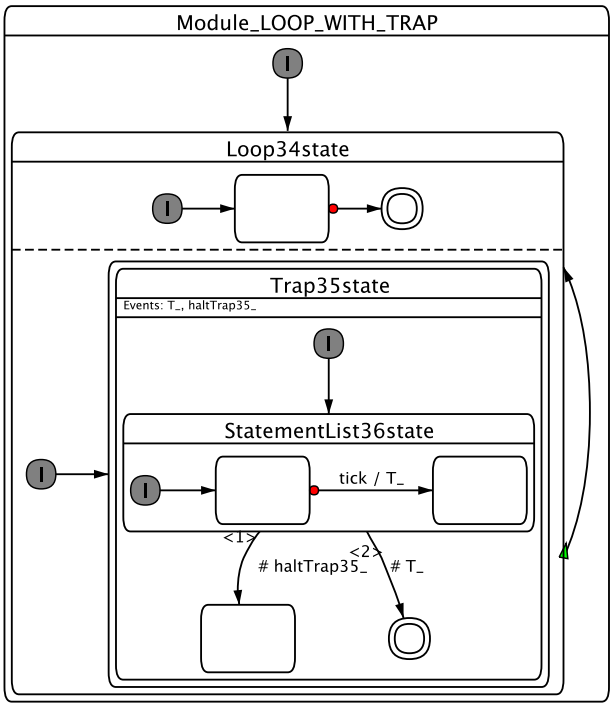
Abortions are closely related to traps, but since they can be directly expressed in SSMs, their transformation is much simpler. Both weak and strong abortions stop to execute their bodies when an abortion trigger is active. Depending on which trigger is active, different code parts can be executed.

```
module LOOP_WITH_TRAP:
loop
  trap T in
    pause;
    exit T
  end trap
end loop
end module
```

(a) Esterel program with trap

```
module LOOP_WEAK_ABORT:
loop
  signal T in
    weak abort
      pause;
      emit T
    when immediate T
  end signal
||
  pause
end loop
end module
```

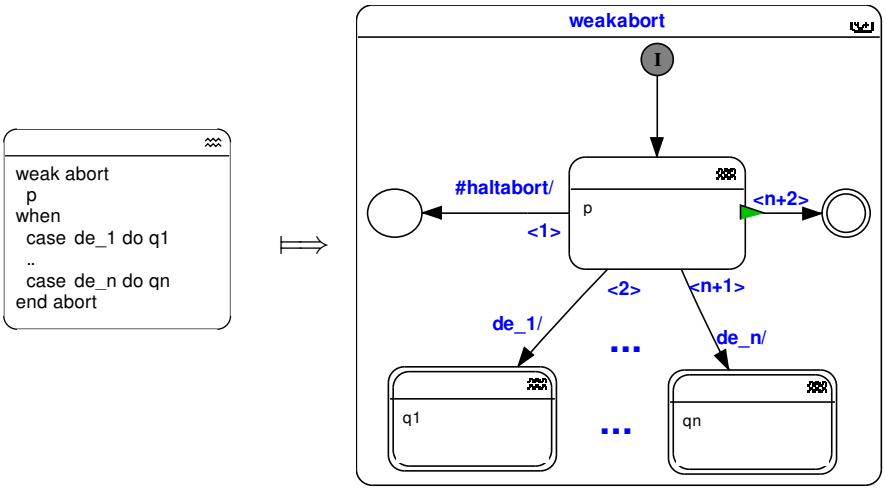
(b) Equivalent Esterel program with weak abort



(c) SSM synthesized for *LOOP_WITH_TRAP*

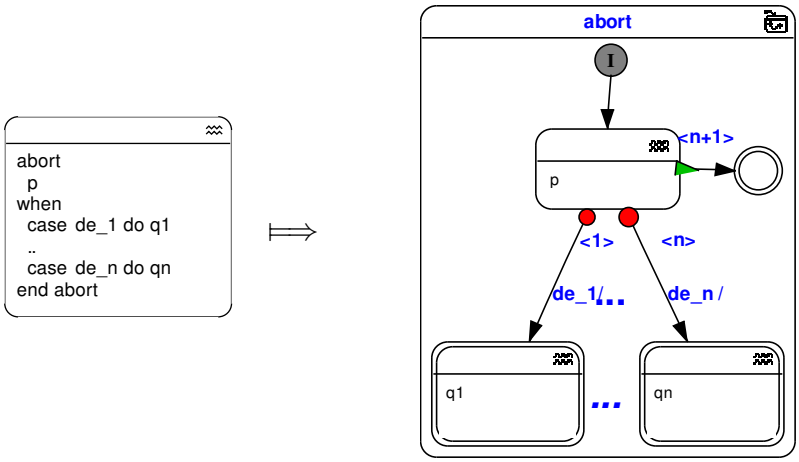
Figure 20: Handling of potentially instantaneous loops, introduced by the transformation from traps to weak abortions: the loop in the Esterel examples (Figure 20a and Figure 20b) is not instantaneous, which in the synthesized SSM (Figure 20c) is made explicit with an extra parallel thread that is not instantaneous

Transformation Rule 9 (*weak abort*)



For *weak abort*, the body is still executed in the instant it is aborted. This makes it possible to raise an outer trap, which has priority over the abortion; the code that follows the *weak abort* is not executed. This is assured by the *haltabort* signal, which is raised by the *exit* statement similar to the *traphalt* signals.

Transformation Rule 10 (*abort*)



In contrast, for strong abortion, no exception can be raised inside if the abortion condition is true, since the abortion body is not executed in this case. The same holds for other statements which are derived from strong abortion, such as *loop each* and *every*.

3.5.2 *The Optimization*

The optimization rules reduce the number of graphical elements after the transformation.

As we have seen, the transformation produces verbose SSMs, but they have a very regular structure. This makes it possible to obtain a readable chart by applying some simple optimization rules. The rules are completely syntactical and make no assumptions on the actual execution of the SSMs, *e. g.*, whether transitions can actually be taken. Each rule takes one macro state and transforms it. Neither information about a possible surrounding macro state nor about any substates is necessary. This is justified by the fact that we can replace any macro state by another one with the same observable behavior.

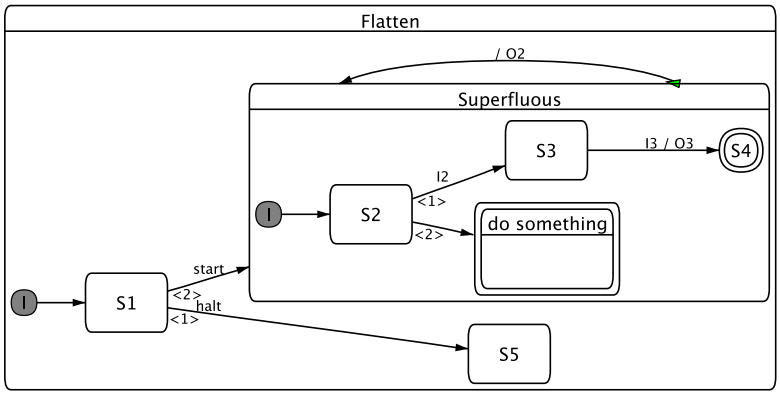
In the following, we assume that only states which can actually terminate, *i. e.*, have at least one terminal state, have a normal termination transition originating from it. Similarly, a final state inside a macro state without a normal termination is changed to a normal state. This constraint, imposed by Esterel Studio, assures that final states and normal terminations always match, which makes the chart easier to read. We also assume that states without an incoming transition, which therefore are unreachable, are removed. These conditions can easily be checked.

Flattening Hierarchy

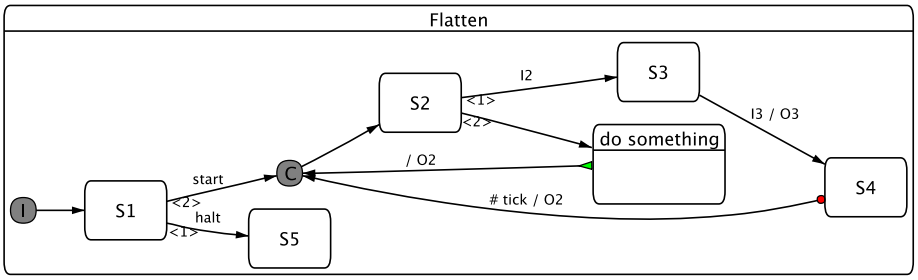
Since the state hierarchy is increased for every nesting of Esterel statements, the generated SSM contains usually many more hierarchy levels than necessary. In fact, one could also apply traditional Statechart flattening to remove *all* hierarchy levels; however, this can result in exponential state explosion. What we want to do is to remove hierarchy levels (macro states) if this actually reduces the overall number of states and transitions. There are two cases where we may remove macro states.

The first case is that a macro state cannot be preempted, does not declare any local signals, variables, or history, and is not a parallel state. This may for example be generated for sequences (Transformation 6). Such a state can be removed and replaced by its internal state transition graph. The initial state is replaced by a conditional pseudo-state, which connects all incoming transitions.

If no normal termination exists, we have to assure that the termination conditions do not change. This is done by changing the



(a) A Statechart with a superfluous macro state



(b) The same Statechart, after flattening

Figure 21: Example for removal of macro states

terminal attribute of inner states, depending on whether the macro state we intend to remove is itself a terminal state.

If a normal termination exists, every contained final simple state gets an abortion which leads to the target of this normal termination, with the trigger *immediate tick*, where *tick* is a signal which is present in each instant. Thus these transitions are always enabled. These states, as the conditional pseudo-states that replace the initial state, might be removed by further optimization steps. Removing these states immediately would require further information about the surrounding state, leading to more complex rules. For each final macro state, a normal termination is added, which leads to the same state and has the same effect as the normal termination of the surrounding state.

An example for the application of this rule can be seen in Figure 21. The state *Superfluous* is removed, and its initial state is replaced by a conditional node to which the transition from state *S1* leads now. The self loop is replaced by a strong abortion from

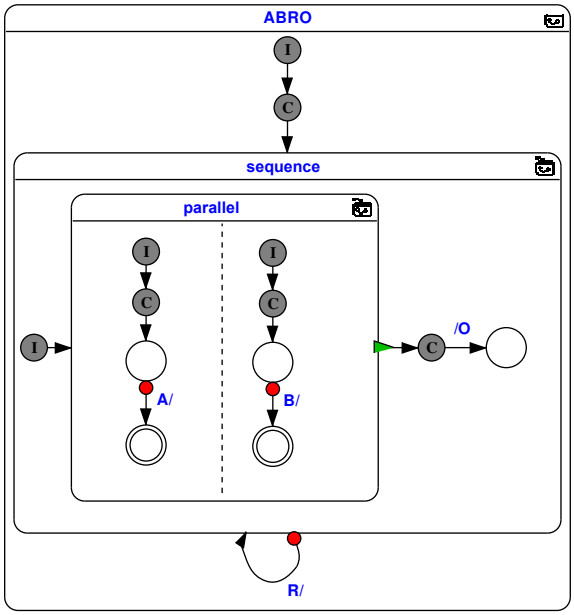


Figure 22: SSM for *ABRO* after unoptimized transformation (Figure 19) and subsequent flattening the *loopeach*, *await*, and *emit* states

state S_4 and a normal termination from the macro state *do something* to the same conditional. Neither S_4 nor the *do something* state are final anymore.

The second case is that a macro state contains only the initial state and one further state without any transitions except the initial one, as for example generated for *emit* statements (Transformation rule 2). Such a state can simply be removed. Again, the initial state is replaced by a conditional state. Local declarations of signals and variables are assigned to the surrounding state. Normal terminations are handled as in the first case, while other outgoing transitions are simply redirected and get the contained state as new source.

Applying these rules on the *loopeach*, *await* and *emit* states of the SSM for *ABRO* from Figure 19 yields the SSM in Figure 22, which already has a readable form, but is not optimal yet.

Removing Simple States

When removing macro states, no simple states are eliminated. The initial state is transformed into a conditional state, and simple final states, which are only needed to indicate termination, are

changed to transient, normal states. Both kind of states might be superfluous. We consider the following cases:

- If a conditional pseudo-state has just one outgoing transition, it can simply be removed. Such states are inserted into the chart when macro states that only have one initial transition are removed. We can use this rule to remove the conditional states from the *ABRO* SSM in Figure 22, which in this example suffices to produce an optimal chart.
- We can also remove simple states that have no internal actions and only one outgoing transition with trigger *immediate tick* and no condition. The incoming transitions are then redirected to the target of the unique outgoing transition.
- If a simple state has only one incoming and one outgoing transition that both have the same trigger and condition, it can be removed and the transitions be combined to one. This rule can only be applied if both transitions are delayed. Hence the incoming transition may not originate at an initial or conditional pseudo-state, since such transitions are always immediate. Furthermore, the state where the transition originates from must be a simple state without any internal actions.
- In SSMs, simple final states may neither have outgoing transitions, nor any internal actions. Thus, they only indicate the termination of the state and do not specify any further behavior. Therefore, all simple final states of a macro state are interchangeable and can be replaced by one.

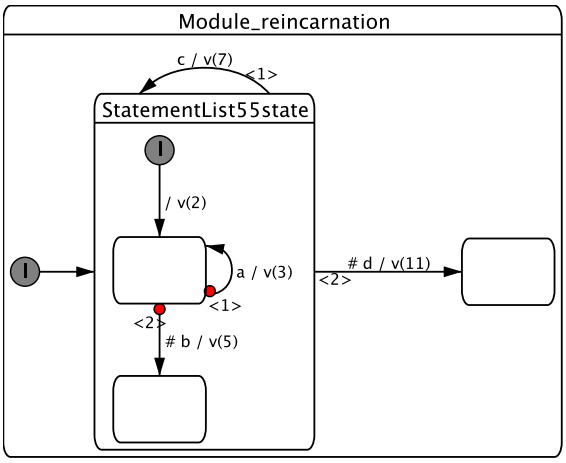
The iterative application of these simple rules on the generated charts produces in most cases well readable, relatively small charts. For the *ABRO* example from Figure 14a, the automatically synthesized SSM is, after optimization, identical to the SSM shown in Figure 17b, except for the naming of states. As another example, consider Figure 23, where the Esterel code generated for the reincarnation example from André [3] is transformed back to its original, terse SSM.

Note that even though these rules are motivated by the transformation from Esterel, they can be useful for SSMs in general. Unnecessary hierarchy and superfluous simple states can also be found in

```
module reincarnation:
input a, b, c, d;
output v := 1 : combine
     integer with *;

loop
  weak abort
    emit v(2);
    loop
      await
      case a do
        emit v(3)
      case immediate b do
        emit v(5);
        halt
      end await
    end loop
  when
  case c do
    emit v(7)
  case immediate d do
    emit v(11);
    halt
  end weak abort
end loop
end module
```

(a) Esterel source



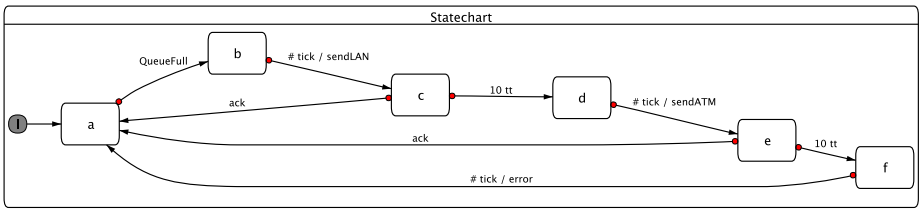
(b) Synthesized SSM

Figure 23: Transformation of the reincarnation example [3]

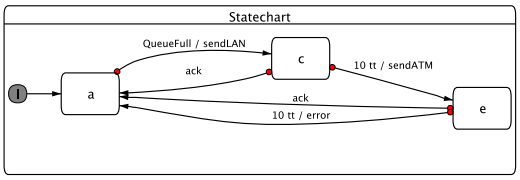
manually created charts. Especially novices tend to produce unnecessary large models with needless states, for example by splitting trigger and effect into separate transitions. An application of an optimization rule to a "real" SSM, which was not generated from Esterel but modeled by hand, can be seen in Figure 24. The Statechart which was developed as part of an automated subway control system [131]. In fact, in the original model (see Figure 24a), the modeler did not introduce an explicit *immediate tick*, even though he intended the states to be transient.

3.5.3 *Correctness of the Transformation*

Both the transformation and the optimization were tested with various Esterel programs. While this gave good confidence in the transformation process, it does not prove the correctness in general due to the possible complex interaction between the sub-statements of an Esterel statement. It has to be shown that the behavior of the generated SSM is equivalent to the behavior of the original Esterel



(a) A Statechart with unnecessary, transient states



(b) Statechart after removing the transient states *b*, *d*, and *f*

Figure 24: Example for removing transient states

program. It follows a sketch of an informal correctness proof. First, let us exclude traps from our considerations.

Esterel without Traps

For the Esterel kernel language, the control flow can only rest at explicit *pause* statements. Hence, a configuration of an Esterel program is a set of currently active *pause* statements, called *registers* in this context. For the Esterel kernel language without traps, the behavioral equivalence can be proven by structural induction over all Esterel programs, giving a bi-simulation between the active registers of the Esterel program and the active states of a stable configuration of the constructed macro state, *i. e.*, a configuration which can occur at the end of an instant.

Since both Esterel programs and SSMs are fully deterministic, both can be executed in lock-step. For each register of the Esterel program, exactly one non-final simple state is generated. Whenever this state is active, the corresponding register of the Esterel program is also active. When a simple final state is active, the Esterel statement that corresponds to the macro state has terminated, hence no register in it is active anymore. Most non-kernel statements preserve this correspondence. The expansion of *halt*, *sustain*, and *await* contain pauses. Accordingly, their transformation into SSMs produces exactly one non-terminal state.

The *loop each* and *every* statements contain registers which are not directly expressed in the SSM. Instead of waiting in an explicit simple state, the termination of the included macro state is not caught by a normal termination. For assessing correctness, both can be expressed by equivalent SSMs, which contain explicit states that are entered after the termination of the substate. These states correspond exactly to the registers of the Esterel program.

A little more involved is the behavior of *suspend*. The usual kernel statement *suspend p when S* executes *p* in the first instant regardless of the status of *S*. Therefore, a register inside the *suspend* statement is active whenever the suspend statement itself is active. This does not affect the bi-simulation. The statement

suspend p when immediate S

corresponds to

await immediate not S; suspend p when S.

The *await* contains an extra register which might be active even if *p* is not active. In the SSM, both behaviors can be modeled by a suspend transition of the macro state. If this transition is tagged immediate, this might lead to a macro state which is active, even though no substate in it is active. There is no simple state that corresponds to the new *pause* register. For the correctness proof, we would generate different SSMs for immediate and delayed suspension, where the immediate suspension contains an extra simple state, which waits for the absence of the trigger signal, before it starts the substate. The behavior of this macro state is equivalent to the one we actually generate in the transformation.

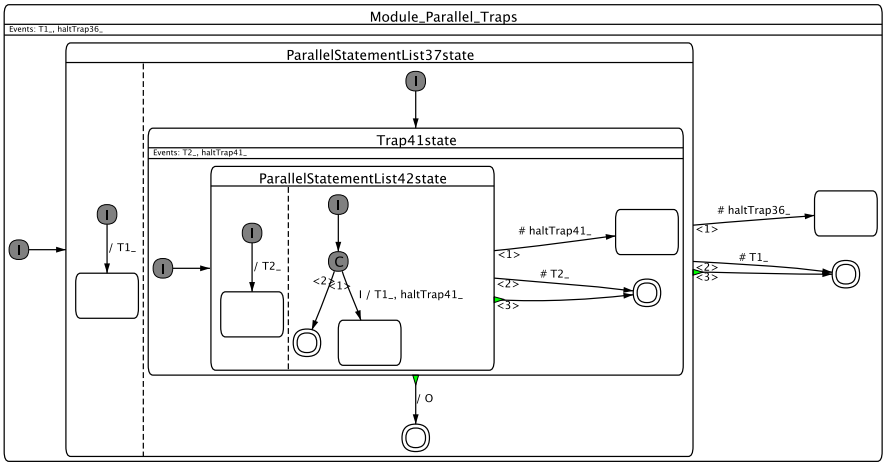
Traps

Raising an exception stops the execution of the control-flow of the current thread; however, all concurrent threads finish their execution for the current instant. This makes it possible that multiple, different traps are raised in the same instant. When multiple traps are raised inside a trap scope, only the handler of outermost trap is executed. If, however, a trap is raised in parallel to another trap-declaration, both handlers may be executed in the same instant.

Expressing *traps* in SSMs is similar to expressing traps by local signals and *weak abort* in the Esterel program, which is not trivial

```
module Parallel_Traps:
input I;
output O;
trap T1 in
    exit T1
  ||
    trap T2 in
      exit T2 || present I then exit T1 end
    end trap;
  emit O
end trap
end module
```

(a) Esterel version



(b) SSM version

Figure 25: Transformation example of nested traps into an SSM, illustrating the problems of replacing traps by signals

for the general case. This situation also arises, for example, in the Esterel derivate Quartz [146], which does not support traps directly, but weak and strong abortion. In contrast to signals, for exceptions it is important where they are raised, see also Figure 25a. Even though in this example the exception *T1* is active in any case, the second *exit T1*, which is guarded by *I*, determines the output of signal *O*. Therefore, it is in general not possible to replace each *trap* by only one local signal.

First, let us consider how this problem is handled in the context of Quartz. Since it is possible in Quartz to test which *pause* statements are active, preconditions on each raising of a trap can be computed. These preconditions can be used in an *abort* statement that replaces the *trap*, to test whether traps with higher priority were raised

inside. Thus, the trap determines which exits are relevant for it. We use a different approach, which assigns extra signals to trap that indicate that their handler may not be executed, because a trap with higher priority was raised inside. The information for which traps the halt signal must be emitted is assigned to each *exit* statement.

We also have to show that the combination of macro states derived from *trap* and *exit* statements behave correctly. First, we notice that the macro state derived from *exit* does not terminate. This assures that no statements in a sequence are executed later, while statements in parallel branches are executed normally. In the Esterel semantics, the raising of exceptions is usually encoded by their depth, *i. e.*, the number of *trap* declarations between the point where an exception is raised, and its declaration. When an exception is raised, this depth (+2, since 0 and 1 are used to encode normal termination and *pause*) is returned as completion code. This code is passed to the surrounding statements until it reaches a *trap*. Here the completion code is examined. If it is greater than 2, which indicates that it is not the corresponding *trap*, it is decreased. If it has reached the corresponding *trap*, this value has reached 2; the handler of this trap is executed. This scheme is exactly preserved by our transformation, which sends a halt signal for all traps between the *exit* statement and the corresponding *trap* statement. In the SSM example shown in Figure 25b, the *traphalt41_* signal determines whether the macro state for the inner trap terminates and the *O* is emitted.

This transformation at the Esterel level can be easily extended to compound statements. Therefore, also for weak abortion a halt signal is needed. This halt signal is only emitted from within an abortion. Observe that the *exit* that is used to implement the trap has always depth 2, *i. e.*, no *traphalt* signal needs to be emitted by a weak abortion.

Since a strong abortion suppresses the execution of its sub-statements in an instant where the abortion trigger is active, it can neither emit a halt signal, nor terminate in such an instant. Therefore, no halt signal is needed for strong abortion or statements like *every* and *loop each*, which are based on it.

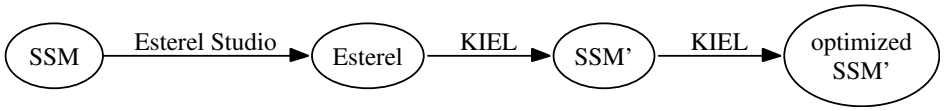


Figure 26: End-to-end validation of the transformation: From Esterel Studio SSMs to Esterel to KIEL SSMs

3.5.4 *Correctness of Optimizations*

It remains to be shown that the behavior of the SSM is not changed by the optimizations. Since the optimizations consist of iteratively applied single rules, it suffices to argue that applying one rule does not change the behavior of a chart.

Obviously, the removal of superfluous normal terminations, unreachable states, and the conversion of final states into normal states when no corresponding normal termination exists, does not change the behavior. The only crucial part in omitting hierarchy is that no signal declaration may be moved outside a self-loop. This is due to so called *schizophrenia* [19]; each activation of a macro state in one instant creates new local signals. Since we explicitly prohibit the removal of macro states that contain outgoing transitions and local signals, the behavior of schizophrenic signals is preserved.

The merging of final simple states is justified by restricting SSMs that a final state may neither have any *on exit* or *during* action nor outgoing transitions. Therefore, different final simple states are indistinguishable. Similarly, two simple states with the same outgoing transitions are merged only when the two states are indistinguishable. Then the correctness follows directly from the semantics of count-delayed transitions. The correctness of the removal of pseudo-states and simple states with trigger *immediate tick* and without a condition is straightforward.

3.5.5 *Experimental Validation*

In addition to the theoretical considerations above, we have validated the transformation and optimization process and its implementation by experimentally applying a round-trip tool chain that employs Esterel Studio's Esterel synthesis capabilities (see Figure 26). Starting with SSMs developed with Esterel Studio, one can employ Esterel Studio's Esterel code generator. From the resulting Esterel code, our transformation generates an equivalent SSM. After

optimization, the round-trip produces SSMs corresponding to those starting the round-trip. When starting from well-written SSMs, they tend to be identical.

Similarly, one may perform a round-trip synthesis at the Esterel level, synthesizing a given Esterel program into an SSM with KIEL, and then synthesizing the same SSM into Esterel using Esterel Studio. We did this for all of the transformation rules individually, using dummy expressions for sub-statements. Due to Esterel Studio's rather elaborate Esterel synthesis, the resulting Esterel programs were not identical to the original programs (*e. g.*, they contained a lot of spurious *nothing* statements), but they could relatively easily be proven to be equivalent [90] using the Structural Operations Semantics rules of Esterel's constructive behavioral semantics [19].

The complete transformation has been implemented in the KIEL modeling tool, which allows to demonstrate the practicality of this approach and the compactness of the generated SSMs. The KIEL modeling tool will be described in Chapter 6.

In Section 7.1 we present the results of an empirical study on the usability and practicability of the macro-based Statechart editing (see Section 3.3) and the text-based Statechart editing (see Section 3.4). The study also includes a Statechart layout comparison. The run-time analysis of our layout method using KIEL is presented in Section 7.2.1; KIEL's run-time for the SSM synthesis is presented in Section 7.2.2.

4

SIMULATING GRAPHICAL MODELS

In the previous chapter we described different methods to create and modify Statecharts which enable an efficient editing process of graphical models. Especially for complex SUDs these techniques can improve the developing speed. Once a system has been created, it usually needs to be simulated. In this chapter, we address the question of how to perform such simulations for complex Statecharts. Specifically, we present the concept of Dynamic Statecharts, which are a novel paradigm for visualizing a Statechart model under simulation.

Statechart simulation facilitates the development of a reactive system.

Statecharts are supported by several commercial tools; once a Statechart is drawn, it can be inspected by developers, and one can typically synthesize code from it. Furthermore, Statechart modeling tools generally support Statechart *simulation*, where the SUD is subjected to some input stimuli, and the Statechart model is animated according to the current configuration of the SUD.

Graphical languages are good for understanding structures, but limited for analyzing dynamics.

However, as mentioned in the introduction, these animation capabilities are so far rather limited; Statechart modeling tools often provide very restricted facilities to explore the structure and the behavior of complex Statecharts. The paradigm generally offered is that the Statechart is shown as drawn by the user, and active states and transitions are marked in a specific color. This is fine if the model is small enough to be entirely visible on the screen; it becomes problematic for realistic, larger models. Such Statecharts consist of a large number of states, transitions and inter-dependencies. There is no mechanism to automatically bring "active" parts of the model to the foreground, which gets further complicated by concurrency in the system.

This problem becomes even more serious, if a simulation steps through hierarchy levels. *E. g.*, a common method to display hierarchical states is to provide separate views for each hierarchy level. To inspect *e. g.*, a deeper hierarchy level, the modeler has to open the associated view *e. g.*, by clicking in the parent state. Thereupon, the parent view is replaced with the next deeper hierarchy level. Figure 27 demonstrates this state browsing behavior in modeling

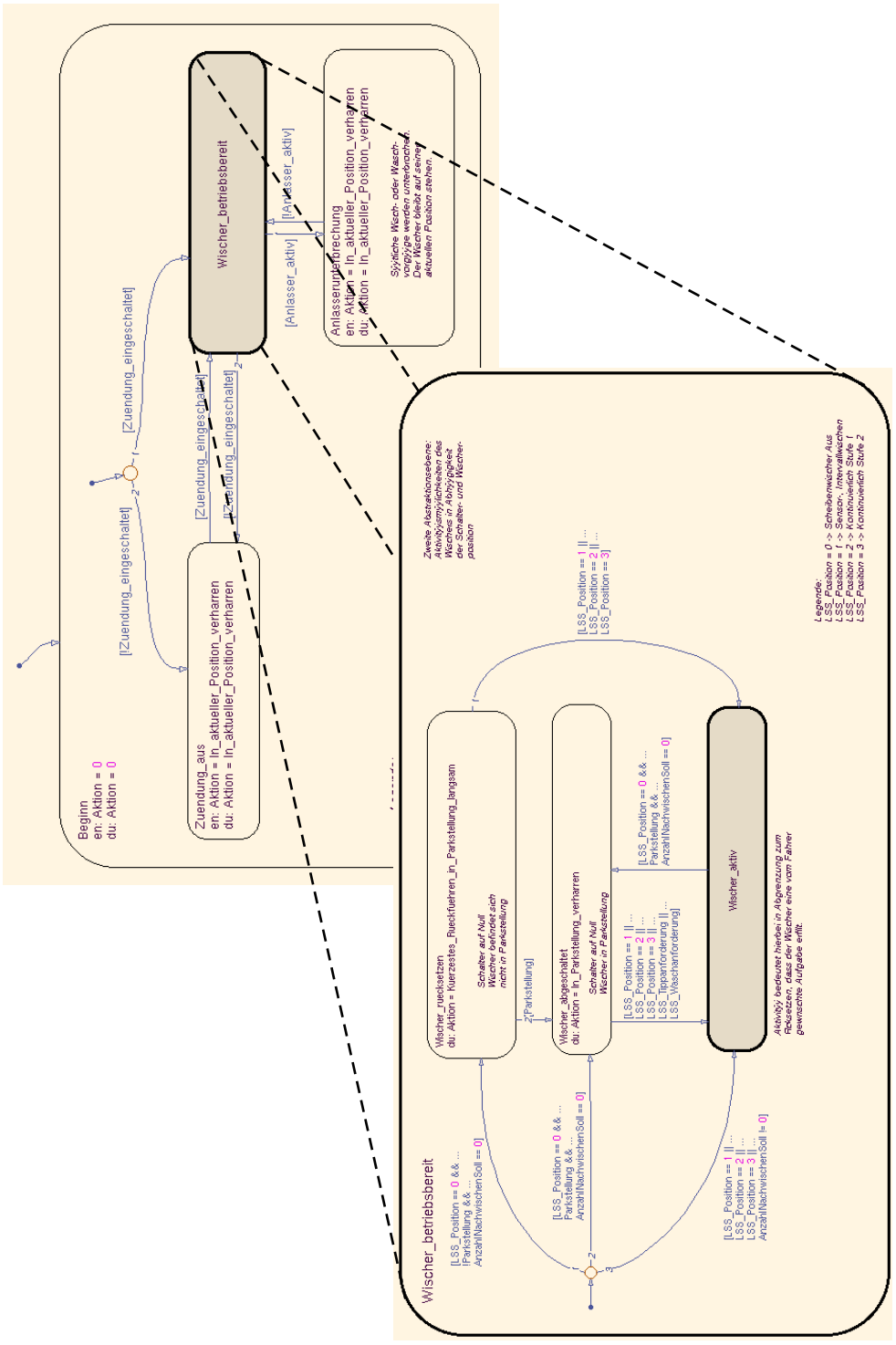


Figure 27: Changing of hierarchy level using Stateflow

tools (The figure depicts a window wiper system using Matlab Simulink/Stateflow; in Section 4.2 it is explained in detail.) In this case, it is very hard to follow the state transitions manually. For comparison, consider the world of debugging textual programs: when stepping through a program, it is a standard capability to automatically display the currently executed source code region. The source code may be spread through numerous files, but the debugger makes this transparent; it is not left to the user to open up the appropriate file to see how the program executes.

Heidenberg et al. [74] also observed this behavior of Statechart modeling tools and they state:

> *"Although it is possible to open different design windows showing different levels of a statechart, readability still suffers as a lot of screen space is wasted by displaying the different hierarchy levels as tiled or overlapped windows."*

And furthermore, for a better usability of Statechart modeling tools they recommend:

> *"New visual languages and development tools can simplify this task significantly by making sure that the developer can easily browse and navigate through all the information required to understand a subsystem."*

In the next section we propose a simulation technique, which enhances the Statechart simulation using "structure-based" abstraction.

4.1 THE DYNAMIC STATECHART NORMAL FORM

Semantic focus-and-context simulation for Statecharts

This section deals with an alternative paradigm for visualizing Statecharts during simulation. The basic idea is to dynamically construct a view of the system model that includes all active states (the *focus*) and their parent states (the *context*); all other states are hidden. This constitutes a dynamic variant of the semantic focus-and-context representation [88]. Compared to the SNF introduced in Section 3.2, this is a normal form that not only considers the static structure of a Statechart, but also a specific configuration that the system is in. We call this a Dynamic Statechart Normal Form (DSNF), which leads to *Dynamic Statecharts*. This technique keeps

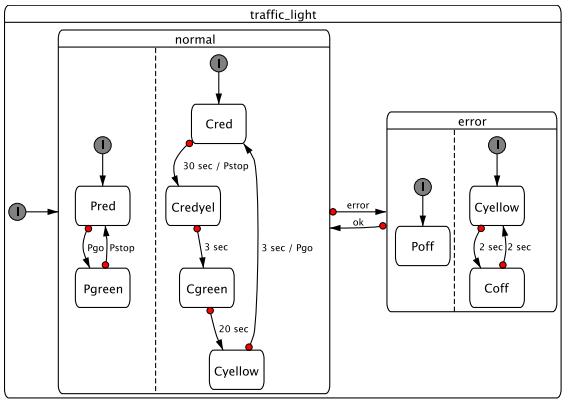
the displayed number of states small, but prevents the modeler from a comprehensive understanding of the whole SUD.

An example sequence of Dynamic Statecharts is shown in Figure 28. In the initial state, shown in Figure 28b, the system enters the configuration consisting of the active simple states *Pred* and *Cred* within the state *normal*. The active simple states, their siblings, and their parents (with their siblings) are visible; the remaining, inactive states, in this case, the sub-states of *error*, are hidden ("collapsed"). Compared to the full layout shown in Figure 28a, the presentation is more compact and focuses attention on the active states—yet the full context needed to orient the viewer is still present. Figure 28c shows the simulation after 30 simulated seconds. The component controlling the car light has progressed from state *Cred* to *Credyel*, hence the markings of states have changed; but one sees the same set of states as in the initial step. We say that the *configuration* has changed, but the *view* remained the same. Figure 28d shows the simulation after the signal *error* has occurred. Now the view has changed; the state *normal* has collapsed, and the sub-states of *error* have become visible.

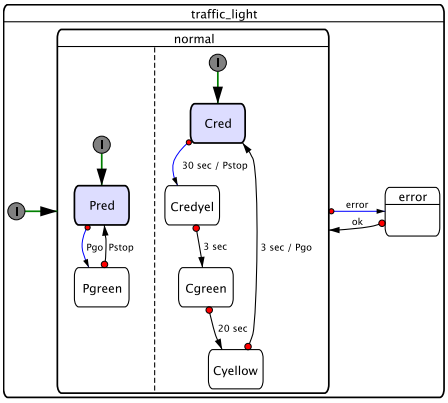
4.2 SIMULATING COMPLEX STATECHARTS WITH DSNF

The application that served as our case study is a window wiper controller (see Figure 29). It was provided with courtesy of DaimlerChrysler AG, Research REI/SM, and is an example without any reference to a production design. The model describes a complete wiping system and allows (rain and velocity) sensor-triggered wiping, interval wiping, wiping interruption due to engine starter activity, *etc.* It consists of 36 states and pseudo states (nine of them are hierarchical *OR* states, and one is an *AND* state) and 82 transitions. The chart is distributed to eleven sub-charts. The left-hand side provides the top level view of a Statechart. It shows two simple states, and the hierarchical state *Wischer_betriebsbereit* ("wiper operational"). To show its inner states, the modeler has to double-click with the mouse on the state's border, and the view on the inner states (right-hand side) replaces the top-level view.

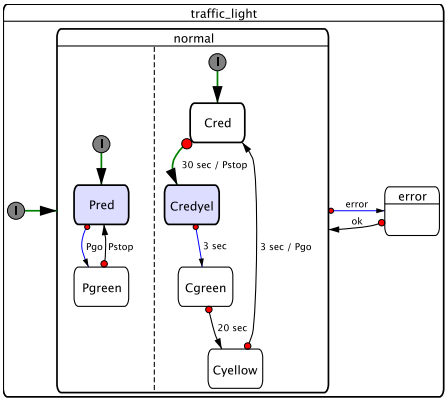
An example sequence of Dynamic Statecharts is shown in Figure 30. Here, we take the aforementioned example from DaimlerChrysler, omitting transition labels. In Figure 29, the whole wiper system conforming to the SNF is depicted. One can see all



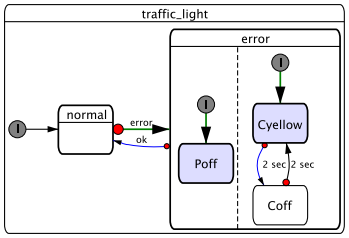
(a) Statechart laid out conforming to the SNF



(b) Entering the initial state—all lights red



(c) Cars get a red/yellow light



(d) Entering the *error* state

Figure 28: Simulating the traffic light Statechart in DSNF

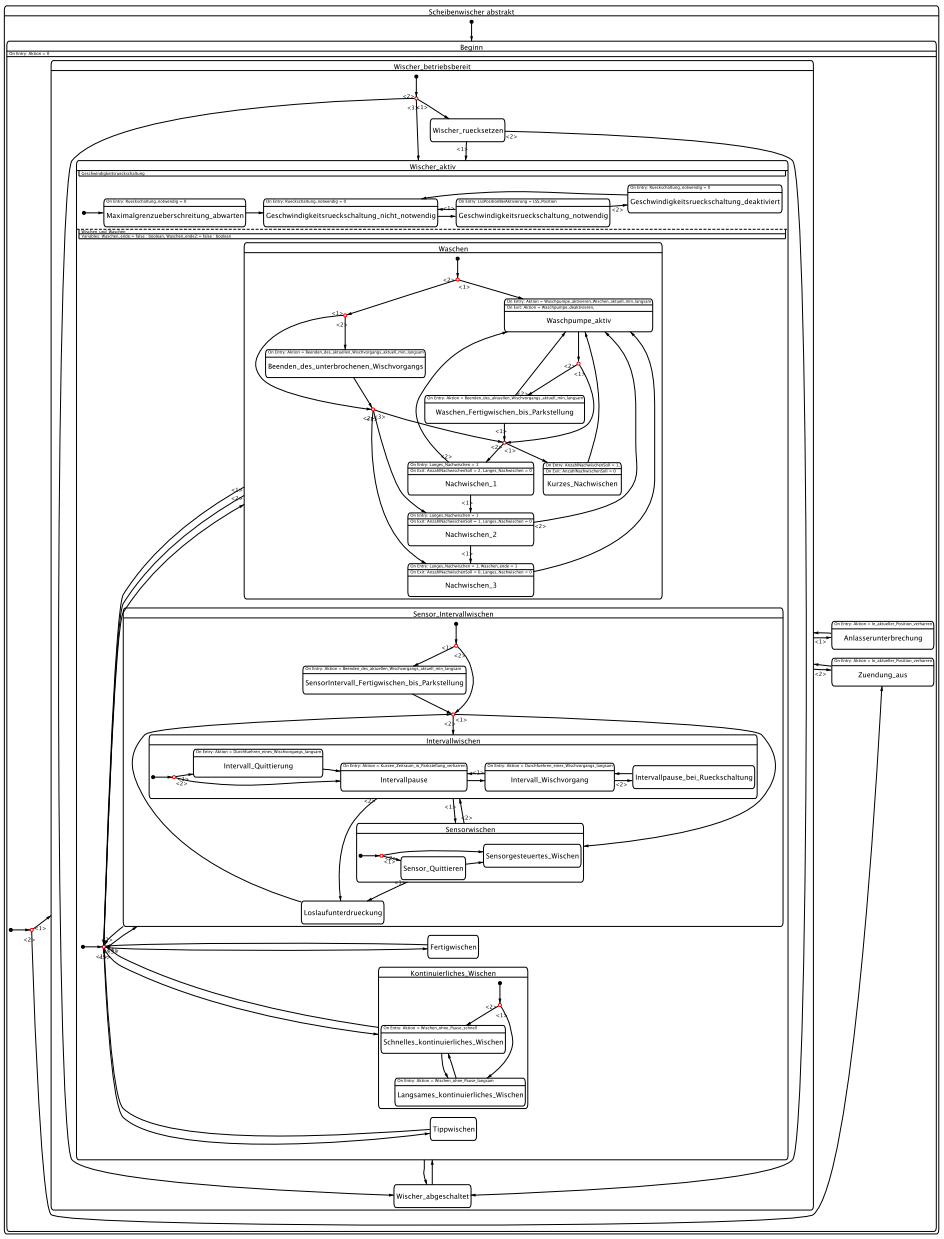
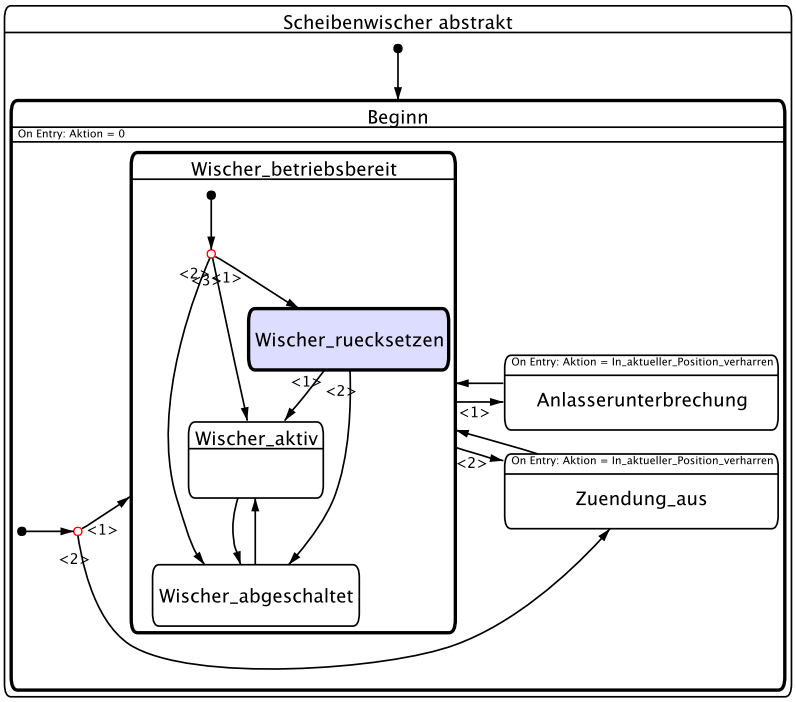
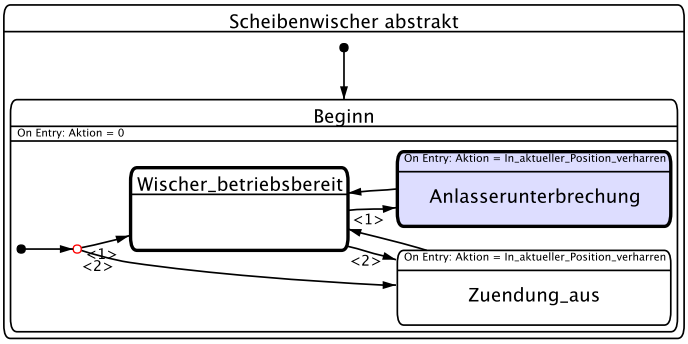


Figure 29: Layout of the whole window wiper Statechart according to the SNF



(a) Simulating of the window window: entering the wiper activation mode



(b) Simulating of the window window: interrupting the wiper mode by starting the motor

Figure 30: Simulating the window wiper Statechart in DSNF

states with their properties. In the first activated state, shown in Figure 30a ("reset wiper"), the system enters the configuration consisting of the active simple state *Wischer_ruecksetzen* ("reset wiper") within the state *Wischer_betriebsbereit* ("wiper operational"). The active simple states, their siblings, and their parents (with their respective siblings) are visible; the remaining, inactive states, in this case the sub-states of *Wischer_aktiv* ("wiper active"), are hidden ("collapsed"). Compared to the full layout shown in Figure 29, the presentation is more compact and focuses attention on the active states—yet the full context for orientation is still present. Figure 30b shows the simulation entering the state *Anlasserunterbrechung* ("stater interruption"); here the wiper activity is interrupted engaging the motor starter. Now the view has changed; the state *Wischer_betriebsbereit* ("wiper operational") has collapsed.

The maintenance of the modeler's mental map

The example simulation heavily changes view point, size, position and formation of Statechart objects. This may pose cognitive difficulties to the user. Hence, during a simulation using Dynamic Statecharts, it is essential to maintain the mental map that the user builds up for a Statechart. This can be done, for example, by *animating* one view into the next view, instead of abrupt view changes—thus providing visual clues to the user about how the view is changing during a simulation step as described by Diehl et al. [31].

The experimental results in Section 7.2.1 show, that the concept of Dynamic Statecharts significantly reduces the size of a displayed Statechart during simulation. We also show, that KIEL is very fast, if it uses Dynamic Statecharts to display simulation results.

5 PREVENTING ERRORS IN GRAPHICAL MODELS

The assurance of quality is one of the most essential aspects in the development of reactive systems. System quality thus becomes an important part of the certification process required for embedded safety-critical systems, since the failure of such systems—attributable to programming flaws—can often cause loss of property or even human life. Parnas [120] stated that human code reviews are time-consuming and highly undependable in revealing errors. Taking part of the burden off the reviewers, as well as off the designers, and letting a computer perform preliminary checks, is the rationale for *automated error prevention*.

Graphical languages are easy to model syntactically correct, but bad for avoiding modeling errors.

This chapter addresses how to ensure certain aspects of safety in developing Statecharts. We achieve this by applying methods of automated error-source detection. Therefore, we propose a set of rules that forms a fundamental Statechart style guide. Based on the well-structured set of robustness rules, both syntactic and semantic, an automated checking framework has been implemented as a plug-in for the KIEL Statechart modeling tool.

Great care was taken in devising the set of rules and in designing the checking framework implemented in KIEL (see Section 6.7) as not to constrict the modelers' creativity, but to cater for more explicit, easy to comprehend, and less error-prone models. Our approach therefore was developed adhering to the following requirements:

Requirements for robustness rules

MODULARITY AND CONFIGURABILITY: Inspired by the notion of *flexibility* and *adaptability* [24], all robustness checks are independently implemented, individually selectable, and parametrizable via preference management.

EXTENDABILITY OF THE RULE SET: The set of checks is easily extendable by either adding an appropriate Object Constraint Language (OCL) [166] constraint or, if required, by implementing a new Java class.

AUTOMATIC CONFORMANCE CHECKING: The compliance with the robustness rules can be checked very rapidly—a key qual-

ity, imperative for end-user acceptance. Due to the uncoupling of the checking process from the modeling process, the checks may be applied at all stages of system development, even to partial system models.

Even though a wide range of applications for Statechart verification already exist, none fulfills all of our needs. They are either highly specialized and therefore not extendable or they are extendable but do not provide the possibility to check complex problems. These problems were taken into consideration for our approach we present here.

5.1 Statechart modeling errors

To support the early detection and elimination of modeling errors, a design methodology must provide effective communication among the various design stages of the product. This section gives an overview of common error sources in developing Statecharts; the next section describes how these may be avoided. Especially when dealing with complex charts the automated error avoidance is of great importance

Categories of errors in modeling Statecharts

Errors in development of graphical models like Statecharts have a large diversity of types and reasons. Errors in developing Statecharts can basically be distinguished by two categories:

1. *Errors arising from the modeling tool environment:* A paramount reason for producing erroneous Statecharts is apparently the misunderstanding of utilized modeling tools and their simulation behavior. This may be caused by *counterintuitive specification* of the model semantics (*e. g.*, unbound behavior) and missing comprehension of the modeler. Scaife et al. [143] and Shi et al. [148] address this problem by developing a safe subset of Statecharts. The intention is to confirm the applicability of Statecharts in the development of safety-critical systems. Other errors in models originate from discrepancies in system requirements, such as designer oversight or specification ambiguity. Besides, design intents can be misinterpreted.

2. *Errors arising from the developed model:* Errors also originate *e. g.*, from the often *sheer size of graphical models*: Because of the extensive requirements in software design technology,

the dimension of graphical model increases significantly. If a Statechart model increases, this also emerges a large *complexity*: Because of the discrete nature of Statecharts, small changes do not always have small effects. In some cases, it is impossible for human modelers to trace such effects. The results are errors resulting in wrong functionality of specified behavior. Furthermore, some problems can be inherently undecidable or computationally intractable, requiring a possible exponential blow-up in running time or memory. In some cases, no amount of testing could prove that a system is correct.

Moreover, Statecharts are *interactive and distributed systems*: Large collections of interconnected components usually involve interactive and concurrent processes. Therefore, the potential errors can be very subtle and hard to locate for human modelers.

How to solve detected problems

Errors of the first category can often be eliminated by synchronizing the requirements and the SUD. These problems can generally only be solved by human reviewers. This process is often tedious and cannot detect all appearing errors exhaustively. The second category addresses problems which concern implicitly appearing errors. The modeling of realistic applications results often in large and unmanageable graphics. Therefore *e. g.*, we provide a method for easy development and understanding of complex Statecharts (see Chapter 4). Besides, errors resulting from complexity are predestined to be revealed by automated detection due to inconsistencies between specified modelers intent and the resulting simulation behavior.

5.2 ERROR PREVENTION IN MODELING STATECHARTS

The approach to error prevention in textual and visual languages faces essentially the same problems. Due to this, we propose a common error prevention nomenclature and at first consider textual programming languages.

Kinds of error prevention

Software error prevention in general encompasses a number of different techniques designed to identify programming flaws. As outlined in Figure 31, we can basically distinguish between automated error prevention and human code review. As already

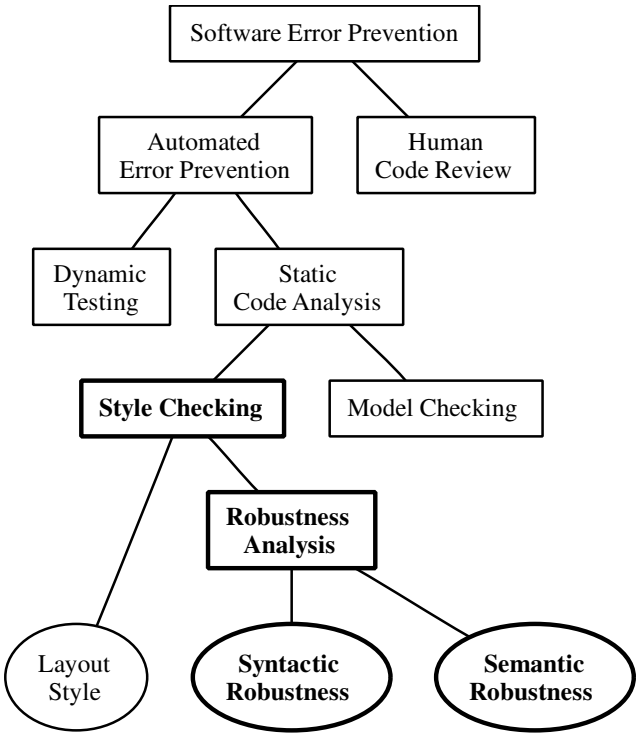


Figure 31: Software error prevention and its taxonomy (source: Schaefer [144])

pointed out, human code reviews are exceedingly time-consuming and often undependable in revealing errors. However, they may sometimes find conceptual problems which cannot be detected automatically.

Automated error prevention is commonly separated into dynamic and static methods. *Dynamic testing* performs code evaluation while executing the program and attempts to detect deviations from expected behavior. *Static code analysis*, on the other hand, performs an analysis of computer software without actual execution of programs, but by assessing source or binary files to identify potential defects. Moreover, while dynamic testing requires executable code, static methods can be applied much earlier in the development process. Static code analysis covers examinations ranging from the behavior of individual statements and declarations to the complete source code of a program. Use of the information obtained from the analysis varies from highlighting possible coding errors to formal methods that mathematically prove properties

about a given program, *e. g.*, that its behavior matches that of its specification, commonly known as *model checking*.

Style checking, another aspect of static code analysis, is concerned with layout style, *i. e.*, common appearance, as well as syntactic and semantic style. The latter two are often collectively referred to as robustness analysis (see below). Style checking always requires the syntactic and semantic correctness of the code. If there are less sources for errors, there should be less errors. Roughly 80% of the lifetime cost of a piece of software goes to maintenance [151], and hardly any software is maintained for its whole life by the original author. Therefore, style checking is especially important in large software projects. *Robustness analysis*, as an important field of style checking, refers to "the objective of eliminating certain types of errors and enforcing sound engineering practices" [69, page 17]. The use of the word robustness must be clearly distinguished from other connotations, *e. g.*, robust control or robust design.

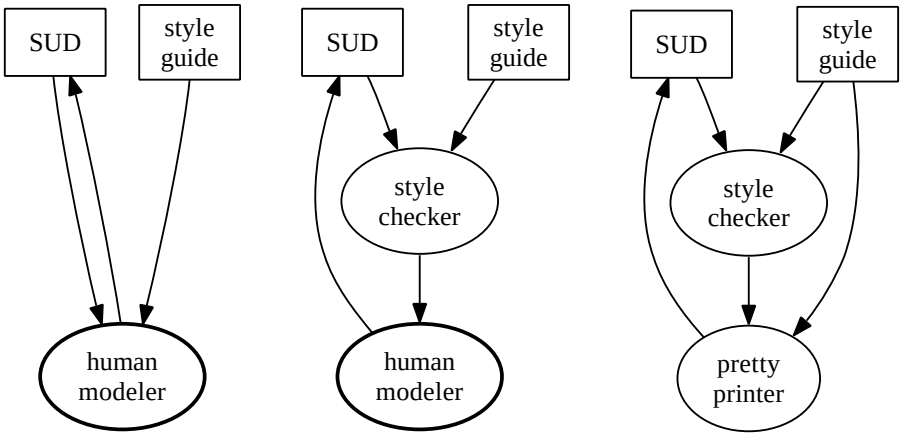
Robustness rules limit the general range of a given modeling respectively programming language, as they are entirely independent of what is being designed. Robustness may either describe robust data structures or robustness checks of implementations, with the aim of retaining the proper functioning of programs in the presence of hardware malfunctions and failures. By these means, a *subset* of the programming language is specified which—though less comprehensive and flexible—is considered to be also less error-prone by MISRA [107] and by Hanxleden et al. [70]. Hence, robustness analysis is a preventive instrument and substantially draws from the knowledge and understanding of software implementation flaws [41].

Robustness rules define a safe subset of a language.

Style checking as well as human reviews, are based on *style guides*. Here, style guides constitute a set of design rules, concerning the textual programming, respectively the modeling of Statecharts. Layout style and robustness analysis are clearly distinguished within the realm of this work, as will be pointed out in the sections that follow.

5.3 STYLE GUIDES FOR ERROR PREVENTION

Style guides provide general instructions on how to use languages. They are commonly provided as (in-)formal specifications, containing lists of rules. Style guides concern



(a) Process of human reviewing an SUD

(b) Process of automated style checking an SUD

(c) Process of automated styling an SUD

Figure 32: The role of style guides in making an SUD style conform

1. human languages (*e. g.*, the *The Chicago Manual of Style* [117]),
2. textual programming languages (*e. g.*, the MISRA guide [107]), as well as
3. visual programming languages, such as Statecharts (*e. g.*, *The Elements of UML 2.0 Style* [1]).

They define a subset of usable elements for (visual) languages. Regarding textual and visual programming languages, style guides consider associations, aggregations, and compositions of elements.

Application areas of style guides

The informal as well as the formal specifications are primarily These affect the programmed or modeled result. Figure 32a depicts a process where the human modeler develops an SUD taking the style guide into account.

Formal style guides act as the configuration for *automated style checking*, *i. e.*, style checkers. Such tools are commonly employed as the number of rules in a style guide is usually too large for human developers to remember. Moreover, some context-sensitive rules demand inspection of several files at once and are thus very hard to check for humans. The support of a style checker is depicted in Figure 32b. Although style checkers can reduce formal code review time considerably, it certainly does not mean that human code reviews are obsolete.

Apart from operational instructions for humans and automated style checking, rules of style guides can be *applied automatically* to an SUD. Such tools are known as *pretty printer* for programming languages. In this sense, an automatic layouter for graphical diagrams can be seen as a pretty printer. Figure 32c depicts the process of pretty printing of an SUD according to the rules of a style guide. This technique can relieve the modeler from laborious work of making the SUD style guide conform. We propose such an approach *e. g.*, for the automated layout of a Statechart (cf. Section 3.2). However, automatic styling is exclusively applicable to the Secondary Notation (the style) of an SUD. An automated semantic styling seems to be impossible; therefore, the SUD's intention must be known. Here, automated style *checking* would be the best choice.

5.3.1 *Taxonomy for Style Checking in Statecharts*

In the general context of static code analysis, one must separate syntactic and semantic correctness on the one hand and style checking on the other hand. As a first step towards systematically devising an extensive style guide for Statecharts, the following taxonomy, also depicted in Figure 33, was laid down based on this:

SYNTACTIC ANALYSIS: In general, the enforcement of syntax-related rules does not necessitate knowledge of model semantics.

- *Readability* (or layout style) aims at a graphical normal form, *e. g.*, transitions connect states in a clockwise direction, charts contain a limited number of states, *etc.*
- *Efficiency* (or compactness, simplicity) eliminates superfluous and redundant elements from the Statechart model.
- *Syntactic Robustness* aims at reducing errors due to inadvertence and enhancing maintainability.

SEMANTIC ROBUSTNESS: Deriving and enforcing semantic robustness rules requires knowledge of specific aspects of the model semantics. Exact analysis typically requires the use of verification tools, *e. g.*, reachability, disjunction of transition predicates, shadowing of (less-prioritized) transitions.

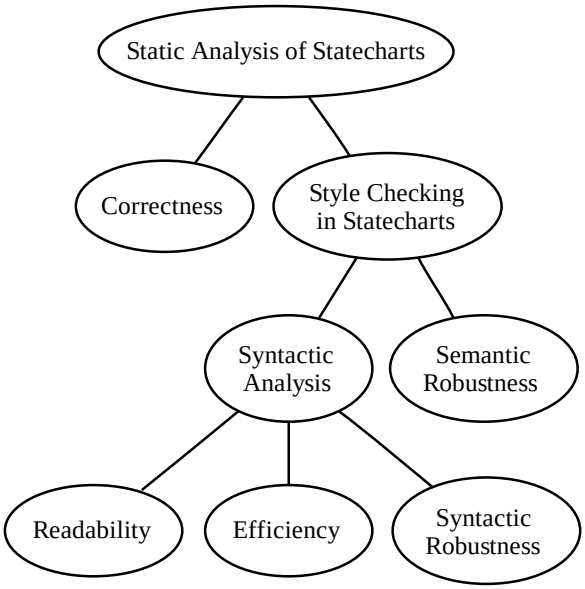


Figure 33: Taxonomy for style checking in Statecharts (source: Schaefer [144])

As a whole, this taxonomy is suggested as a productive groundwork for systematically devising an extensive style guide for Statecharts. In the following, the discussion focuses mainly on Statechart robustness analysis, syntactic as well as semantic.

5.3.2 *Existing Style Guides and Applications*

Style guides and applications for textual programming languages

Since programming style often depends on the programming language, different coding standards and related code checking tools exist for different programming languages [32, 107, 149, 151]. Akin to coding standards, most code checking tools are programming-language specific. Available code checkers for C are *Lint* [80], *LCLint* (aka. *Splint*) [41], and *QA Motor Industry Software Reliability Association (MISRA)* [127]; code checkers for Java are *Jlint* [7] and *Checkstyle* [25]. Figure 34 roughly classifies these code checkers according to their emphasis on layout style *vs.* robustness—a major distinction within style checking (see Section 5.2).

Statechart style checking is much less developed and less sophisticated as compared to style checking in textual computer programming. Nevertheless, when analyzing the dynamics of (safety-critical) reactive systems, it is all the more important that models are designed according to approved rules. Furthermore,

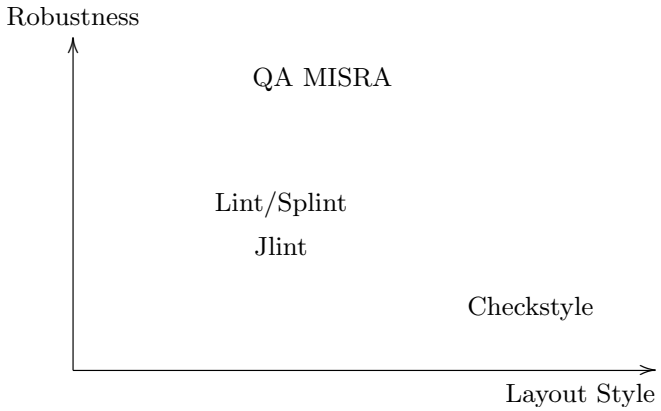


Figure 34: Classification of checking tools for textual programming languages to their emphasis of layout style *vs.* robustness (source: Schaefer [144])

homogeneous modeling and layout of Statecharts within projects, checking rules of a style guide ensure that best practices, tool specific optimizations, and domain or company specific conventions are observed. Checking these rules automatically, the developer can be released from identifying oversights and simple syntactic or semantic errors and, additionally, the model transfer between various Computer-Aided Software Engineering (CASE) tools can be simplified.

Although the commercial Statechart modeling tools Esterel Studio, Stateflow, and Statemate have been supplemented with a number of consistency checks, *e. g.*, Mutz and Huhn [110], and Kossowan [87] expose the following main deficiencies in style checking capabilities of Statechart modeling tools:

Deficiencies in style checking

1. Analyses are tool-specific.

2. Analyses are inalterable, *i. e.*, checks are neither extendable, individually selectable, nor capable of parametrization. Thus, the user has to adhere to the set of checks provided by in the tool.

3. The scope of the rules checked by tools varies from rudimentary analysis to advanced model checking.

4. In Esterel Studio, *e. g.*, checking cannot be invoked explicitly, but is part of the simulation or code generation. Moreover, certain checks cover merely those parts of the Statechart that are

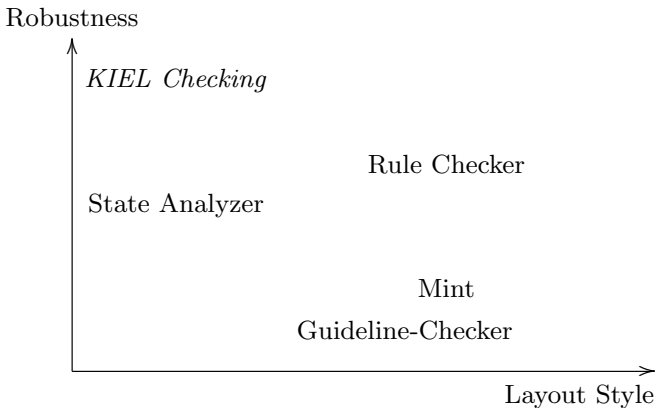


Figure 35: Classification of Statechart checking tools to their emphasis of layout style *vs.* robustness (source: Schaefer [144])

actually reached by the control flow of the current simulation (non-static analysis).

Checking applications for Statecharts

Four representative checking tools—the *Mint* and the *Guideline-Checker* related to Stateflow, the *State Analyser* related to Statemate [145], and the *Rule Checker* [110]—as well as our style checker (see Section 6.7), are roughly classified according to their emphasis on layout style *vs.* robustness in Figure 35.

The two tools developed at DaimlerChrysler as well as Ricardo's *Mint* all address only a single Statechart dialect. *Mint*, the *Guideline-Checker*, and the *Rule Checker* merely perform graphical and—partly trivial—syntactic checks. The do not profound semantic checks, which require automated theorem proving as realized in the State Analyzer. However, semantic checks are particularly important since they eliminate possible non-trivial sources of error, which are very hard to discern for humans. Thus, the checks, presented in the next section, aspire to fill the gap. The positioning of the KIEL *Checking* plug-in within Figure 35 further emphasizes this intention.

5.4 A STYLE GUIDE FOR MODELING STATECHARTS

Based on the aforementioned foundation, practical experience, and available prototypes, this work sets out to define a comprehensive Statechart style guide, striving for general applicability to Statechart dialects within the limits of the Unified Modeling Language (UML)

State Machines specification. The rules presented below were formulated following the advice of Buck and Rau [24].

As mentioned above (cf. Section 5.3.1), style guides for Statecharts can roughly be divided into two parts, namely syntactical analysis on the one hand and semantical analysis on the other hand. Syntactical analysis addresses the syntactical structure of Statecharts, such as layout, possible optimizations, and robustness problems. Therefore, when it comes to formulating syntactical rules, one basically has to focus on problems that deal with the relations of individual Statechart elements to each other. Furthermore, syntactical analysis opens up two fields of possible applications. One field analyzes whether the syntactical relation of the elements used corresponds to the rules specified by a certain dialect (*i. e.*, syntactical correctness). Within the UML these kinds of rules are called well-formedness rules. The well-formedness rules "[...] specify constraints over attributes and associations defined within the Statechart meta model" [113, Section 2.3.2.2]. The other field of possible applications is formulating rules dealing with the syntactical robustness of Statecharts. Taking into account that robustness rules principally deal with more sophisticated problems one can say that syntactical correctness is essential for being robust.

Syntactical and semantical robustness rules

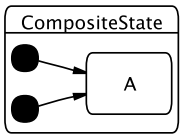
Concerning the location of a problem, the set of rules can be divided into

1. rules locating problems based on transitions and their labels.
2. rules dealing with problems based on states.

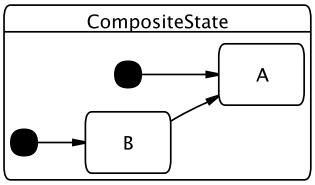
Locating problems resulting from the part of both—syntactical correctness and syntactical robustness (see Figure 33)—works the same way. Since Statecharts are directed graphs, one can use pattern matching here. If used for locating problems, one would create a pattern which targets the problem.

The location of problems uses pattern matching.

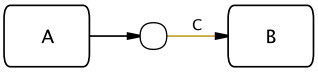
Following the taxonomy (see Figure 33) for style checking in Statecharts, the rules are grouped in different sections. In the following, we present the rules incorporated into our Statechart style guide. First of all, the rules dealing with the syntactical correctness—*the well-formedness rules*—are presented. On this basis we extend the style guide by presenting the rules for syntactical robustness afterwards. Finally, the rules for semantical robustness are presented. Bell [15] describes the syntactical rules and Schaefer [144] describes semantic rules in full detail.



(a) Violation of rule *CompositeState-1*



(b) Complicated violation of rule *CompositeState-1*



(c) Violation of rule *Transition-5*

Figure 36: Violation of well-formedness rules (source: Bell [15])

UML *Well-Formedness Rules*

The realization of well-formedness rules

As mentioned above, syntactical correctness is mandatory for robustness. Therefore, it is necessary to check whether a Statechart is syntactically correct or not. For most Statechart dialects, this is done within a dialect-dependent modeling tool. But when dealing with UML State Machines one has to make sure manually that the above mentioned well-formedness rules are preserved as some UML tools do not check those rules. Within the UML, the well-formedness rules themselves are described using OCL. Given a context of application and the constraint itself, problems are detected fairly easily. In the following, we present some examples for violations of the well-formedness rules. Section 6.7 elaborates on the OCL implementation of the elaborated examples.

The rule *CompositeState-1* denotes that "a composite can have at most one initial vertex" [113, Section 2.12.3.1]. Detecting violations of this rule as presented in Figure 36a is done by a two-part pattern. One part contains a composite state with no initial vertex and the other part contains a composite state with one initial vertex. Violations of this rule are present in a Statechart, if both parts do not match at the same time. Fixing this rule has to be done with great care because the intended behavior has to be carefully remodeled, as modeled Statecharts can include parts in which it is not clear what should be done as depicted in Figure 36b.

The rule *Transition-5* denotes that "transitions outgoing pseudostates may not have a trigger" [113, Section 2.12.3.8]. Violations of this rule are detected by a simple pattern. The pattern may just

contain a *Transition* with the type of the source set to *PseudoState* and no trigger specified. The pattern will match for each violation present.

Syntactical Robustness Rules

Besides the well-formedness rules, we collected rules from different sources and from our own experience in Statechart modeling to extend the proposed style guide. This extension was done to make the style guide feasible for daily work. As mentioned in Chapter 2 style guides already exist for different Statechart dialects, especially for Matlab Simulink/Stateflow. The style guide for Statecharts proposed in our work, in contrast, aims at covering a wide range of dialects. Therefore, we extracted rules from various other style guides that are applicable to different dialects. This goal is not achievable for all rules, as some dialects differ in their syntax.

The rules portrayed below were first presented by Mutz [109]. All of them apply to the area of syntactical robustness rules and are collected within the set of dialect-independent rules.

MIRACLESTATES: All states except the root state and the initial states must have at least one incoming transition.

ISOLATEDSTATES: An even stronger version of *MiracleStates* is the check for isolated states. A state is isolated, when it has neither incoming nor outgoing transitions.

EQUALNAMES: Ensuring that all states are named differently simplifies the maintenance of a Statechart.

INITIALSTATE: Demanding that all regions respectively non-concurrent composite states contain one initial state greatly simplifies the understanding of the model: This rule should also be checked on dialects in which a region or non-concurrent composite state can be entered by an interlevel transition.

ORSTATECOUNT: Checking if all non-concurrent composite states contain more than one state, delivers valuable hints for possible optimizations. Composite states that contain only one state, can be subject to dialect independent optimizations and should be avoided from the beginning.

REGIONSTATECOUNT: Closely related to *OrStateCount*, this rule checks the number of states within a region of a concurrent composite state. Such regions can also be optimized and should be avoided for simplicity.

From the Ford style guide [45], the following rule was extracted as it is also applicable to dialects other than Stateflow.

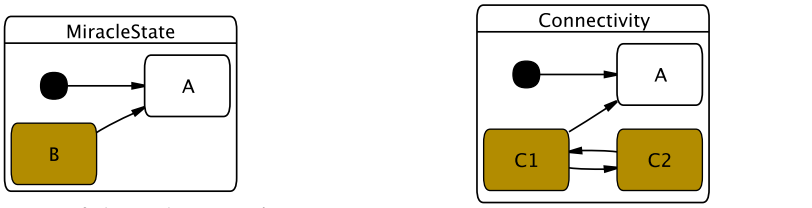
DEFAULTFROMJUNCTION: When using connective junctions to model decisions within a Statechart, one should always add an outgoing transition with no label. The unlabeled transition is then the default transition. The default transition is provided so that the control flow does not stop when the other conditions do not hold.

From our own experience in modeling with Statecharts the following rules were formulated.

TRANSITIONLABELS: Ensuring that all transitions are specified with a label makes the understanding of the model easier. When all transitions are labeled appropriately, the intended meaning gets clear for readers. This is especially important for dialects in which a default signal exists as it would be assigned invisibly to an unlabeled transition.

INTERLEVELTRANSITIONS: Ensuring that a Statechart does not contain interlevel transitions *i. e.*, transitions bypassing level borders, has two benefits. The first benefit is that understanding a Statechart without interlevel transitions is easier as the execution semantics is not always clearly defined. The second benefit is that a model gets portable between various tools. Even porting from tools supporting interlevel transitions to tools not supporting interlevel-transition gets possible.

CONNECTIVITY: Other aspects closely related to *MiracleStates* are states not connected to any initial state. This is checked with the test for connectivity. States being not connected to any initial state are superfluous, as the intended behavior will never be executed. See Figure 37b, where the states *C1* and *C2* are not connected to the initial state. This rule extends the already mentioned *MiracleStates*, as it also detects states that have incoming transitions and are still never entered as depicted in Figure 37b.



(a) Violation of the rule *MiracleStates*

(b) Violation of the rule *Connectivity*

Figure 37: Violation of syntactical robustness rules (source: Bell [15])

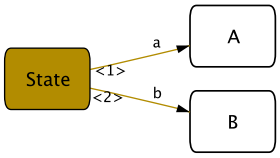
The resolution of detected syntactic problems

As mentioned above, locating a problem is fairly easy. In contrast, resolving a syntactic problem can be more difficult. Depending on the context in which the problem is found and the problem itself, a different approach has to be used for each problem. Basically, one can say that there is no pattern that works for all problems. Resolving found problems has two benefits. One benefit is that the syntactical correctness of a Statechart will be achieved. This applies especially to the well-formedness rules of the UML. The other important benefit is that the maintainability and the readability will increase enormously.

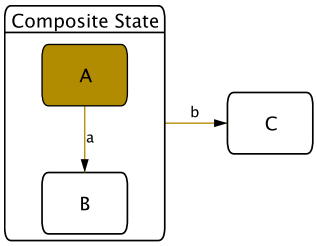
Semantic Robustness Rules

In line with the taxonomy presented in Figure 33, we now turn to semantic robustness rules, addressing the model's behavior. As opposed to model checking, however, semantic robustness analysis is concerned with the behavior of individual statements and their interaction at a local level, *e. g.*, determinism and race-conditions. The three rules presented below are all Statechart dialect-independent. As transitions are considered pairwise, let $trans_1$ and $trans_2$ be the two transitions to be investigated. The label of $trans_i$ is l_i, which consists of the predicates e_i (event expression) and c_i (condition expressions) as well as an action expression a_i, where $i \in \{1, 2\}$.

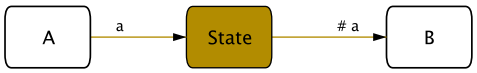
TRANSITION OVERLAP All transitions (directly or indirectly) outgoing from a state should have disjoint predicates [87]. Ensuring this, warrants that at most one transition is enabled at any time. *I. e.*, no transition shadowing can occur, leading to guaranteed deterministic behavior independent of potential transition priorities. Figure 38a depicts a basic case of two departing transitions from a simple state. A *Transition Overlap*



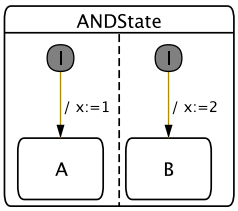
(a) State with overlapping transitions



(b) "Indirectly" overlapping transitions



(c) *Dwelling* violation



(d) Write/write race condition

Figure 38: Application examples of the semantic robustness rules (source: Schaefer [144])

violation exists if e_1 and c_1 are not disjoint from e_2 and c_2. A *Transition Overlap* violation may be eliminated by, *e. g.*, adding $\neg e_2$ and $\neg c_2$ to the predicates of $trans_1$, yielding $(e_1 \wedge \neg e_2)$ for the event expression and $(c_1 \wedge \neg c_2)$ for the condition expression. For Statechart dialects with prioritized transitions, the enforcement of this rule proves the modelers intend. In contrast to Statechart which do not provide transition priorities, this rule can find general non-determinist execution problems. In addition to transitions departing directly from a state, transitions departing from an enclosing state may also be enabled (see Figure 38b).

DWELLING The predicates of all incoming and outgoing transitions of a state should be pairwise disjoint, or at least not completely overlapping [87]. This rule ensures that the system pauses at every state it reaches. A state in which the system cannot pause is not in accordance with the concept of a *system state*. Careless use of Esterel Studio's *immediate* flag, denoted by #, may lead to a *Dwelling* violation (see Figure 38c for an example). An immediate transition is evaluated in the same

instant, in which its source state is reached; a non-immediate transition is not evaluated until the following instant.

RACE CONDITIONS Concurrent writing or concurrent reading and writing of a variable should not exist in parallel states (cf. Figure 38d). Since race conditions are generally not detectable, we have chosen a conservative approximation. We detect a race condition in concurrent threads, if a variable is written in one thread and read in another.

5.5 ASSESSMENT

In complex systems it can be very difficult for the modeler to keep all interdependent Statechart elements in mind. As a consequence the modeler tends to make errors. To avoid this, we have outlined an approach to make model driven system development with complex Statecharts less error prone.

Assessment of well-formedness rules

However, The well-formedness rules do not necessarily improve the quality of Statecharts in the sense of robustness. These rules apply to the field of syntactical correctness and therefore they do not point out any sophisticated problems. But nevertheless, these rules are needed before any further checking can be applied to a Statechart because the robustness checks rely on the correct syntax. Conforming to the well-formedness rules has to be checked explicitly due to the fact that not all tools prevent the user from misplacing objects in State Machines, especially when it comes to UML State Machines.

Assessment of syntactical robustness rules

Syntactical robustness rules focus on intricate problems, but are not as sophisticated as the rules dealing with the semantical robustness. Nevertheless, the information gained by applying the checks can be valuable. The information delivers sources for possible optimizations that lead to a better understanding of the checked Statechart. *E. g.*, the readability of charts gets significantly improved if all states are labeled with different names. Furthermore, the tests for *Connectivity* and for *MiracleStates* may detect design flaws that may lead to misbehavior of the modeled system. Therefore, these problems should always be corrected to fix the model and also to increase the maintainability.

As an example, we use a Statechart which was developed as a part of an automated subway control system [131]. The application

of our rules on the Statechart presented in Figure 24a (page 66) delivered the hint that violations of the rule Dwelling are present. Figure 24a shows a possible way how the violation can be fixed. Our Statechart optimization method in Section 3.5.2 can automatically solve especially this problem for SSMs.

Assessment of semantical rules

The *Transition Overlap* and *Dwelling* rules certainly improve the structural clarity of Statecharts, as all behavior is diagrammed explicitly. Especially in a non-deterministic dialect such as Statemate, the introduction of determinism greatly eases model comprehension. The *Race Conditions* rule on the other hand, might be too restrictive in real life. If applied, though, it leads to immense structural improvements as potential race conditions in far apart regions of a Statechart are eliminated *a priori*.

Finally, a duality between semantic robustness and minimality of Statecharts is evident. *E. g.*, eliminating a *Transition Overlap* or *Dwelling* violation by adding the negation of the predicates of one transition to the predicates of the other transition, as suggested above, constitutes an infringement of the *write things once* principle of modeling. Thus, fully explicit behavior specification postulated by robustness rules, may stand in contrast to the clever exploitation of implicity.

A further comment has to made here: One has to take into account that the hints returned by our checking framework do not point out any errors. Merely, the hints shall serve as an advice to locations where a problem might occur and where a modeler can improve the readability and maintainability of a system. Therefore, it obliges the modeler to take the time to fix the remarked problems.

Both syntactic and semantic rules have been implemented as a plug-in for automated checking in the prototypical modeling framework KIEL. Thereby, the high level of possible customization is achieved by employing an OCL framework. The possibility to check complex features is given by utilizing a theorem prover. The following chapter describes KIEL and its functionality. An experimental evaluation, based on the aforementioned checking rules shows the time efficiency of our rule set.

6 THE KIEL MODELING TOOL

In Section 2.2 we have initially discussed the realization of several prominent examples of Statechart modeling frameworks. In this chapter, a description of the prototypical KIEL will be given. Beside knowledge about our layout method, details on the concrete implementation are given to maintain, modify, and extend the framework. First, we will introduce the basic architecture of the framework. A general core library realizes reusable parts of the layout method; it is described in Section 6.2. Section 6.3 explains the realization of the Dynamic Statechart Normal Form (DSNF) described in Section 4.1. In KIEL we implemented two ideas of structural Statechart editing—the KIEL macro editor and the KIT editor; they are explained in Section 6.5. The technique of synthesizing Statecharts from Esterel is discussed in Section 6.6. Finally, issues on automated rule-checking extensions to the KIEL framework will be described in Section 6.7.

6.1 THE KIEL ARCHITECTURE

The KIEL tool is a prototypical modeling environment that has been developed to explore novel editing, browsing and simulation paradigms in the design of complex reactive systems. This section gives a description of the architecture of the KIEL modeling tool; an overview of the implemented KIEL modules is presented in Figure 39. A central enabling capability of KIEL is the automatic layout of Statecharts, which computes bottom-up layouts at each hierarchy level using Graph Visualization Software (GraphViz) [57].

KIEL can import Statecharts that were created using other modeling tools. The currently supported dialects are those of Esterel Studio, Stateflow, and the UML via the XML Metadata Interchange (XMI) format, as, *e. g.*, generated by ArgoUML. KIEL also provides an editor to create Statecharts from scratch or to modify imported Statecharts. With KIEL, it is possible to use a structure-based editor (based on the proposed editing process in Section 3.3), and one can also synthesize graphical models from textual descrip-

KIEL's central software modules

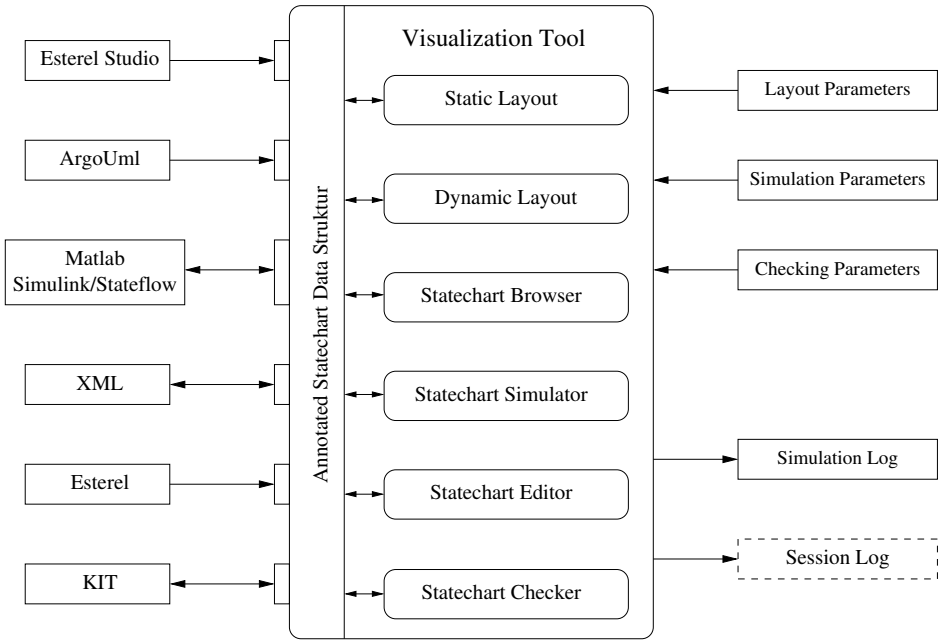


Figure 39: Module view on the KIEL tool

tions (based on the proposals in Section 3.4 and Section 3.5). KIEL provides a simulator; it simulates the behavior of imported Statecharts according to the semantics of their original modeling tool. The automatic layout is also used to present different Statechart views for each Statechart configuration. Based on these views, KIEL animates the simulation with a dynamic focus-and-context representation. In order to apply the checks presented in Section 5.4 to modeled Statecharts, a robustness checker has been integrated into KIEL. A customizable checking framework, based on OCL and, for more powerful checks, the Cooperating Validity Checker Lite (CVC Lite) [10] theorem prover, checks for compliance with the *robustness rules* proposed in Section 5.4.

KIEL is implemented in Java and is based on the Model View Controler (MVC) concept. It employs a generic concept of Statecharts, which can be adapted to specific notations and semantics. The central module of the KIEL tool is the Statecharts data structure. All modules implemented in KIEL interact with the KIEL data structure. It contains the pure component view of a Statechart and is based on the class structure of UML State Machines [113] (see Figure 40). The class *Node* is the *superclass* of the KIEL's state objects and the class *edge* is the *superclass* of *Transition* objects; the *Delim-*

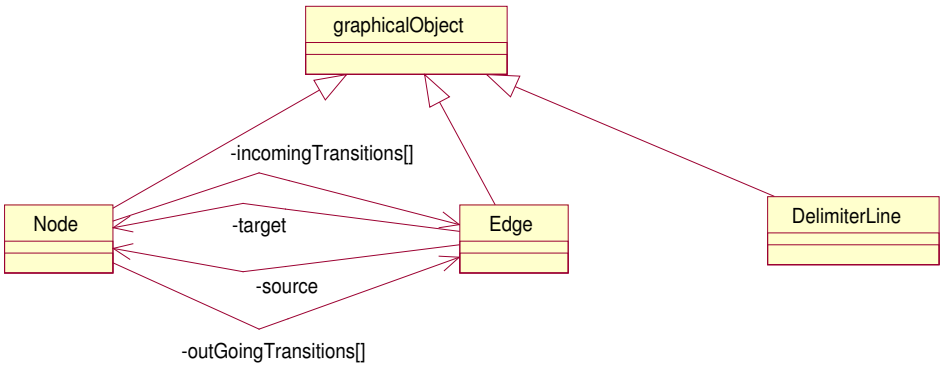


Figure 40: Simplified Class Diagram of the KIEL data structure

iterLine denotes a separation of orthogonal states into concurrent *regions*.

As the name suggests, KIEL's central capability is to automatically layout Statecharts. This is needed for both, formatting Statecharts according to an SNF (see Section 3.2) and also for Dynamic Statecharts (see Section 4.1). The realization of our layout method with KIEL is described in the following section.

6.2 AUTOMATED LAYOUT IN KIEL

A design goal for KIEL was to provide an efficient automatic layout of Statecharts, as this is not only needed for formatting Statecharts according to an SNF, but also for Dynamic Statecharts. As discussed further in Chapter 2, numerous tools and packages for automated graph layout, and even approaches specific for the layout of Statecharts exist. However, after a survey of these we concluded that none of them were really sufficient for our purposes. We finally chose to adopt the layout framework GraphViz [57] which is a collection of tools implementing several graph layout algorithms, *e. g.*, *dot* (for directed graphs) and *neato* (for undirected graphs). The limitation of GraphViz is that it does not solve the problem of hierarchical layout (their *cluster mechanism* does not suffice for laying out hierarchical states of Statecharts). However, it provides decent results for flat charts that closely match the SNF we defined, and it is extremely efficient. Efficiency was a high priority, as even for large models, the on-the-fly layout of Dynamic Statecharts should not noticeably slow down simulations. GraphViz provides highly effective and thereby appealing results of graph layout computation.

KIEL uses the program dot for Statechart layout.

Particularly, the *dot* layout generates an excellent result, adopting the idea of an SNF. The realization rests upon numerous aesthetic criteria [30] and relies on heuristics [58].

Handling of state hierarchy

To layout nested states, we have developed a hierarchical layout engine that computes Statechart layouts bottom-up in the state hierarchy. After computing the layout of one layer, the resulting graph is used to compute the size of the enclosing superstate. Furthermore, the layout direction alternates between layers to minimize overall space requirements. The hierarchical layout engine can employ any layout mechanism that can layout flat graphs, such as in our case GraphViz.

Another Statechart layout algorithm

Using the KIEL tool, the user can choose between several layout features. Here all layout algorithms offered by GraphViz and an own Statechart layout algorithm (a simple horizontal-vertical placement approach) can be enabled. The user is free to switch back to the original layout assigned by the originating modeling tool. The layout results of some the diagrams automatically drawn by KIEL are presented in Appendix A.

One can use KIEL simply to perform a layout of a given Statechart; however, the functionality of KIEL goes significantly beyond that. The tool's main goal is to enhance the intuitive comprehension of the behavior of the SUD. While most available Statechart development tools merely offer a static view of the SUD during simulation, KIEL provides a dynamic visualization (cf. Section 4.1).

6.3 SIMULATING STATECHARTS IN KIEL

In addition to the usual animation of static model views, with highlighting active states and transitions, KIEL also provides the Dynamic Statechart mechanism outlined in Chapter 4. For each configuration, a *corresponding view* is computed that conforms to the SNF, using the automated layout mechanism also employed for static views. If a configuration is entered for which the corresponding view has already been computed, a cached view is used instead. Following view changes are facilitated by morphing, with adjustable speed.

Computation of Statechart views

The structure of KIEL's view concept is depicted in Figure 41. In KIEL, a view contains the graphical information of all Statechart objects. As every view corresponds to the corresponding Statechart configuration, all possible configurations represent the view set of

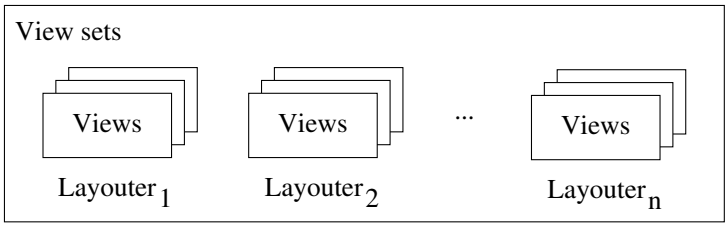


Figure 41: The view concept in KIEL

one layouter (as *e. g.*, the *dot* layout module); all view sets represent all graphical Statechart information. This very flexible approach allows us to manage and access a special view easily and quickly. Thus, the view computation in KIEL can be done

A PRIORI: *i. e.*, the views are computed at the start of KIEL (as a result, KIEL performs well during simulation) or

ON DEMAND: *i. e.*, the views are computed if a view change is triggered or on user request (as a result, KIEL performs well at startup).

Representation of a simulation step

For simulating SSMs, KIEL not only provides the macro step computation as offered by Esterel Studio, but it also allows to compute and visualize micro steps. Such micro steps are *e. g.*, testing a transition condition, executing a transition action, *etc.* The user can decide to simulate the Statechart using micro steps or macro steps. In both cases micro steps are graphically and textually visualized. This facilitates the understanding of intricate charts as the example shown in Figure 42. Furthermore, macro steps can be executed forwards and backwards. In the backward simulation, the last simulation state is assigned and a new signal injection is expected.

KIEL can simulate Statecharts, according to the semantics of SSMs used in Esterel Studio (using a KIEL-internal simulator) or alternatively, according to the semantics of Stateflow, using the Stateflow Application Programming Interface (API).

6.4 KIEL AND STATEFLOW

Matlab Simulink/Stateflow is a commercial modeling tool that is routinely used by control engineers to design and simulate discrete controllers. Simulink provides the ability to model hybrid systems (mixed continuous and discrete dynamics) in a block-

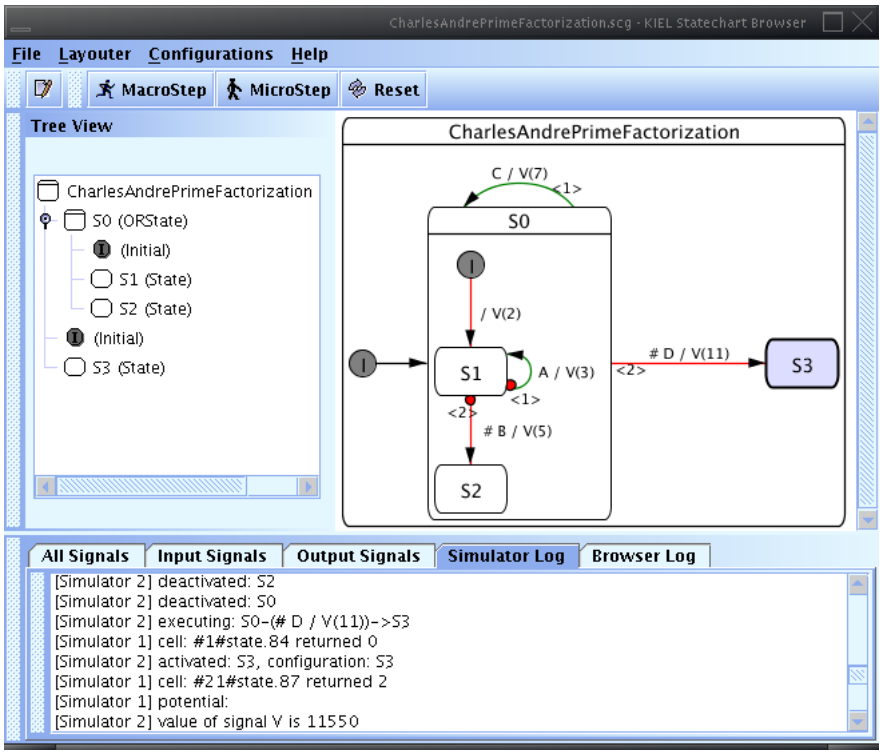


Figure 42: A screen shot of KIEL as it simulates a Statechart example suggested by André [3]

diagrammatic notation. Stateflow provides the ability to model hierarchical parallel Statecharts. A Stateflow model can be inserted in a Simulink model as a block and the Simulink blocks can provide input stimuli and receive outputs from the Stateflow block. The Stateflow model can be simulated within the Matlab framework for a desired time period, which in effect steps through the State Machine for the given input excitation. The responses from the State Machine can be plotted graphically and the trajectory of the State Machine can be observed visually, as well as recorded for post analysis. Additional analyses are possible in terms of the time spent in different states, the latency from the time of a change in excitation, to the time of change in outputs in the State Machine, stability of the system, etc. Thus, the Matlab Simulink/Stateflow tool suite provides a convenient and intuitive framework for observing and verifying the behavior of the system.

We have implemented an engine, using KIEL, that

1. translates the KIEL representation of Statecharts into a Simulink/Stateflow model,

2. translates the Stateflow model into KIEL representation of Statecharts, and

3. controls the non-interactive simulation of Simulink/Stateflow models using KIEL.

Matlab provides an API that is available in the Matlab scripting language (*m-file*), for procedurally creating and manipulating Simulink/Stateflow models. The simulation engine produces an *m-file* that uses the API to create and simulate Simulink/Stateflow models.

To apply our methods of SNF and DSNF to charts modeled with Stateflow, our prototype needs access to its model data. Basically, the API offers users a way to load, save and simulate models non-interactively. However, the API can be used to get information of the chart object by object. We use this information to create an equivalent chart in our prototype. The exchange process is initiated by KIEL. After retrieving the model data, the Statechart data will be adapted and depicted within KIEL.

The simulation of Stateflow Charts in KIEL is also performed by the API. Before a simulation step can be performed by KIEL, we have to ensure that Stateflow can access the chart to be simulated. When a step in KIEL is triggered, the tool collects all information that is necessary to proceed a step in Stateflow. Then KIEL sends the collected components and the simulation command to Matlab. Last, KIEL collects the simulation results and displays them. In doing so, KIEL works as a wrapper for the simulation with Stateflow (see Figure 43). Thus, using the KIEL tool to load, save and simulate Stateflow Charts, KIEL communicates to Matlab Simulink/Stateflow, which works in the background. Figure 44 presents a screen shot of KIEL as it simulates an SUD developed using Matlab Simulink/Stateflow.

In addition to the model import capabilities, as the Stateflow import mechanism used in our case study, KIEL provides a built-in, structure-based editor, and can also synthesize graphical models from textual descriptions (see Section 3.4 and Section 3.5).

6.5 DEVELOPING MODELS IN KIEL—THE EDITOR

We have implemented the above proposed Statechart editing techniques described in Section 3.3 and Section 3.4 in KIEL, which resulted in the *KIEL macro editor* and the *KIT editor*. Both editors are

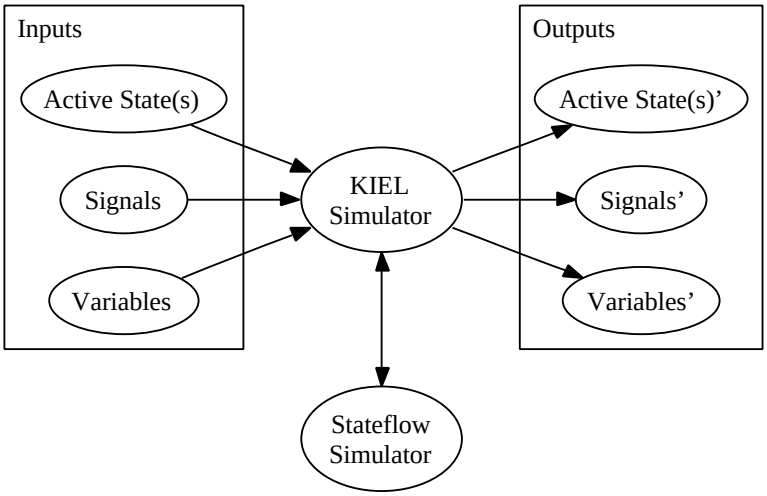


Figure 43: The information flow between Stateflow and the KIEL simulator

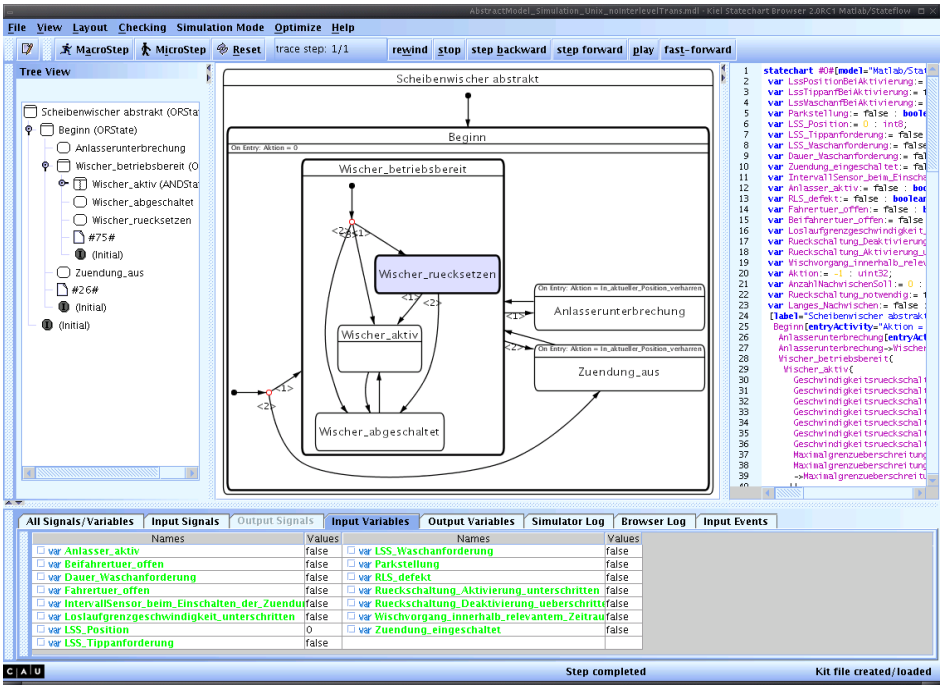


Figure 44: Screen shot of KIEL simulating the window wiper introduced in Section 4.2

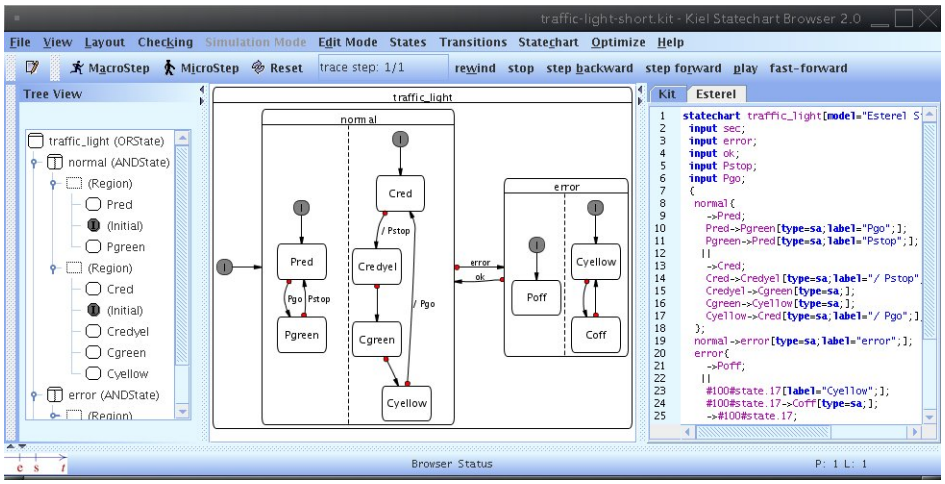


Figure 45: Screen shot of KIEL displaying the Statechart tree structure, the graphical model, and the KIT editor

accessible simultaneously and are arranged side by side so that they allow alternative views on the same SUD, as can be seen in Figure 45. The user may thus choose to manipulate either the textual or the graphical view, and the tool keeps both views automatically and continuously consistent.

Interaction of the KIEL macro editor and the KIT editor

The KIEL macro editor is implemented as an extension of the graphic displaying window; there, the modeler marks and modifies graphical elements directly. The KIT code is kept in sync with the graphical model. In the opposite direction, if the modeler is using the KIT editor, the graphical model is synthesized from KIT code. This combines the advantages of both techniques to allow the designer to work with textual and graphical representations of the SUD simultaneously.

This bidirectional process employs a parser/synthesizer generated by SableCC [52, 53]. A tool which is similarly generated performs the application of the production rules using the KIEL macro editor. The productions are specified using an underlying grammar; the set of productions can be easily extended with further production rules. Figure 46 depicts the tool chain of the KIT editor and the KIEL macro editor and their integration within KIEL. The solid lines characterize the information flow during runtime, and the dashed lines represent dependencies during compile time of KIEL.

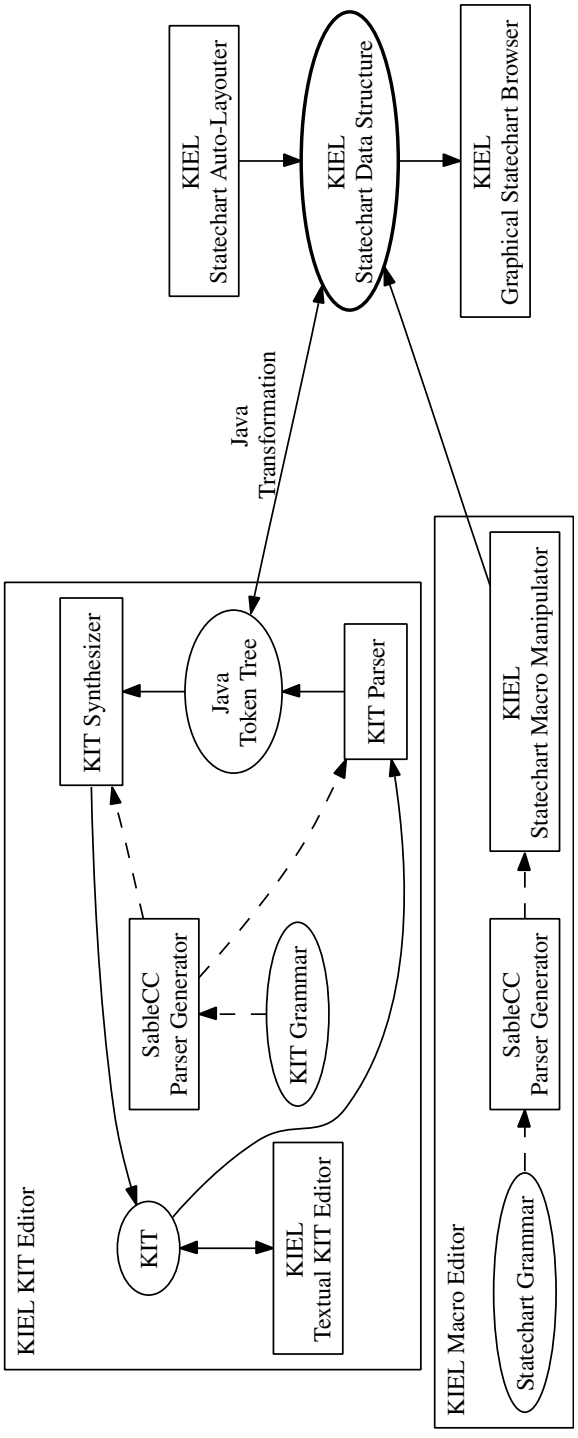


Figure 46: Integration of the KIT editor and the KIEL macro editor into KIEL

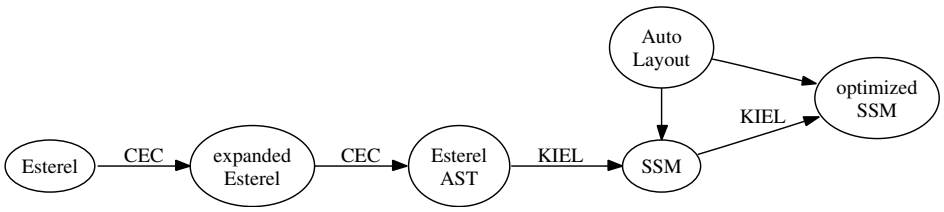


Figure 47: Tool chain transforming Esterel

6.6 SYNTHESIZING STATECHARTS FROM ESTEREL

In KIEL, we also implemented the SSM synthesis and optimization described in Section 3.5 and Section 3.5.2. As an alternative to import a Statechart from another tool or to editing a Statechart using the built-in graphical editor, the modeler may simply import an Esterel v5 program, which is transformed on the fly into an SSM and can then also be saved as SSM. Figure 47 shows the tool chain used in KIEL. When importing an Esterel program, KIEL first employs the Columbia Esterel Compiler (CEC) [34] to perform the expansion of all sub-modules which potentially exist in an Esterel program. Then the CEC is used again to transform the code into an XML formatted Abstract Syntax Tree (AST) representation. We process this using Java XML extensions and apply the transformation rules of Section 3.5 to produce the topology of the synthesized SSM. The KIEL auto-layouter augments this topology with graphical layout information; with numerous configuration options the appearance of the synthesized SSM can be adapted. In general, the whole process is not noticeably slower than importing an already existing Statechart from another tool. As a final step, we may apply the optimization rules specified in Section 3.5.2. This is typically done all at once, but if the user wishes to trace the effect of individual optimization rules, one may also perform the optimizations step-by-step. After each step, a new SSM layout is computed.

Usage of the CEC

6.7 STYLE CHECKING IN KIEL

The checking plug-in of KIEL was designed to be very flexible in usage. All checks have been implemented independently. It is easily possible to manually select which checks to apply via the user interface of KIEL. Furthermore, it is possible to define profiles containing different sets of rules in accordance with different State-

chart dialects. Dependent on the Statechart dialect of a model loaded into KIEL, the plug-in decides automatically which profile can applied. Figure 48 shows a screen shot of KIEL as it checks the UML realization of Harel's wristwatch example [71] with respect to robustness rules.

Realization of the syntactic robustness checker

Great care was taken to develop the plug-in to be easily extendable with new checks. Therefore, the user can extend the rule set by either adding an appropriate OCL constraint for a syntactical check or by adding a new Java class for semantical checks. Depending on the seriousness of a detected problem, the robustness checker delivers two kinds of messages.

ERRORS in modeling are violations of rules that have to be addressed because further work such as simulating the model is impossible.

WARNINGS indicate that a problem was found, which does not need to be fixed immediately for simulation, *i.e.*, possible sources of errors or ambiguous constructs. In the following, an overview of the implementation of the aforementioned rules is presented.

Our intention to use OCL was based on the benefits stated by Mutz, *e.g.*, checks can be formulated on a higher level of abstraction, and neither knowledge of a programming language nor of the underlying data structure is needed [109]. The evaluation of OCL constraints based on a transformative approach is preferable to an interpretative approach. The transformative approach proved to be more flexible and faster in execution time. Therefore, we chose to use the *Dresden OCL Toolkit* v. 1.3 [159] discussed by Richters [137] to transform OCL constraints to Java.

The Transformation of OCL to Java code makes the style checking very efficient.

Our approach for the checking framework contains the possibility of returning customized messages when a violation is found. Therefore, the OCL constraint is encompassed by additional information as Java code snippets. The union of OCL and Java code snippets were named KIEL wrapped OCL (KOCL). The developed KOCL to Java translator utilizes the Dresden OCL Toolkit which is supplied in accordance with the meta model of the KIEL data structure. Figure 49 basically shows how the different parts of the KOCL files are processed. The workflow and the specified rules were described in detail by Bell [15].

KOCL allows to return customized messages.

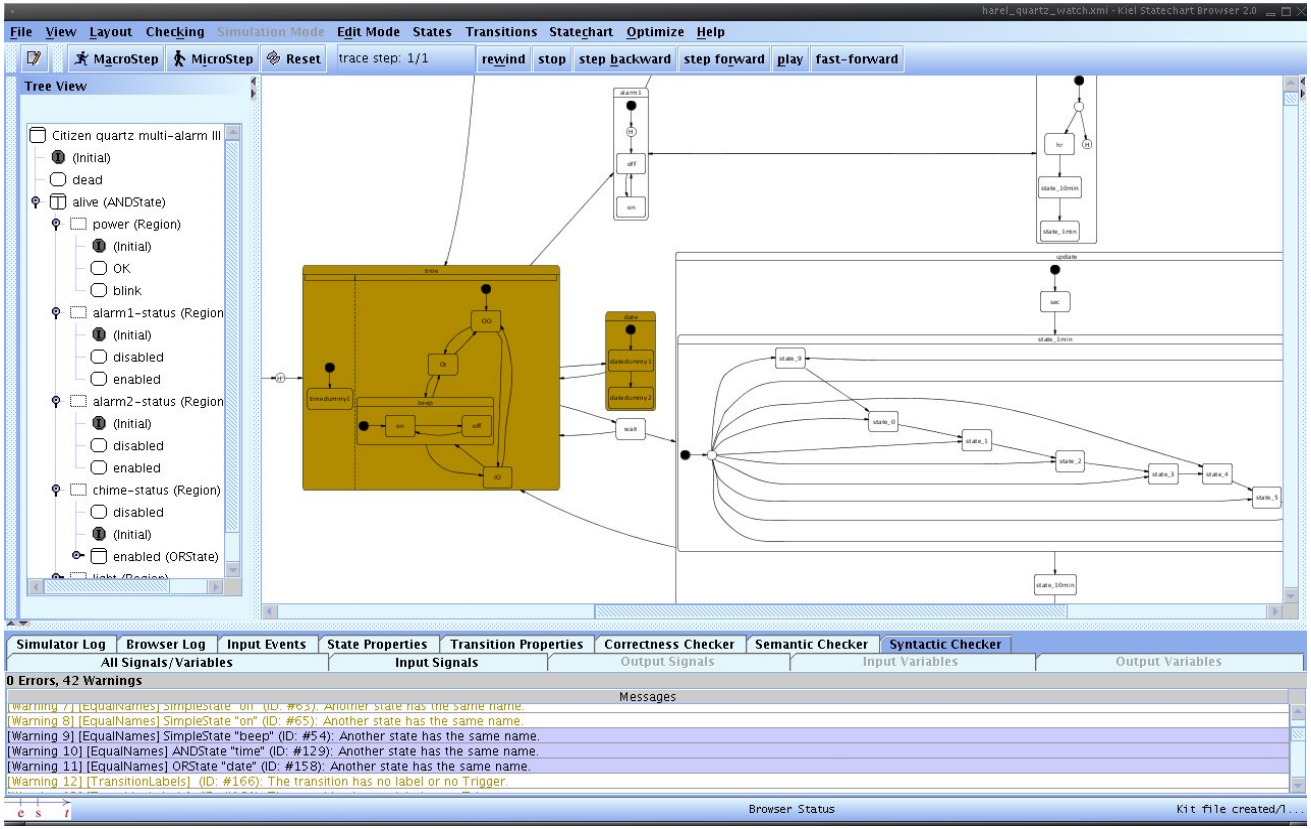


Figure 48: Screen shot of KIEL checking robustness of the UML realization of Harel's wristwatch example [71]

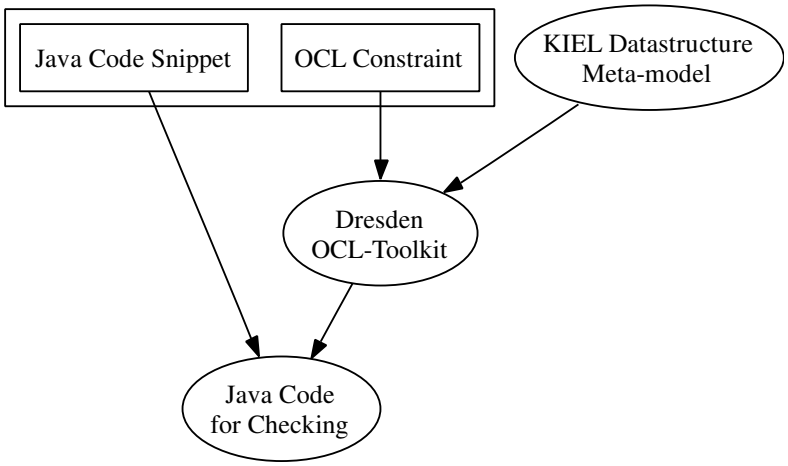


Figure 49: Processing KOCL with KIEL

As the framework is designed to handle rules formulated as OCL constraints, we have implemented the rules elaborated above (cf. Section 5.4). Most of the well-formedness rules were specified in KOCL. The rules not specified in KOCL deal partly with features of UML diagrams. As the KIEL project is focused only on simulating and modifying Statecharts so far, the representations of classes and packages were left out for the sake of simplicity. Hence, *e. g.*, rule *State Machine number 1* which states that "a State Machine is aggregated within either a classifier or a behavioral feature" from the UML specification was left out.

We will not present all of the transfered rules in detail. The example presented in the following gives an overview about how the additional information is capsuled within KOCL files. An example which is easy to understand is Rule *CompositeState-1* (cf. Section 5.4, Figure 36b) as specified in Figure 50a. The OCL constraint states that the set *subvertex* of a composite state (here: *self*) can contain at most one *Pseudotate* of kind *initial*. The *Dot* notation is used to access members of a class. An arrow (->) is used to access properties or functions on sets.

The rule specified in KOCL is presented in Figure 50b. The separation of the message declaration, the constraint definition and the specification of the returning message is clearly seen in this example. The *declarations* part is designed to hold more than one message. The *fails* part specifies which message to return to, if a violation of the constraint is found. It is even possible to return different messages (if defined) depending on the context in the

```
self.subvertex->select(
  v| v.oclIsKindOf(Pseudostate))->
select(
  p:Pseudostate| p.kind = #initial
       )->
size <= 1
```

(a) The UML Representation

```
rule UML13CompositeStateRule1 {
  declarations {
    message "A composite state can
         have
          at most one initial vertex
               .";}
  constraint {
    context ORState or Region;
    "self.subnodes->select(
       v| v.oclIsTypeOf(InitialState))
            ->
       size <= 1";}
  fails {message;}
}
```

(b) The KOCL Representation

Figure 50: The rule *CompositeState-1*

fails part by simply using a common *if-then-else*-statement. The constraint itself is even shorter than specified in the UML due to the meta model.

The *Transition Overlap* rule, the *Dwelling* rule, and the *Race Conditions* rule (see Section 5.4) cannot be specified using OCL constraints. Java code is needed for formulating theorem-proving queries and sending them to an outside tool for analysis.

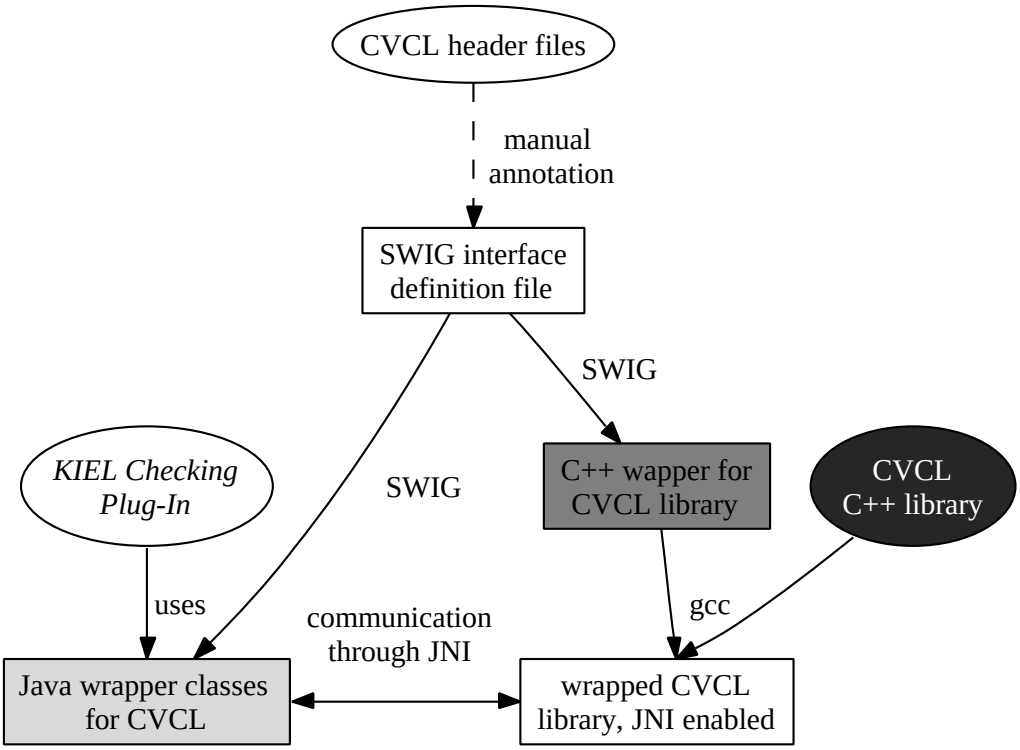
Realization of the semantic robustness checker

To perform the semantic robustness checks, a Satisfiability Modulo Theories (SMT) solver [9] is needed. SMT problems are a variation of Automated Theorem Proving (ATP) [54], which in turn, is part of automated reasoning. After a thorough evaluation of available SMT solvers, CVC Lite [10] was chosen. Here, in order to determine whether, *e. g.*, two transitions $trans_1$ and $trans_2$ (cf. Section 5.4) have overlapping labels, satisfiability of the formula

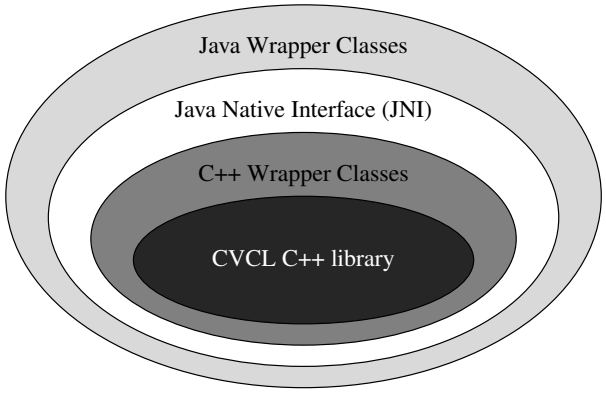
$$\Big((e_1 \wedge c_1) \wedge (e_2 \wedge c_2)\Big)$$

must be decided. Unsatisfiability implies that the predicates of $trans_1$ and $trans_2$ are disjoint.

Furthermore, the Simplified Wrapper and Interface Generator (SWIG) [12] was employed to generate wrappers and interface files for CVC Lite, enabling its immediate use from within Java. Here, the Java and C++ Java Native Interface (JNI) wrappers are produced from CVC Lite's annotated C++ header files, as shown in Figure 51.



(a) Outline of the SWIG workflow, including the link-up of the CVC Lite library to the KIEL Checking plug-in



(b) Conceptual diagram of the hierarchical composition of wrapper layers around the CVC Lite library

Figure 51: Interfacing of KIEL and the CVC Lite library via JNI and SWIG (source Schaefer [144])

7 USABILITY ANALYSIS OF KIEL

We have used successive generations of KIEL in the classroom since 2005, including the macro-based and text-based editors presented here. The feedback on these editing approaches has been generally quite positive, also in comparison with the classical editing paradigms employed by the other commercial modeling tools, also used in classes. However, to gain a better, objective insight into the effectiveness of our modeling approaches, we have performed an experimental study in order to investigate the differences in editing performance between the conventional WYSIWYG approach, the KIT editor, and the KIEL macro editor, in combination with our idea of an SNF, described in Section 3.2. The design and the statistical analysis of this experimental study was supported by Prof. Dr. Jürgen Golz (Department of Psychology, CAU Kiel) and Christiane Gross (Department of Sociology, CAU Kiel). In this chapter, experimental results using KIEL are presented and discussed in two parts: (1) Section 7.1 presents the results of a controlled experiment using KIEL. (2) Section 7.2 presents a performance analysis, which provides measurements of the capability of techniques introduced in Chapter 3 to Chapter 5 and presents response times of the modeling environment KIEL.

7.1 AN EMPIRICAL STUDY ON STATECHART TECHNIQUES

In the field of computer science, knowledge obtainment results usually from mathematical analysis of a referred software process or method. Due to the complexity of such methods or raising of psychological questions of software tools or methods, empirical methods are often used in practice. Empirical studies affirm research results and bring new insights. Quoting Freimut et al.:

> *"Empirical studies, such as case studies, experiments or surveys investigate the strengths and weaknesses of existing software engineering methods, techniques and tools. Empirical studies result in empirical knowledge or proven concepts."* [46]

However, empirical studies in computer science are highly underestimated and they are often omitted. *E. g.*, Lukowicz et al. determine that

> *"[...] over 40 % of articles about new designs and models completely lack such experimentation. For samples related to software engineering, this factor is higher; it is over 50 % for IEEE Transactions of Software Engineering."* [96]

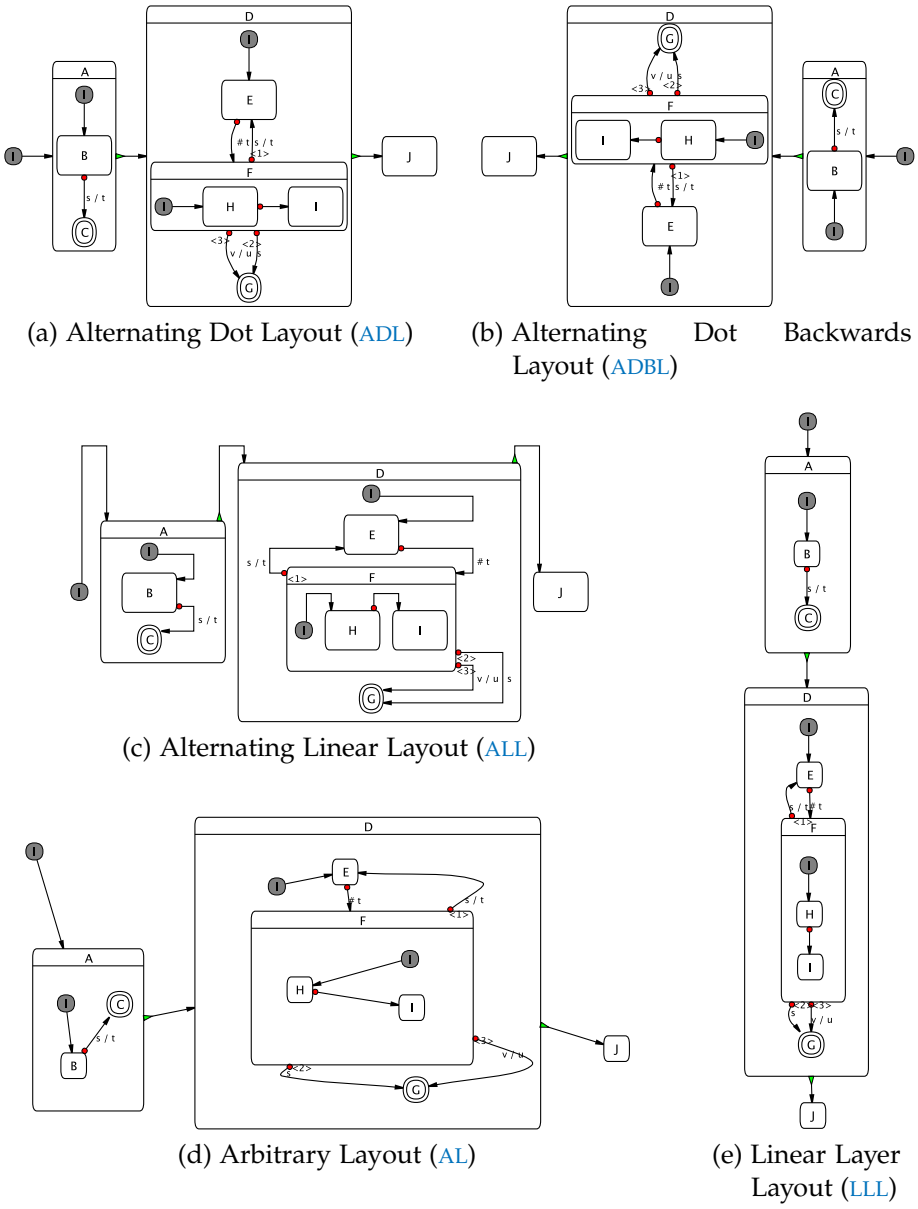
The reason for the lack of experimental studies can be ignorance or fear of experiment results. Wohlin et al. [170] give a well-structured and easy-to-understand introduction to experimentation for software engineers for *controlled experiments*, and Juristo and Moreno [82] describe in-depth methods for advanced experiment designs and data analysis with examples from software engineering. Prechelt [126] includes many practical hints for design, execution and analysis of controlled experiments; the proposals of Prechelt have been the basis of the experimental design described in this section.

As mentioned in Section 6.2, KIEL provides a mechanism that automatically produces our preferred Statecharts arrangements; in the following, we call this the Alternating Dot Layout (ADL) (see Figure 52a). Hence, a further goal of the experiment was to compare the readability of the ADL with other layout strategies.

7.1.1 *Experimental Design*

The experiment was conducted twice: for beginners and for advanced modelers.

The participants in the experiment (the *subjects*) were graduate-level students attending the lecture "Model-Based Design and Distributed Real-Time Systems" in the Winter Semester 2006/07 [163]. Most of them were not familiar with the Statechart formalism in advance. The experiment consisted of two parts. The first part took place early in the semester, after two lecture units introducing the Statechart formalism. The subjects had by then also worked on a first assignment on understanding Statechart semantics. The second part proceeded after the final lecture unit at the end of the semester. In the meantime, the subjects had gained practical experience in modeling Statecharts. Furthermore, they had learned about the importance of modeling paradigms, such as maintainability and secondary notation of Statecharts. 24 students participated in the first experiment, 19 of them took the second one. In the



(a) Alternating Dot Layout (ADL)

(b) Alternating Dot Backwards Layout (ADBL)

(c) Alternating Linear Layout (ALL)

(d) Arbitrary Layout (AL)

(e) Linear Layer Layout (LLL)

Figure 52: Different Statechart layouts for experimental comparison

following, we refer to the participants of the first experiment as *novices* and to the participants of the second experiment as *advanced*. Furthermore, we define as *experts* the modelers that have significant practical experience beyond course work. Both experiments have similar design and consisted of three parts:

1. *Modeling Technique Evaluation:* The subjects had to create Statecharts of complexity using different Statechart modeling tech-

niques: a graphical WYSIWYG Statechart editor (we decided to employ Esterel Studio), the KIEL macro editor, and the KIT editor. Every subject developed the same Statechart model with each referred editor, following a semi-formal Statechart description (see Section C.1) which was borrowed from the Statechart formalization of Pnueli and Shalev [123]. Afterwards, the created Statechart had to be extended and modified. The subjects received instructions on a one-page reference card per modeling tool (cf. Section C.3.1, Section C.3.2, and Section C.3.3). As editing performance metric the elapsed time was measured.

2. *Subjective Layout Evaluation:* The subjects were asked to score readability and comprehensibility of five different Statechart layouts (cf. Figure 52). The idea of this experiment was to vote for a presumed readable and understandable Statechart layout. Therefore, pairs of two Statecharts had to be compared. After a 10-second decision finding, the subject had to decide his preference—left-hand-side or right-hand-side.

   The experiment covered five different Statechart layouts (cf. Figure 52); every layout was paired with each other. Pairs were presented for Statecharts with simple states, hierarchical, and parallel states. Appendix B shows 15 examples of different laid out Statecharts and different Statechart complexity. A total of 75 Statechart layouts were presented to the subjects. The layouts in Figure 52a, Figure 52b, and Figure 52c were automatically generated from the KIEL layout mechanism; the Statechart models in Figure 52e and Figure 52d were drawn manually.

3. *Objective Layout Evaluation:* In this experiment, the subjects had to analyze different Statechart models and they had to construct a sequence of active states and upcoming signal events according to the semantics of SSMs. For this experiment, the set of Statechart layouts was similar to the previous experiment. Unlike the Statechart pairing of the previous part, each Statechart had to be understood according to the semantics of SSMs. Therefore, active states and upcoming signal events had to be entered into a prepared time scale. The elapsed time and the number of each Statechart reading was measured for performance evaluation.

Each subject had to process a personal experiment assignment (see Section 7.1.3) with randomly varied conditions. The assignments contained the tasks described above. The study was realized as a controlled experiment, *i.e.*, the experiment leader checked and rejected the solutions of Part 1 and Part 3 in case of incorrectness. In Section C.1, a complete example of a working document is presented, as handed out to the subjects. The answers had to be entered into the answer template document; an example is presented in Section C.2. Each of the subject's experiments was performed in a single session of one to two hours. The sessions were videotaped, with the subjects written consent, in order to allow later off-line analysis .

7.1.2 *Hypotheses*

The main questions asked in this experiment are the following:

1. Do the macro-based and text-based editing techniques make the Statechart construction process easier and faster than the conventional WYSIWYG method?

2. Are the resulting Statecharts more readable and comprehensible?

To guide the analysis of the results, it is useful to formulate some explicit expectations in the form of hypotheses about the differences that might occur. Hence, the experiment should investigate the hypotheses as follows:

1. *Statechart Creation:* We expect that novices will need less time to create a Statechart using the WYSIWYG editor compared to the usage of the KIEL macro editor or the KIT editor. However, the Statechart creation times of advanced modelers using the KIT editor should be less than when using the WYSIWYG editor.

2. *Statechart Modification:* We expect that the modification of an existing Statechart using the KIT editor or the KIEL macro editor is faster than using the WYSIWYG editor.

3. *Aesthetics:* Statecharts are sensed as aesthetic, if their elements are arranged conforming to a certain layout style guide.

We expect the best aesthetic scores for Statecharts laid out according to the ADL (see Figure 52a).

4. *Comprehension:* We suppose that well-arranged Statecharts influence the readability. Hence, we expect a faster comprehension of the ADL compared to other Statechart layouts.

7.1.3 *Quality of Experimental Data*

Concerning the *internal validity*, all relevant external variables (subjects' Statechart modeling experience, maturation, aptitude, motivation, environmental condition, *etc.*) were equalized between appropriate groups by randomized group assignment. Regarding the *external validity*, there are several sources of differences between the experimental and real Statechart modeling situations that limit the generalization of the experiments: In real situations, there are modelers with more experience, often working in teams, and there are Statechart models of different size or structure. However, we do not consider this to invalidate the basic findings of the experiment. In addition, we suppose that the differences between novices and advanced modelers become even larger when we move from simple Statechart examples as used in our experiment to complex Statecharts as in practice.

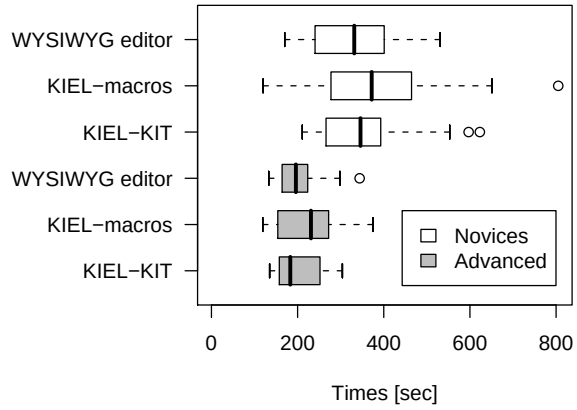
Applicability of experimental results to real world situations

7.1.4 *Results*

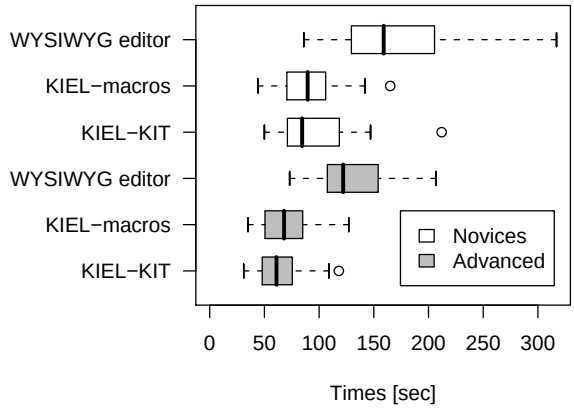
This section presents and interprets the results of the experiments. The analysis is organized according to the hypotheses listed in Section 7.1.2. Box plots present the obtained statistical data. The box plots denote quartiles, small circles indicate outliers. The comparison of means will be assisted by the two sample *t-test* with equal variances. The test compares the difference of sample means from two data series with the hypothesized difference of the series means. It computes the p-value, which indicates a statistical significance. We will call a difference significant, if $p < 0.05$. The analysis and plots were performed with *R* v. 2.4.0 [132].

Evaluation of Modeling Techniques

The plots in Figure 53a and the statistical tests corroborate our Hypothesis 1 for novices. We found out that Statechart creation



(a) Distribution of times for creating a new Statechart



(b) Distribution of times for modifying an existing Statechart

Figure 53: Distribution of times for modeling Statecharts. The box plots denote quartiles, small circles indicate outliers.

Times for Statechart creation

times of novices using a WYSIWYG editor are smaller than using the KIEL macro editor (*t-test* $p = 0.04$) and they tend to be smaller using the KIT editor (*t-test* $p = 0.25$). For advanced, the Statechart creation times using a WYSIWYG editor tend to be smaller than using the KIEL macro editor (*t-test* $p = 0.12$); time differences between WYSIWYG editor and KIT editor are not significant (*t-test* $p = 0.46$). Due to the novelty of the KIEL macro editor and KIT editor, the novices sought advice in the reference cards (cf. Section C.3.2 and Section C.3.3). In contrast, the WYSIWYG editor could be used intuitively and without any reference card. Hence, on average the novices needed less time for creating Statecharts using the WYSIWYG editor than using the KIT editor or the KIEL macro editor. For advanced learners, however, the mean time is slightly less using the KIT editor. We suppose that

for experts in Statechart creation, this difference would increase further.

Times for Statechart modification

Figure 53b illustrates the efficiency using the KIT editor and the KIEL macro editor in Statechart modification; it emphasizes our tests which corroborate Hypothesis 2. The *t-test* resulted for novices and advanced, that the needed times for Statechart modification using the KIEL macro editor or using the KIT editor are smaller than the times using the WYSIWYG editor (both: *t-test* $p = 0.00$).

With the KIT editor and the KIEL macro editor the modeler only works on the Statechart structure, while KIEL's Statechart autolayouter arranges the graphical model. In contrast, the subjects spent most of the time with making room for new Statechart elements and rearranging the existing ones to make the developed chart readable using the WYSIWYG editor. Despite the fewer operations needed using the KIEL macro editor, the subjects needed more time to modify a Statechart. The time was large due to frequent consultations of the reference cards. Hence, we suppose that for experts the KIEL macro editor will provide the fastest modeling method.

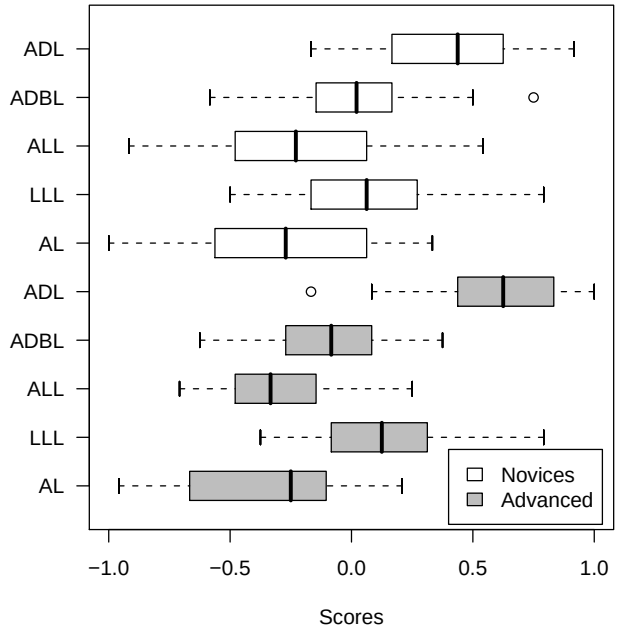
Evaluation of Statechart Layouts

Results from Statechart layout comparison

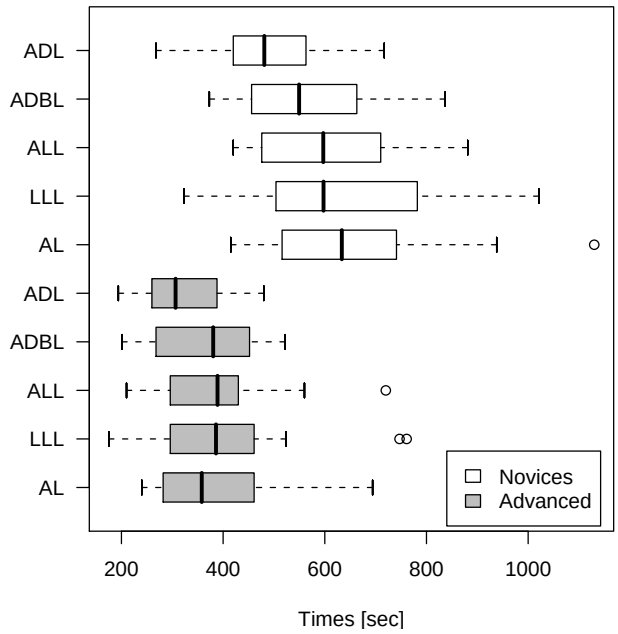
The scores of subjective Statechart layout assessment (cf. Figure 54a) clearly show the subjects' preference for Statecharts laid out according to the ADL. In the figure there means a score of -1.0 a strong rejection and a score of 1.0 means a strong preference for a certain Statechart layout. Also the *t-test* denotes, that hypothesis 3 can be retained. For novices and advanced are the ADL scores better than all other Statechart layouts (all layouts: *t-test* $p = 0.00$). Apparently, it is not sufficient that layouts underlie an automatic layout; in fact, Statechart layouts have to satisfy certain aesthetics to be assessed as good layouts. Accordingly, subjects stated that "transitions must be short and traceable" and "the element structure has to follow the Statechart meaning". *E. g.*, due to unnecessarily long transitions, the ALL scores are lower than the LLL.

Results from Statechart comprehension

Figure 54b demonstrates that a proper layout enhances the readability of Statecharts. This also shows that novices needed less time for comprehending Statecharts according to ADL (ADBL: *t-test* $p = 0.02$, others: *t-test* $p = 0.00$). The times of advanced for reading ADL layouts tend to be smaller than times of the ADBL (*t-test* $p = 0.1$). Less time is needed for other layouts (ALL: *t-test* $p = 0.04$,



(a) Distribution of subjective Statechart layout scores



(b) Distribution of Statechart comprehension times

Figure 54: Distribution of Statechart layout assessments

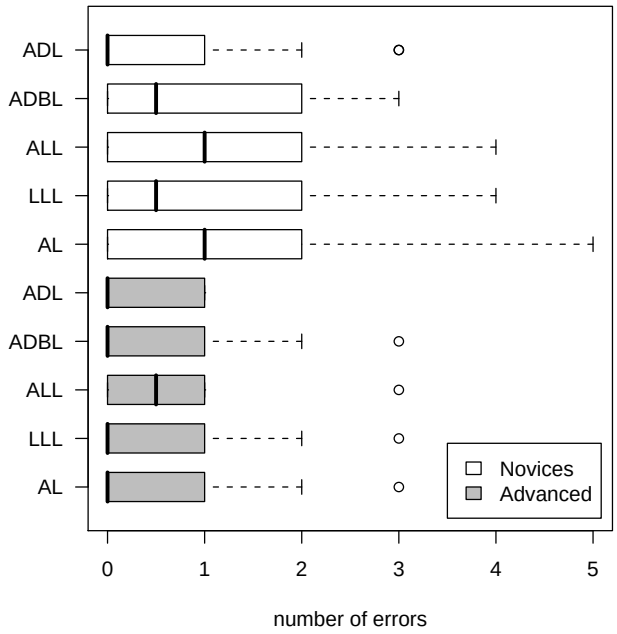


Figure 55: Comprehension errors during Statechart reading

LLL: *t-test* $p = 0.03$, AL: *t-test* $p = 0.03$). Statecharts laid out according to the ADL are comprehensible faster than other Statechart layouts, that corroborates hypothesis 4. The subjects—novices and advanced—also made fewer errors while reading Statecharts laid out according to the ADL.

The box plots in Figure 55 support the statement of Figure 54b and emphasize the quality of our referred Statechart layout. The better readability and less error-prone reading of the ADL Statechart result from the accompanying proper micro layout (*e. g.*, label placement), as well as proper macro layout; *e. g.*, compact and white-space avoiding element arrangement.

7.2 PERFORMANCE ANALYSIS OF KIEL

In the previous section we have shown an experimental study which demonstrates the efficiency of our editing proposals introduced in Section 3.2 to Section 3.4. In addition to a satisfying functioning of a modeling, short answer times on function requests are substantial for an adequate interaction with a Statechart modeling tool. Especially, if one manipulates large-scale Statecharts, the number of graphical objects drawn by the visualization component

can be enormously high. This can bring the user interface to its limits. Hence, when modeling complex Statecharts, it is indispensable for the usability of a modeling tool, that its response times do not increase indefinitely.

This section presents response times of the modeling environment KIEL which were measured for the main components of KIEL processing different sizes of Statechart models. The times were measured on a PC with Linux OS, a 2.6 GHz AMD Athlon 64 processor and 2 GB of RAM.

7.2.1 *KIEL's Simulation and Visualization Performance*

To assess the efficacy and efficiency of KIEL's simulation and visualization performance, Table 1 presents experimental results for simulating the traffic light example, the window wiper example, and two types of generic scalable models. The *bintree*⟨n⟩ models are those whose state trees consist of binary *OR* trees (which are strictly sequential) of height n (see Figure 65 on page 148 f). Furthermore, the *quadtree*⟨n⟩ models have alternating *AND*/*OR* trees (which include concurrency) of height 2n (see Figure 66 on page 152 f). The *graphical elements* category characterizes the graphical complexity of the models. The *state space* category compares the number of reachable configurations with the number of possible views. As the degree of concurrency increases, the ratio between the numbers increases as well, which is desirable to minimize view changes and—if layout information is cached once it is computed, as in our case—to keep memory requirements small. The *occupied area* compares the size of a computed static view of the whole system with a dynamic view. We choose to depict the minimal size of all occurring views for the size of DSNF.

Assessment of simulation using Dynamic Statecharts

As to be expected, the bigger the system, the smaller becomes the DSNF relative to the static SNF. The two lines in Figure 56 compare the size of occupied areas of the static and the dynamic view of Statecharts different sizes. The figure also emphasizes that, for *complex Statecharts* with more than thousand graphical elements, the occupied area of the static views increases significantly. In contrast, the areas of the dynamic views remain *almost constant* for complex systems. This strongly emphasizes the intention of the DSNF: Our method reduces the number of visualized graphical

Table 1: Simulation and visualization performance of KIEL

MODEL	GRAPHICAL ELEMENTS			STATE SPACE		OCCUPIED AREA				COMPUTATION TIMES							
	States	Transitions	Total	Configurations	Views	Static view (pixel × pixel)		Dynamic view (pixel × pixel)		Reduction factor	Load (ms)	Next configuration (ms)		Layout (ms)		Display next view (ms)	
												min	max	min	max	min	max
traffic-light	15	25	40	10	2	652	460	398	268	0.36	828	31	67	1	24	213	356
window-wiper	36	82	118	80	12	4200	2895	870	311	0.02	1120	535	1497	1	50	497	1759
bintree-1	3	5	8	2	1	248	103	248	103	1	226	5	26	1	5	147	273
bintree-2	9	15	24	4	2	288	268	268	268	0.93	346	3	46	1	18	172	274
bintree-3	21	35	56	8	4	636	384	442	342	0.62	834	31	153	2	44	181	260
bintree-4	45	75	120	16	8	716	712	462	506	0.46	1446	36	188	1	28	264	436
bintree-5	93	155	248	32	16	1412	944	636	580	0.28	1772	41	250	1	32	272	532
bintree-6	189	315	504	64	32	1572	1600	656	744	0.19	2392	45	374	1	38	390	647
bintree-7	381	635	1016	128	64	2964	2064	830	818	0.11	2937	214	709	1	39	456	837
bintree-8	765	1275	2040	256	128	3284	3376	850	982	0.08	8636	555	1389	2	41	954	1802
bintree-9	1533	2555	4088	512	256	6068	4304	1024	1056	0.04	20695	1429	2640	2	48	1911	3504
quadtree-1	10	11	21	4	1	300	268	300	268	1	668	20	54	2	4	214	264
quadtree-2	50	55	105	64	4	696	712	696	506	0.71	1413	277	374	1	27	287	453
quadtree-3	210	231	441	16384	64	1488	1600	1488	744	0.47	1985	545	969	2	77	594	1834
quadtree-4	850	935	1785	1073741824	16384	3072	3376	3072	982	0.29	10186	2220	2852	3	127	3151	3633

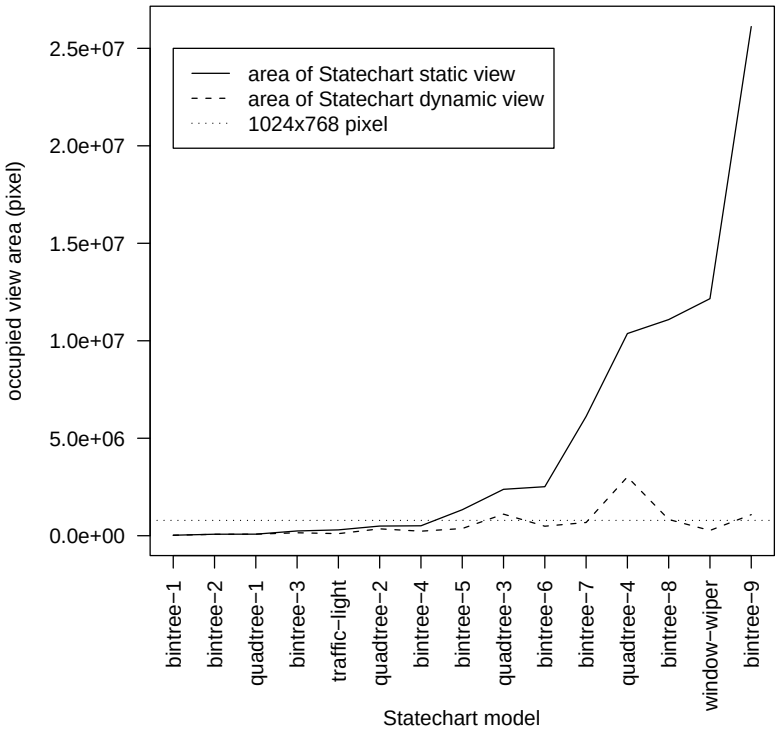


Figure 56: Comparison of static view and dynamic view areas

elements on a screen and this reduces the effort to keep the whole system under simulation in mind.

Note that the static SNF already reduces the size compared to a manual layout. The DSNF allows the full view of comparatively large models without losing details, but of course, there still comes a point when details become unreadable. However, in our experience, even for very large models, the full-model DSNF view provides a very valuable overview of where the "hot spots" of a model are during simulation, which is the difficult part with the traditional Statechart animation part. Once these areas of interest are detected, it is easy to use normal zooming and panning to get more detailed information, such as state names or transition labels.

Assessment of computation times for next simulation views

Similarly, the time to compute the *next configuration* during a simulation is typically less than two seconds, despite our rather inefficient, interpretative simulation approach. Figure 57 compares the total of the time for the computation of a new configuration, the computation times of a layout, and the times for display of a new view of different Statecharts. What we consider most remarkable, however, is the efficiency of the layout computation. This was

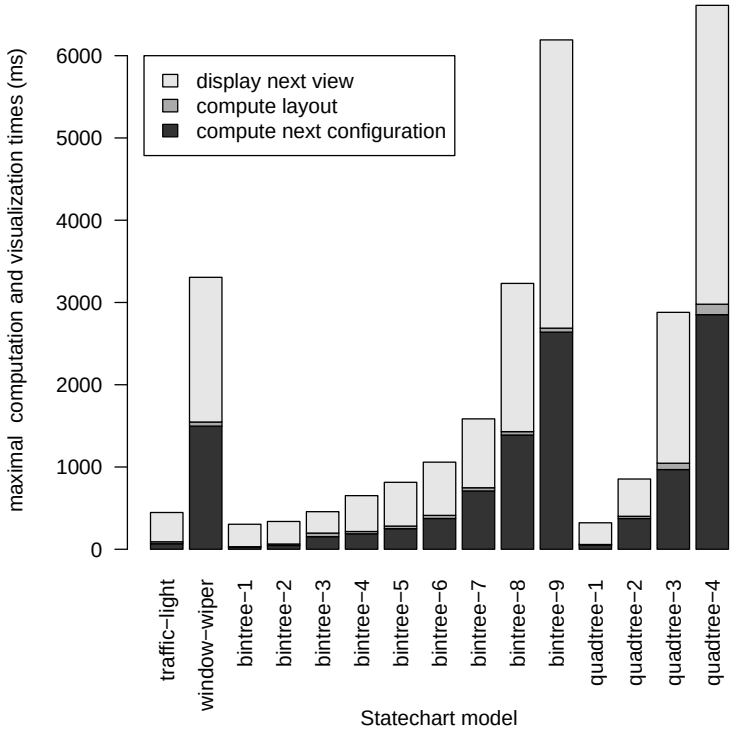


Figure 57: Maximal computation times in KIEL to present a new view during simulation

always in the sub-second range, even for the largest models we simulated, which underlines the efficiency of the GraphViz layout system and our hierarchical layout scheme. The times to display the *next view* are hence not dominated by the layout, as we had originally expected, but by the simulation itself and the rendering mechanism, both of them still leave room for optimizations, that we have not applied yet.

Furthermore, Figure 58 illustrates that we can assume a linear increase of view computation times in progress of the number of graphical elements. In the figure there are two linear functions depicted. They represent convergence functions of the computation times for our both generic *tree* Statechart example series. To the linear scaling view computation times always a fix amount of time has to be added. This causes from the times for rendering and simulation as mentioned above. The stronger increase of view

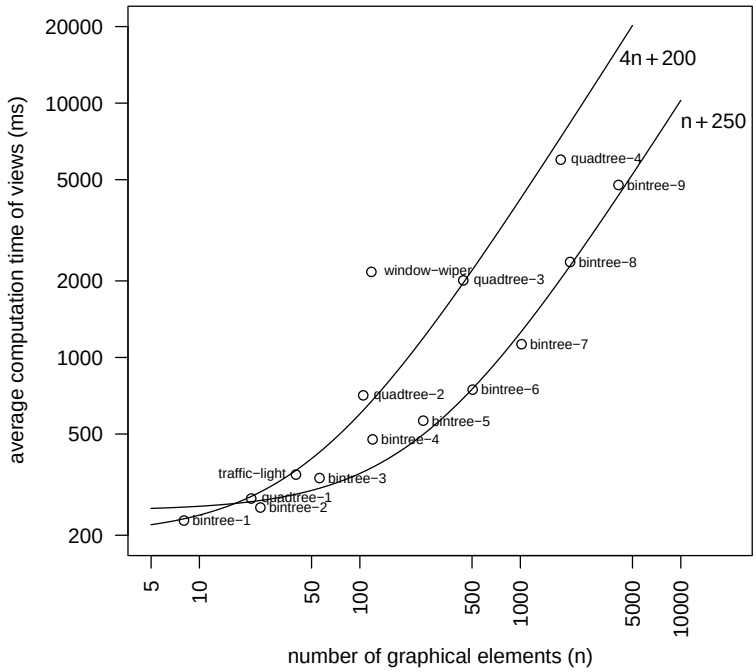


Figure 58: Average computation times in KIEL to present a new view

computation times for the quadtree examples have their cause in the occurring parallelism. Overall, the collected view computation times suggest the complexity class $O(n)$, where n is the number of graphical elements, as an upper bound to the view computation times.

Assessment of model load times

To quantify efficiency, the *load* time indicates how long it takes to load and visualize a static system view. For very large models, this time becomes noticeable, but typically it takes less than five seconds (cf. also Figure 59). The Statechart load times increase relatively to the model size. However, they do not increase indefinitely: the time tends to converge towards the square root function of the number of graphical elements in a chart. Obviously, the framework needs a fix period of computation time, which is independent from the model size. Variable computation times dependent on the model size have to be added to the fix period. However, we expect linear increase of the load times for more complex Statecharts.

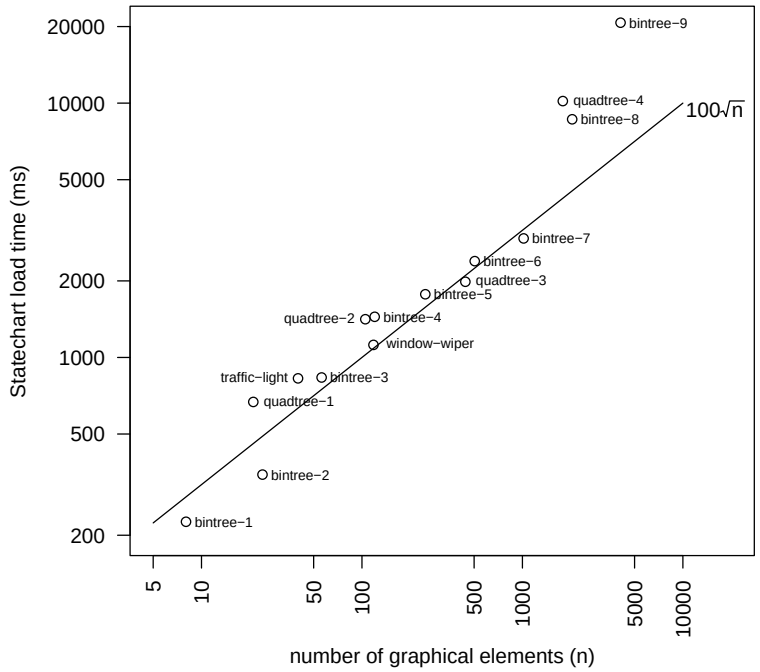


Figure 59: Load times of Statechart models in KIEL

7.2.2 *Analysis of SSM synthesis*

To assess the efficacy and efficiency of SSM synthesis from Esterel with KIEL, Table 2 presents experimental results for transforming various Esterel benchmarks, most of them are taken from the Est-bench Esterel benchmark suite [37] and the CEC distribution [34]. The table compares the size of Esterel code with the complexity of the corresponding synthesized SSMs. *Lines of code* denote the overall size of the (module expanded) Esterel code. The *graphical elements* category characterizes the graphical complexity of the resulting SSM models before and after applying the optimization. We observe that, as the degree of complexity of the resulting chart increases, the ratio between them increases as well, which is desirable to minimize the number of graphical elements. The element-wise reduced Statechart is more compact and has less overhead and redundancy; the optimization often reduces the Statechart by a factor of three or more. Hence, in our experience a synthesized, optimized Statechart is generally very readable and comprehensible.

Figure 60 illustrates the linear increase of generated graphical Statechart elements. The size of circles indicates the intensity of the

Table 2: Experimental results of SSM synthesis

MODEL	ESTEREL	SAFE STATE MACHINES											TIME	
		Before optimization					After optimization							
	Lines of code (loc)	States	Pseudo-states	Transitions	Total graphical elements (ges)	ges/loc	States	Pseudo-states	Transitions	Total graphical elements (ges)	ges/loc	SSM reduction factor	Transformation (ms)	Optimization (ms)
ABRO	7	12	8	12	32	4.57	8	4	8	20	2.86	0.63	861	56
schizophrenia	10	13	9	14	36	3.60	4	3	6	13	1.30	0.36	838	81
reincarnation	25	27	17	28	72	2.88	5	2	6	13	0.52	0.18	642	83
jacky1	27	31	19	33	83	3.07	11	5	12	28	1.04	0.34	913	93
runner	55	39	24	42	105	1.91	21	11	25	57	1.04	0.54	725	160
tokenring3	79	77	61	91	229	2.90	15	20	38	73	0.92	0.32	733	278
greycounter	82	211	148	254	613	7.48	42	50	106	198	2.41	0.32	789	593
abcd	101	231	130	250	611	6.05	78	41	97	216	2.14	0.35	943	731
tokenring10	247	245	194	294	733	2.97	43	62	122	227	0.92	0.31	1267	736
mejia	555	374	246	414	1034	1.86	127	76	181	384	0.69	0.37	3085	1266
tcint	687	475	285	543	1303	1.90	163	81	221	465	0.68	0.36	3382	1310
atds-100	948	961	558	1092	2611	2.75	352	184	504	1040	1.10	0.40	7046	2760
ww	1088	342	228	386	965	0.89	102	85	177	364	0.33	0.38	4470	1053
tokenring50	1207	1205	954	1454	3613	2.99	203	302	602	1107	0.92	0.31	6608	8148
tokenring100	2407	2405	1904	2904	7213	3.00	403	602	1202	2207	0.92	0.31	20910	25528
mca200	7269	5159	3931	5947	15037	2.07	179	925	1794	2898	0.40	0.19	61510	100594

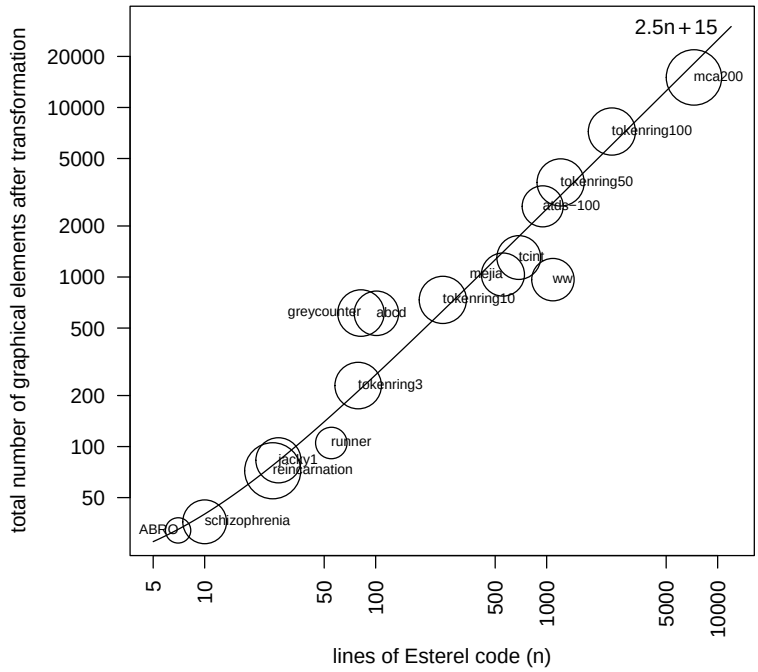


Figure 60: Relation of lines of Esterel code and number of generated graphical elements. The size of circles denotes the the optimization intensity.

optimization. We can observe, that this intensity is not dependent from the model complexity. The reduction capability of our Statechart optimization method is illustrated in Figure 61. Even huge Statechart models can be reduced to relatively small, thus, readable models.

As an example of intermediate size, Figure 62 shows the component "mode selection" of the canonical wristwatch example [37, 73] which is denoted as *ww* in Table 2. This component contains 38 states, 29 pseudo-states and 62 transitions, hence, a total of 129 graphical elements.

Times for Esterel transformations

To quantify efficiency, the *transformation* time indicates how long it takes to load, expand, parse and transform an Esterel program. For large models, this time becomes noticeable, but it typically takes at most a couple of seconds. Similarly, the time to compute the optimized SSM version is less than three seconds for most of the benchmarks considered here. Only the optimization of the industrially sized example *mca200* takes about 100 seconds.

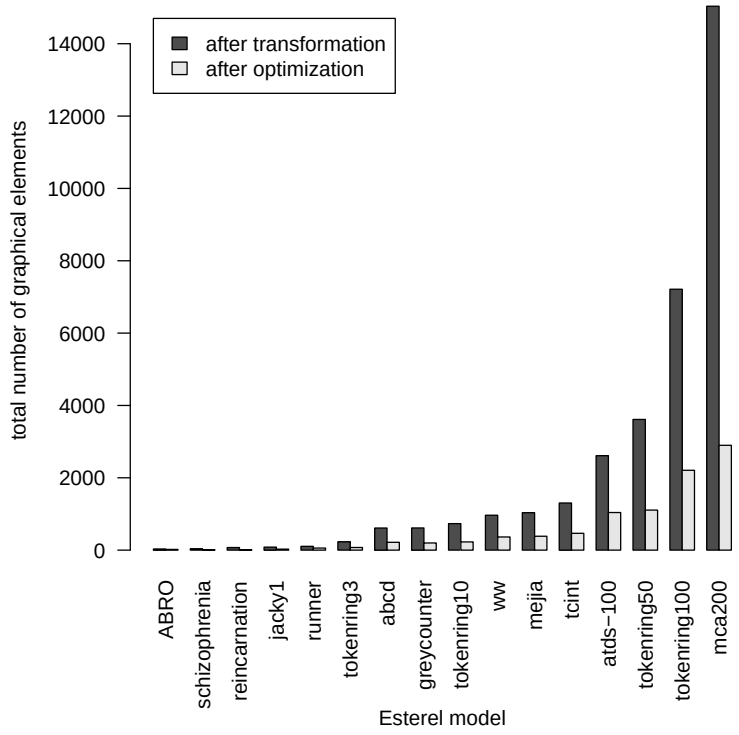


Figure 61: Model size before and after optimization

7.2.3 *Analysis of the Checking Plug-in*

Finally, we show the application of the checking framework on a well-known example, the *Citizien Quartz III wristwatch* Statechart, as introduced by Harel [71]. We remodeled the wristwatch with ArgoUML and the checking was done automatically using the KIEL modeling environment (see Chapter 6). Basically, we adopted the model with the same simplifications made by Harel. Using the UML imposed some restrictions. However, the final model remains the originally modeled behavior. *E. g.*, in the original model some transitions iterate over multiple states which were replaced by conditional constructs accordingly. Furthermore, time events are not envisioned and that is the reason why we had to replace them with simple signal events. Some triggers perform indexing over multiple states which were replaced by constructs using a choice accordingly. Finally, the present interlevel transitions do not possess any further functionality within the model. We avoided to use them from the beginning. The final model contains 120 transitions and

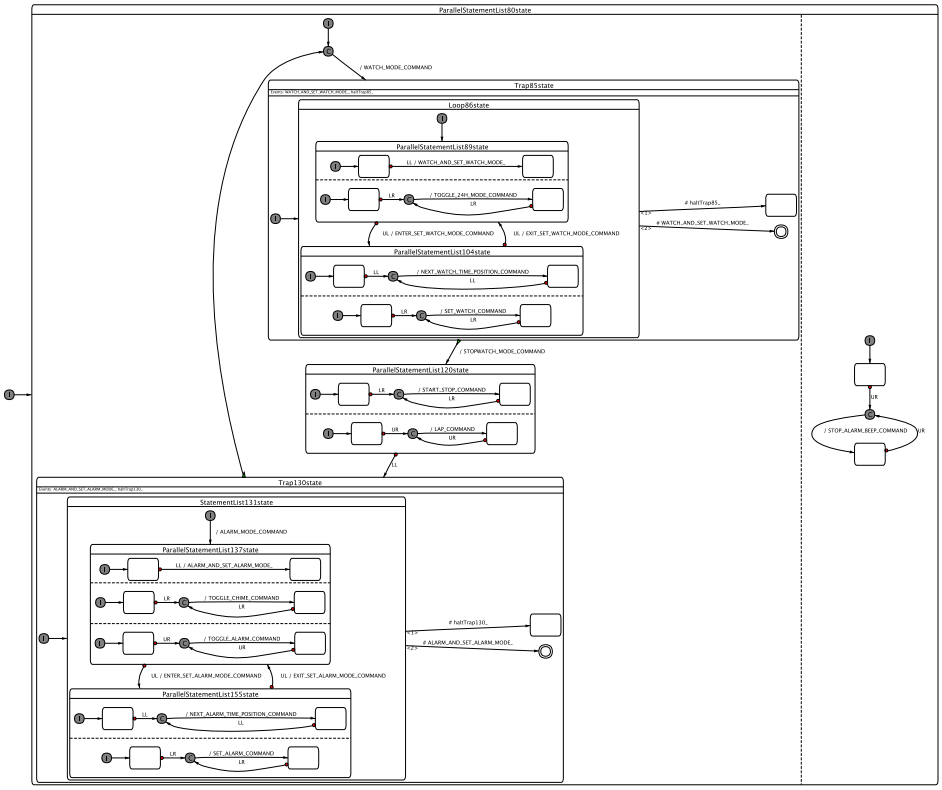


Figure 62: The component *mode selection* of the wristwatch example

108 states. Figure 48 on page 111 presents a screen shot of KIEL as it checks the wristwatch example.

Times for checking robustness

The results from benchmarking are presented in Table 3, where the number of returned hints, the time needed and the number of objects that should be reviewed are presented. The application of the well-formedness rules consumed the least amount of time. Roughly 20 milli-seconds were needed to check those rules on the chart. Except for the check *EqualNames*, the syntactical robustness checks roughly take twice as much time as the well-formedness rules. The check *EqualNames* has a quadratic complexity in the number of states. This is caused by limitations of the OCL. All states have to be compared with the currently handled state. In comparison to the checks dealing with syntactical robustness, except for *EqualNames*, the checks for semantical robustness take about 400 milli-seconds. Here, the check *TransitionOverlap* returns an enormous number of hints compared to the total number of transitions.

Table 3: Experimental results of checking the wristwatch example

CHECKS	HINTS	TIME [MS]
well-formedness checks (total)	0	20
InterlevelTransition	17	14
Connectivity	7	2
EqualNames	33	587
InitialStateCount	7	1
TransitionLabels	6	9
IsolatedStates	1	4
syntactical checks (total)	71	617
Transition Overlap	598	352
Dwelling	0	2
Race Conditions	0	1
semantical checks (total)	598	355
total	669	992

This is due to the fact that almost no transition was designed with an opposing predicate of another outgoing transition.

The application of the framework on the Statechart presented in Figure 24 on page 66 delivered the hint that violations of the rule *Dwelling* are present. Especially novices tend to produce unnecessary large models with needless states, *e. g.*, by splitting trigger and effect into separate transitions. Figure 24 indicates a possible way to curb violation. Since the Statechart is rather small the sets of checks were applied in about 3 milli-seconds.

8

CONCLUSION AND OUTLOOK

Embedded devices are proliferating, and their complexity is ever increasing. Statecharts are a well-established formalism for the description of the reactive behavior of such devices. However, there is evidence that the current use of this formalism is reaching its limits for development of complex systems. The more complex a Statechart model description becomes, the less traceable and manageable it gets. To overcome this, we have presented a methodology to support the easy development and understanding of complex Statecharts. We have implemented these concepts in KIEL, and our experiences with this tool indicate the practicality of this approach.

THE USE OF SECONDARY NOTATION

We are proposing the conscious use of Secondary Notations to improve the readability of complex Statecharts, to make designers more productive and to lower the risk of faults. We have developed specific formatting rules for Statecharts, thus defining a Statechart Normal Form. Furthermore, we have extended this concept into Dynamic Statecharts, which not only takes Statechart topology into account, but also a specific configuration reached in a simulation.

In general, the automatic Statechart layout and the dynamic simulation produced satisfying results. Dynamic Statecharts reduce the size compared to a manual layout. The Dynamic Statecharts allow to view large Statecharts in full without losing the detail. Of course, there comes a point when details become unreadable, and the used examples are complex enough to contain such configurations as well. However, the full-model Dynamic Statechart view provided a very valuable overview of where the "hot spots" of a model were during simulation, which is difficult with the traditional Statechart animation approach.

To assess the efficiency of KIEL's automated layout mechanisms, we have instrumented it to measure computation times. The size especially of large models did not pose any difficulties, the layout computation was always well in the sub-second range. However, in

contrast to academic examples, some complex examples exposed some weakness of the resulting layouts, in particular when long labels were present. Due to transition labels that consist of more than 20 characters and often embrace more than three lines, the element placement was often unsatisfying and the resulting chart was difficult to read. We therefore consider to make the visibility of labels optional. We also observed that especially states that are arranged around a central Statechart element produce long transitions. We suppose that another layout method, *e. g.*, a force directed layout [49], would reduce transition lengths.

TEXTUAL EDITING

A well-established alternative for specifying reactive systems is the textual approach. To benefit from the general ease of handling textual programs (*e. g.*, scripting, preprocessing, and textual revision management), as well as from the intuitiveness of graphical languages, it can be useful to transform one to another. Another motivation for doing so could be the desire to separate structure and layout—akin to, for example, the use of a document processing system (such as LaTeX) that produces a nicely typeset document that adheres to certain formatting guidelines from an ASCII source. Such a system lets the author focus on the contents of the document without worrying about the layout. This benefits especially the development of complex Statecharts with a huge amount of interdependent graphical objects. The developer using a graphical editor has to laboriously and manually manipulate the Statechart, *e. g.*, making room for supplementation of a successor state. In contrast, using a textual language for Statechart specification, the modeler needs only to "write" the new object.

We have presented a description language called KIT that was developed with the intention to describe topological Statechart structures. The KIEL tool combines the ability of easy textual editing and simultaneous viewing of the resulting graphical Statechart model. As another alternative to the classic, low-level WYSIWYG graphical editing paradigm, the graphical model can be modified using high-level editing schemata. This technique employs Statechart production rules that ensure the syntax-consistency throughout the whole editing process. The user feedback on this has been generally very positive, and this has been supported by experimental data.

THE STATECHART SYNTHESIS

We have presented the synthesis of the SSM dialect of Statecharts, using Esterel as input language. The transformation consists of derivation rules, whose application to Esterel statements performs the successive synthesis of SSMs.

We have experimentally validated the transformation and optimizations and have argued their correctness here. We have implemented the transformation in KIEL, and preliminary experience with this tool indicates the practicality of the approach even for large specified systems. A central enabling capability is the automated layout of Statecharts to position the synthesized objects. Experimental results are very promising with regard to the transformation efficiency. Especially, the automatic layout of Statecharts is a promising basis for similar work synthesizing and displaying Statecharts.

The transformation is accompanied by optimizations, which are applicable after the initial transformation, to reduce the complexity of SSMs. The optimization method can be applied not only to synthesized Statecharts; it can also be used for the reduction of manually created models. The optimization method can reduce the number of graphical elements significantly. Hence, it is a valuable contribution to the readability and maintainability of large Statecharts

ERROR PREVENTION

In complex systems it can be very difficult for the modeler to keep all inter-dependent Statechart elements in mind. As a consequence the modeler tends to make errors. To avoid this, we have outlined an approach to make model driven system development with complex Statecharts less error prone. For this intention we have introduced a fundamental Statechart style guide at first. The main goal of the style guide is to incorporate rules that apply to Statecharts in general, *i.e.*, not specific to a single dialect. It is similar to other code checking tools where it still obliges to the developer whether he or she takes care of the delivered hints or not as they do not point at errors. We implemented a flexible checking framework for the general Statechart modeling tool KIEL. We utilize OCL constraints to specify most of the checks as done by other tools.

However, we focused on short response times of the rule checking framework. Hence, compared to Mutz and Huhn's interpretative checking approach [110], we have implemented a transformative approach for the evaluation of OCL statements resulting in a good performance. Furthermore, we implemented some checks in Java directly, because the evaluation of SMT is far beyond the scope of the OCL. As our experimental results show, our checking framework performs the checking of the OCL rules as well as the Java coded rules with acceptable response times. This becomes very important if the style checking is applied to on large systems.

SUMMARY

We have implemented the above mentioned concepts in KIEL, and our experiences with this tool indicate the practicality of this approach for complex Statecharts. A central enabling component is the automated layout of Statecharts, for which we have developed a hierarchical layout engine that uses GraphViz to layout individual charts. Experimental results show that KIEL performs even for complex Statechart models with acceptable response times.

KIEL has also been tested in the classroom, and the feedback regarding the concept of SNF and Dynamic Statecharts has been quite positive. However, we performed more systematic studies on this, also employing expertise from cognitive psychology. The results of the empirical study strongly emphasize the necessity of Secondary Notation for the comprehension of Statecharts. The experiment shows, that Statecharts laid out according to our preferred layout (using GraphViz) scores best in the comprehension part as well in the subjective assessment. The experiment also shows, that developers who use the KIEL macro editor or the KIT editor are faster in modeling Statecharts than those which used a graphical WYSIWYG editor. However, from the tendency of required modeling times using the KIT editor (they get smaller the more experienced a modeler becomes), we consider the KIT editor as most efficient modeling technique for complex Statecharts.

What we see as most promising at this point is to use KIEL's layout capability to construct SNF-compliant Statecharts not from Statecharts already created by a human modeler, but instead to construct these from textual representations. Both, the efficient performing of the KIT editor and the fast generation of an appeal-

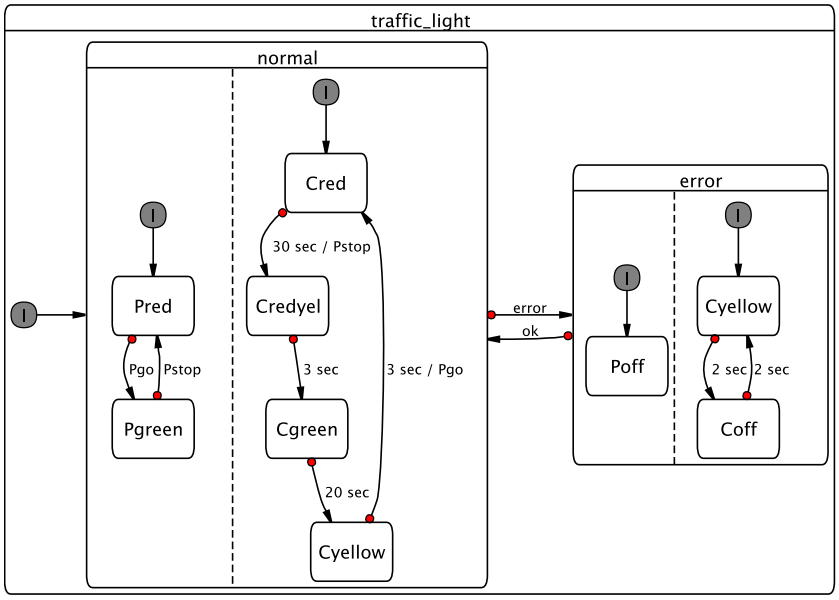
ing Statechart layout using GraphViz, can improve the Statechart development process significantly. It appears valuable to provide the modeler with the possibility to choose to manipulate either the textual or the graphical view in KIEL, where the tool keeps both views automatically and continuously in sync. This results in the best of the textual and the graphical worlds—the efficiency and maintainability of textual entry and the clarity and beauty of visual display.

Regarding ongoing and future work, there are numerous ways in which to extend the layouting, editing, simulation, and customization capabilities of KIEL and the techniques we are exploring. In the future, we intend to experiment further with the simultaneous display of textual and graphical representation of the SUD. *E. g.*, for a better traceability, an indexing mechanism between elements of the textual and the graphical models could be useful. Based on the data of the experimental study mentioned in Section 7.1, we currently investigate the preferences of Statechart developers for certain aesthetic criteria in Statechart layouts and their preference for modeling methods introduced above. We use metrics to determine the aesthetic quality of Statechart layouts and the editing quality of Statechart modeling techniques. Furthermore, we intend to apply the graphical model synthesis from a textual description, in combination with layout and simultaneous display, to data-flow languages such as SCADE/LUSTRE [40, 68] in KIEL's follow-up project—the *KIEL for Eclipse Rich Client Plattform (KIELER)* [50].

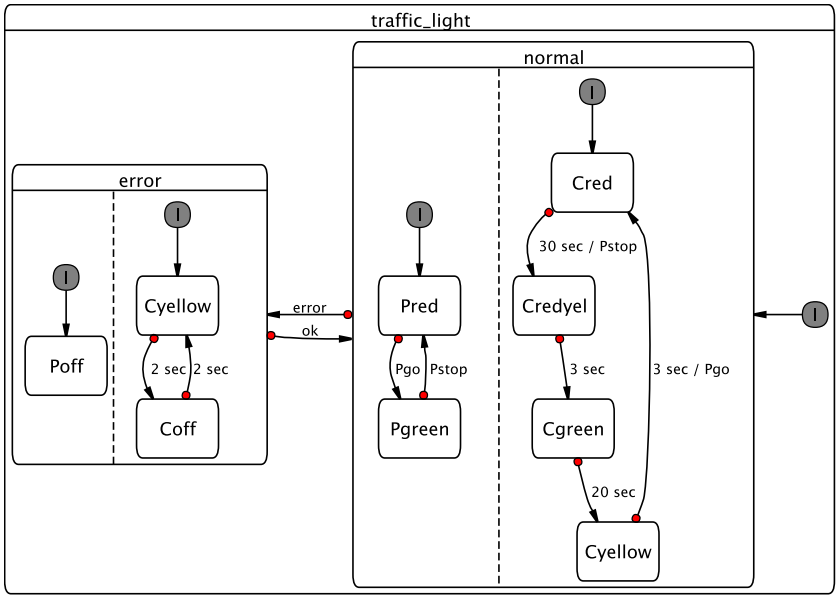
A

LAYOUT EXAMPLES FROM KIEL

In this chapter, some Statecharts drawn by KIEL layouter will be presented. The figures demonstrate, how the KIEL layouter can be adjusted. As an example model we use the *traffic light controller* introduced in Section 4.1 on page 75, the *bintree*, and the *quadtree* introduced in Section 7.2.1 on page 125.

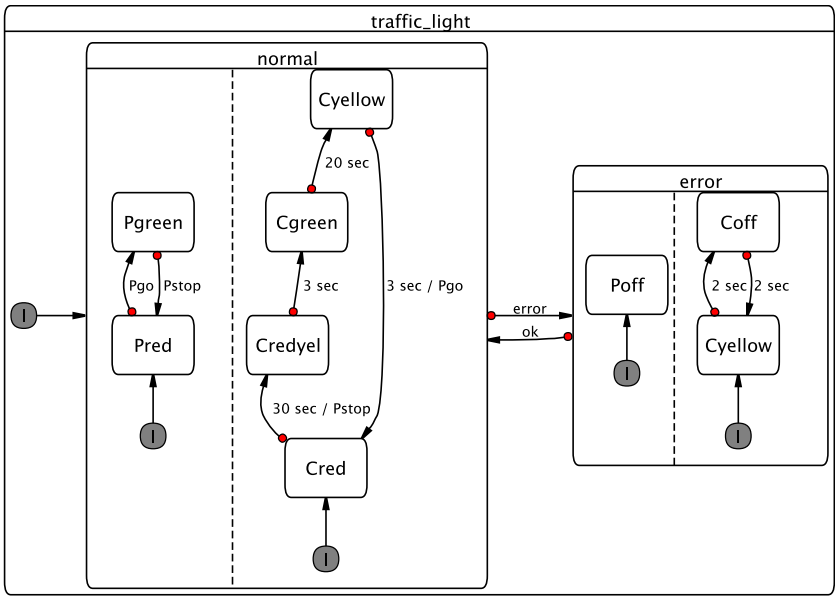


(a) Alternating reading directions

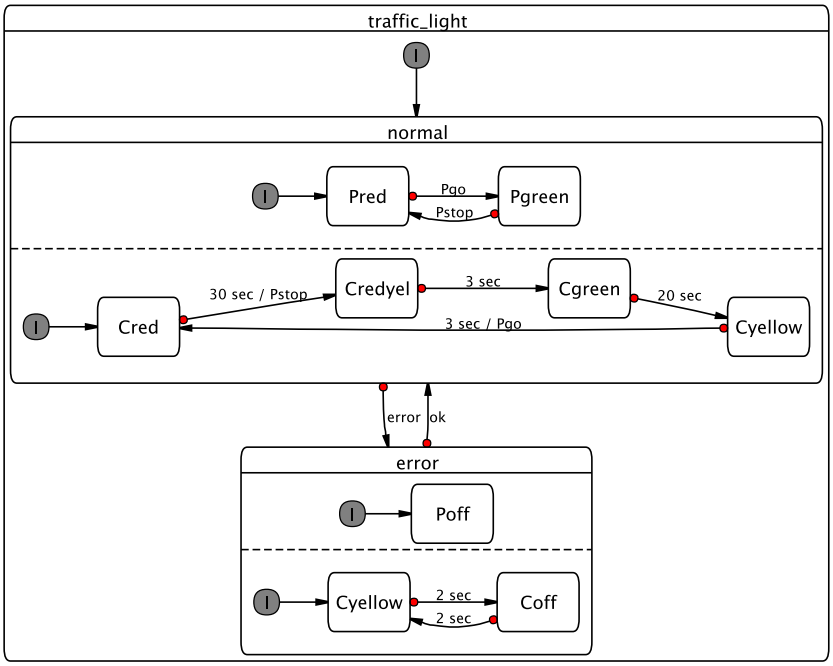


(b) Reversed reading direction

Figure 63: Dot layout with different reading directions

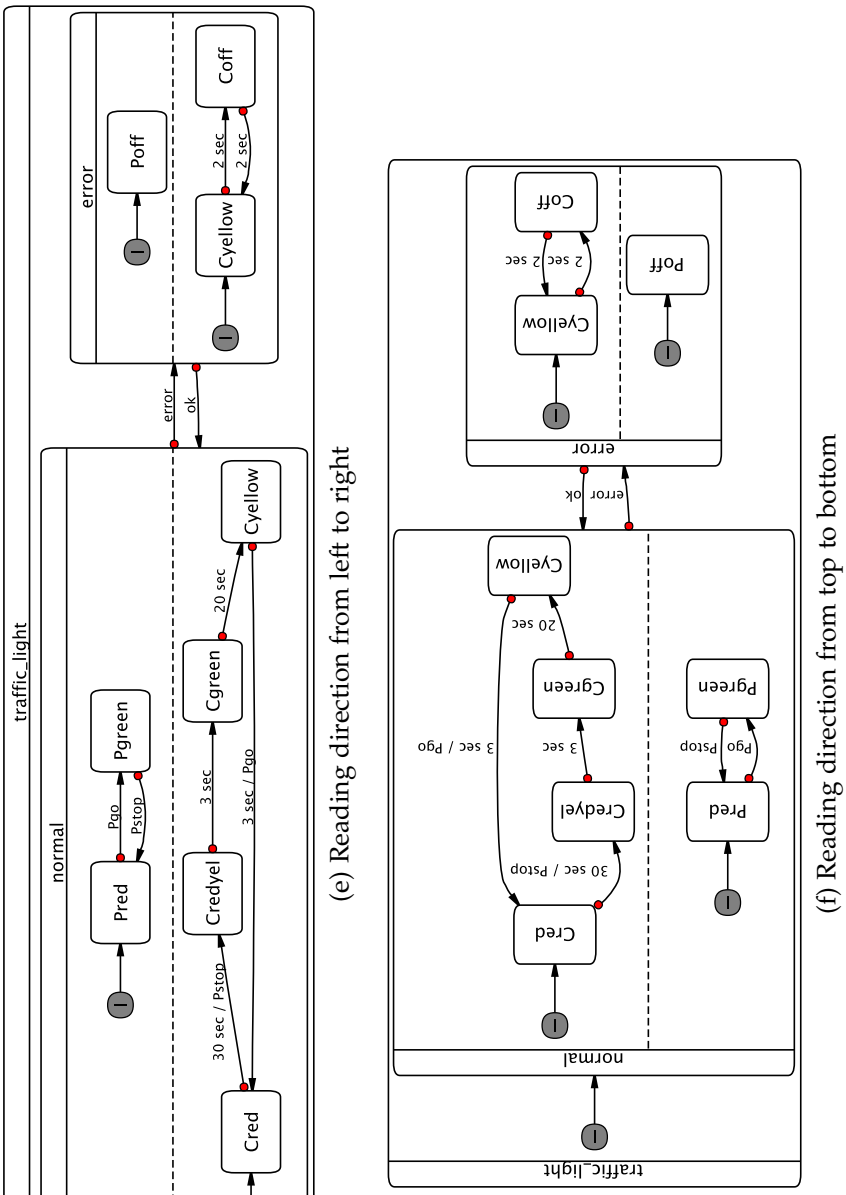


(c) Reading direction from bottom to top



(d) Alternating reading directions and reading begins on top

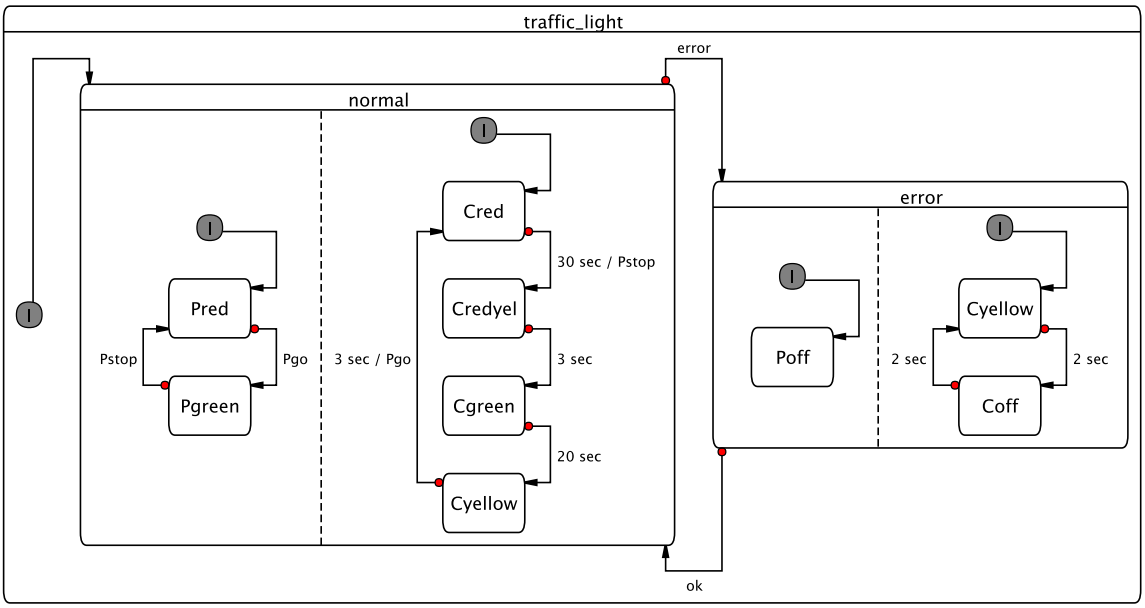
Figure 63: Dot layout with different reading directions



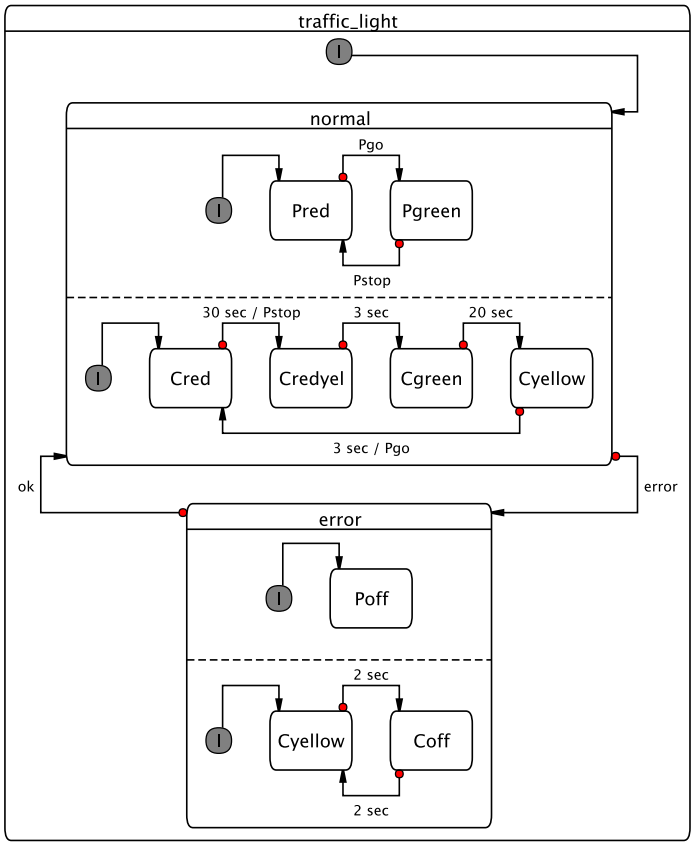
(e) Reading direction from left to right

(f) Reading direction from top to bottom

Figure 63: Dot layout with different reading directions

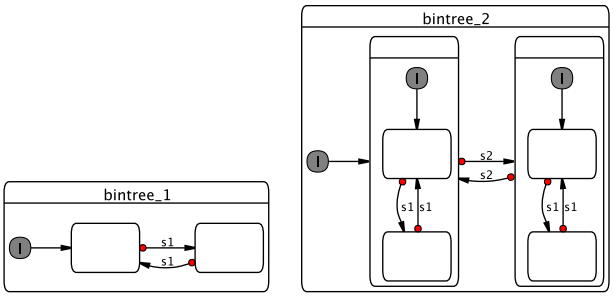


(a) Alternating reading directions

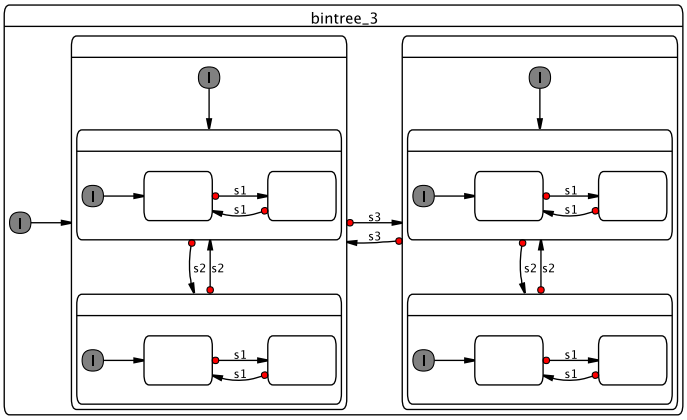


(b) Alternating reading directions and reading begins on top

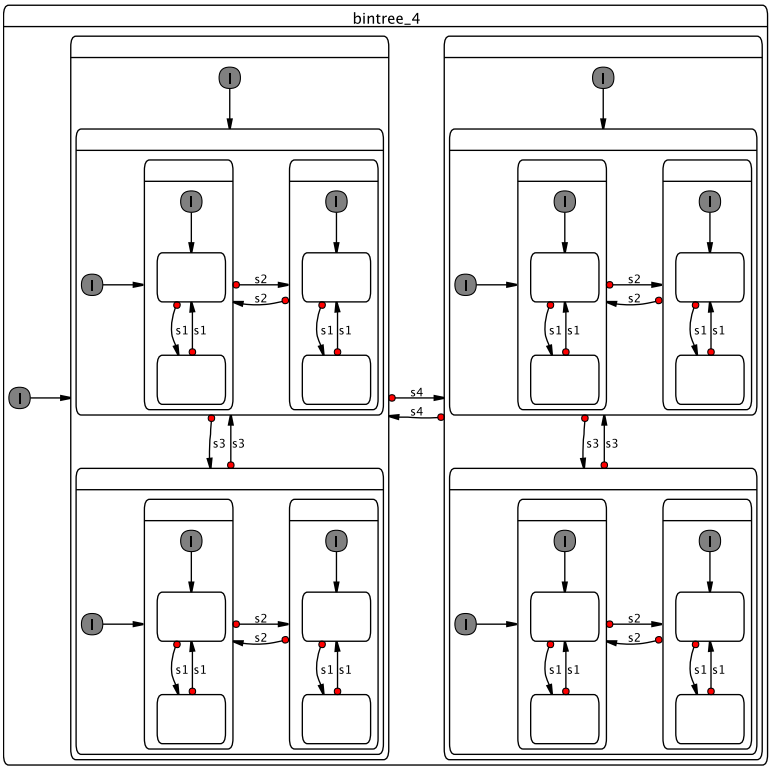
Figure 64: Linear layer layout with different reading directions



(a) bintree-1 (b) bintree-2

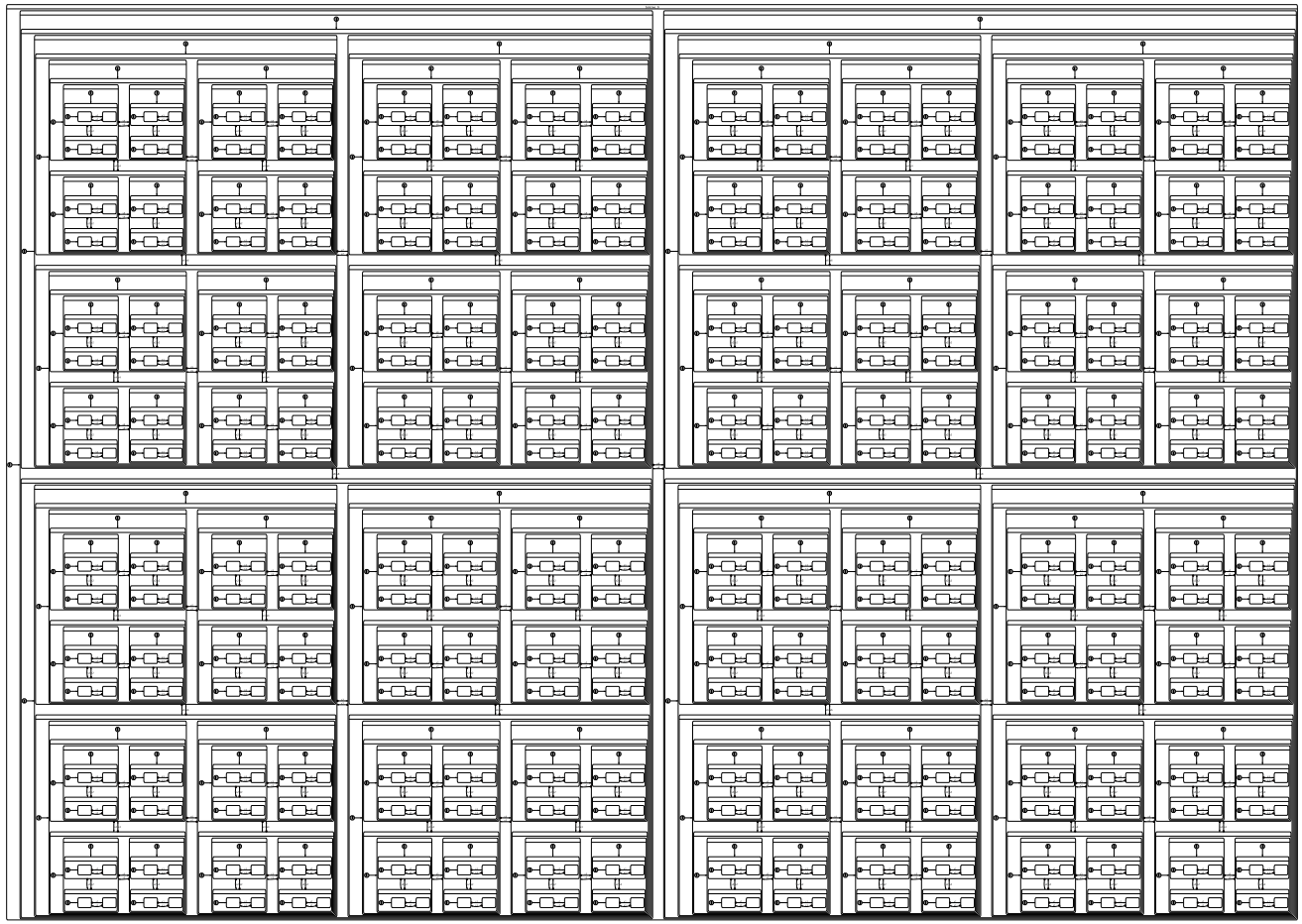


(c) bintree-3



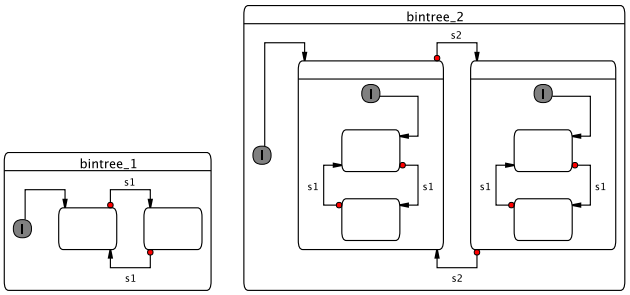
(d) bintree-4

Figure 65: The generic Statechart model *bintree* laid out according to the Alternating Dot Layout (ADL)

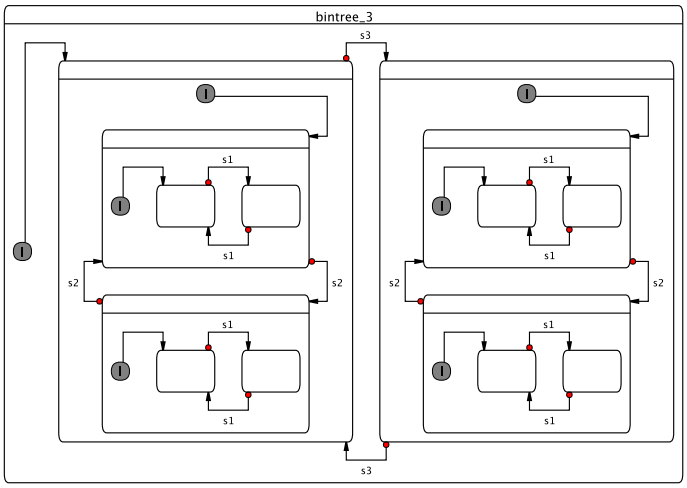


(e) bintree-9

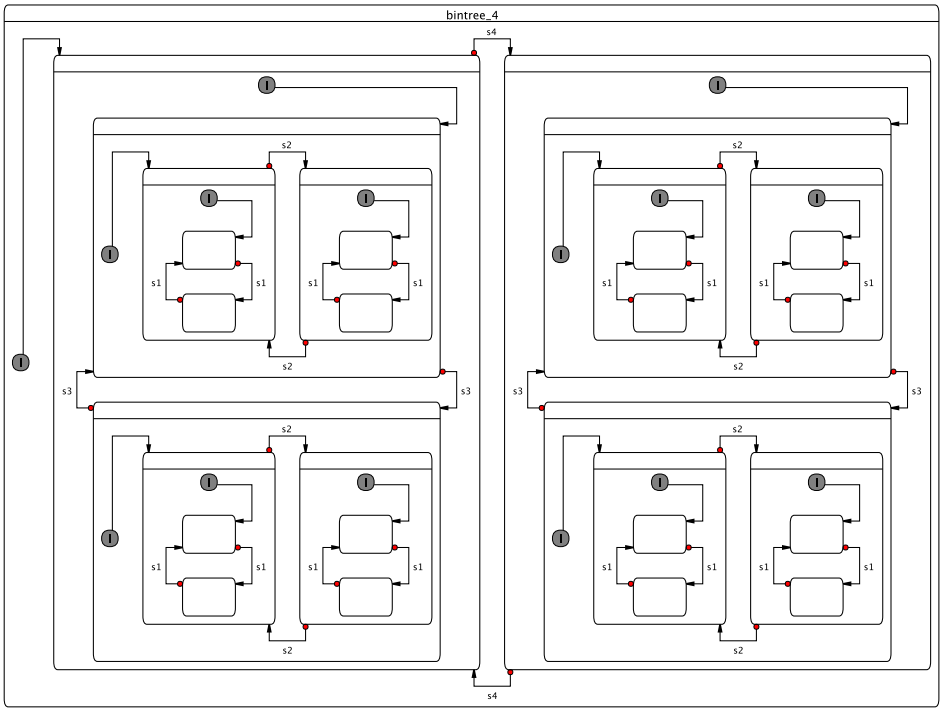
Figure 65: The generic Statechart model *bintree* laid out according to the Alternating Dot Layout (ADL) (continued)



(f) bintree-1 (g) bintree-2

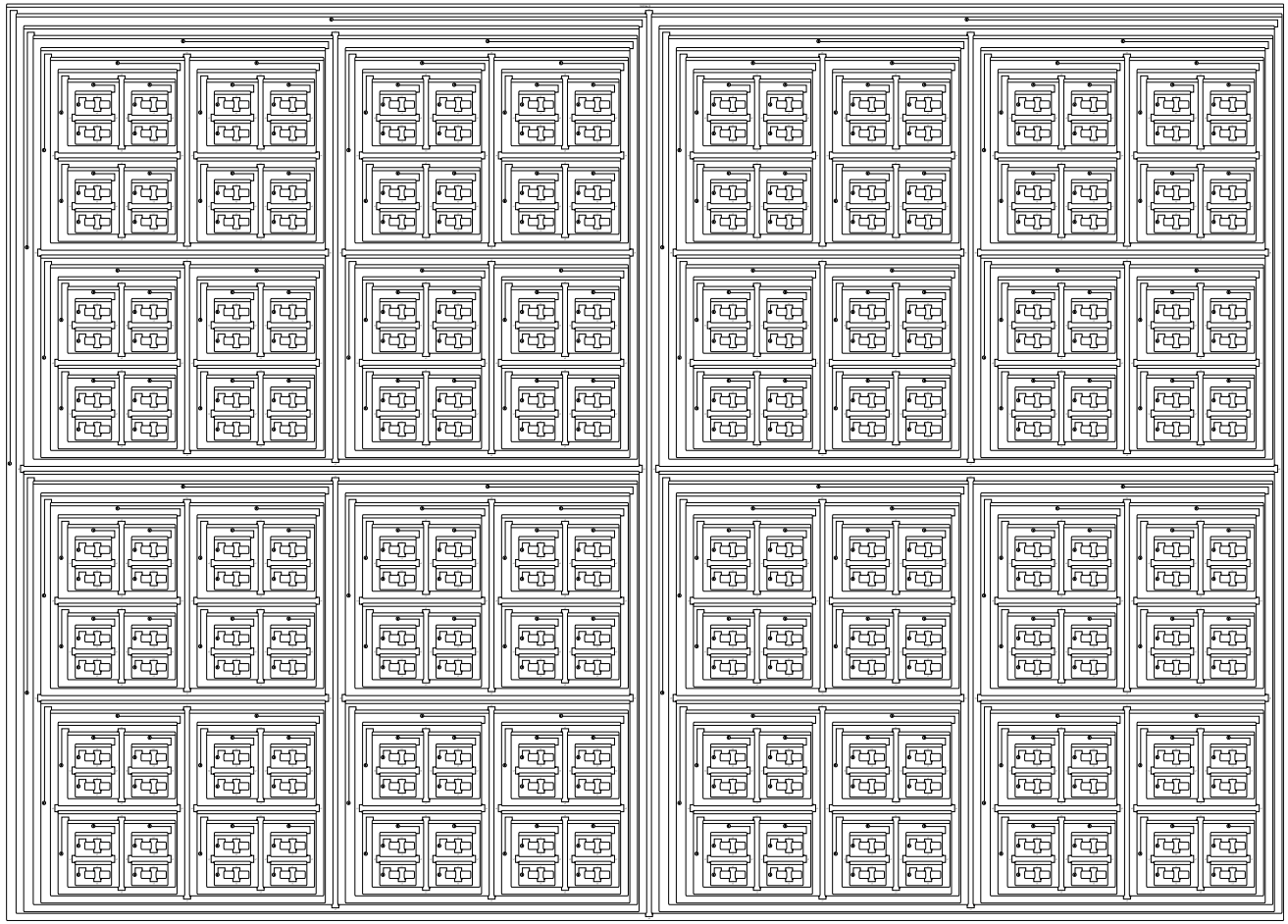


(h) bintree-3



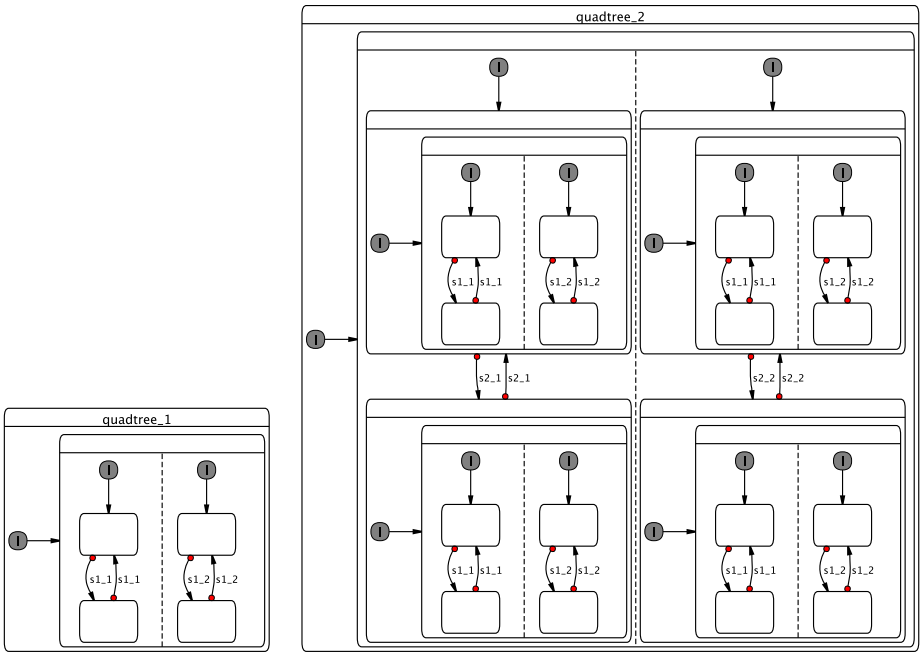
(i) bintree-4

Figure 65: The generic Statechart model *bintree* laid out according to the Linear Layer Layout (LLL)

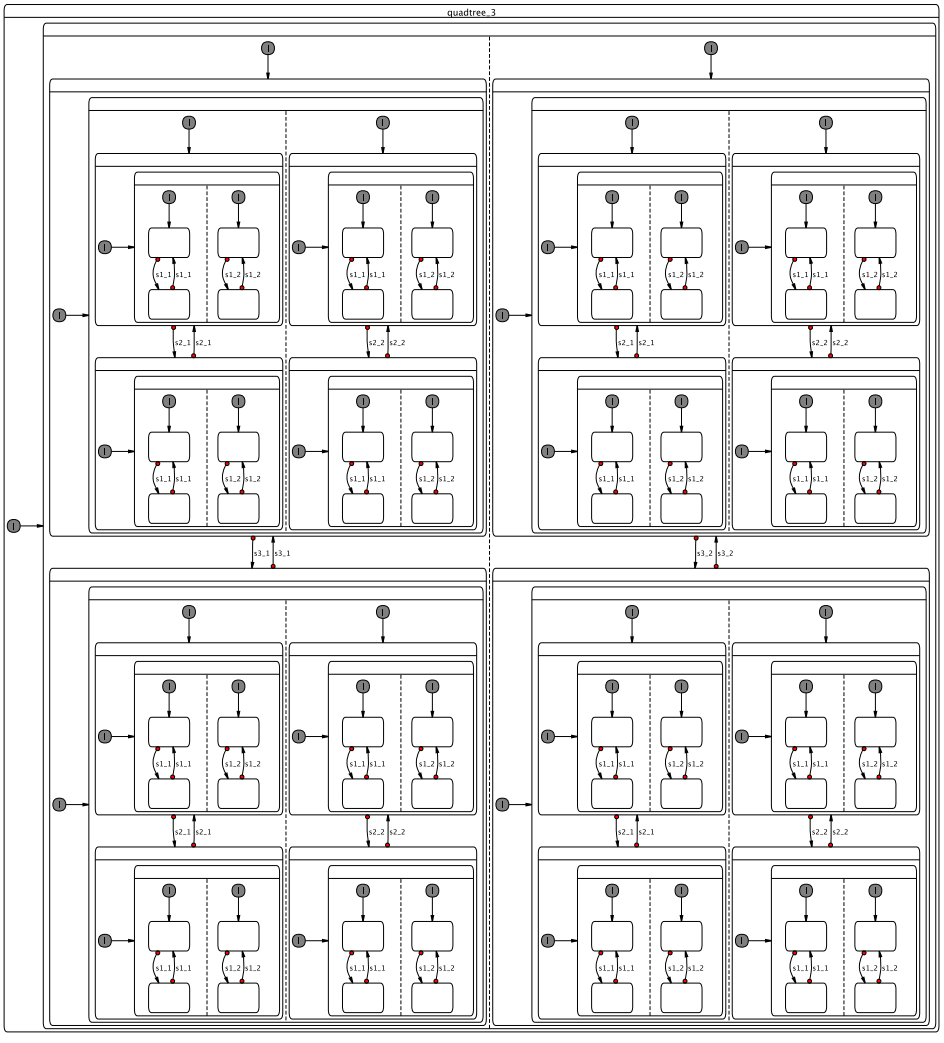


(j) bintree-9

Figure 65: The generic Statechart model *bintree* laid out according to the Linear Layer Layout (LLL) (continued)

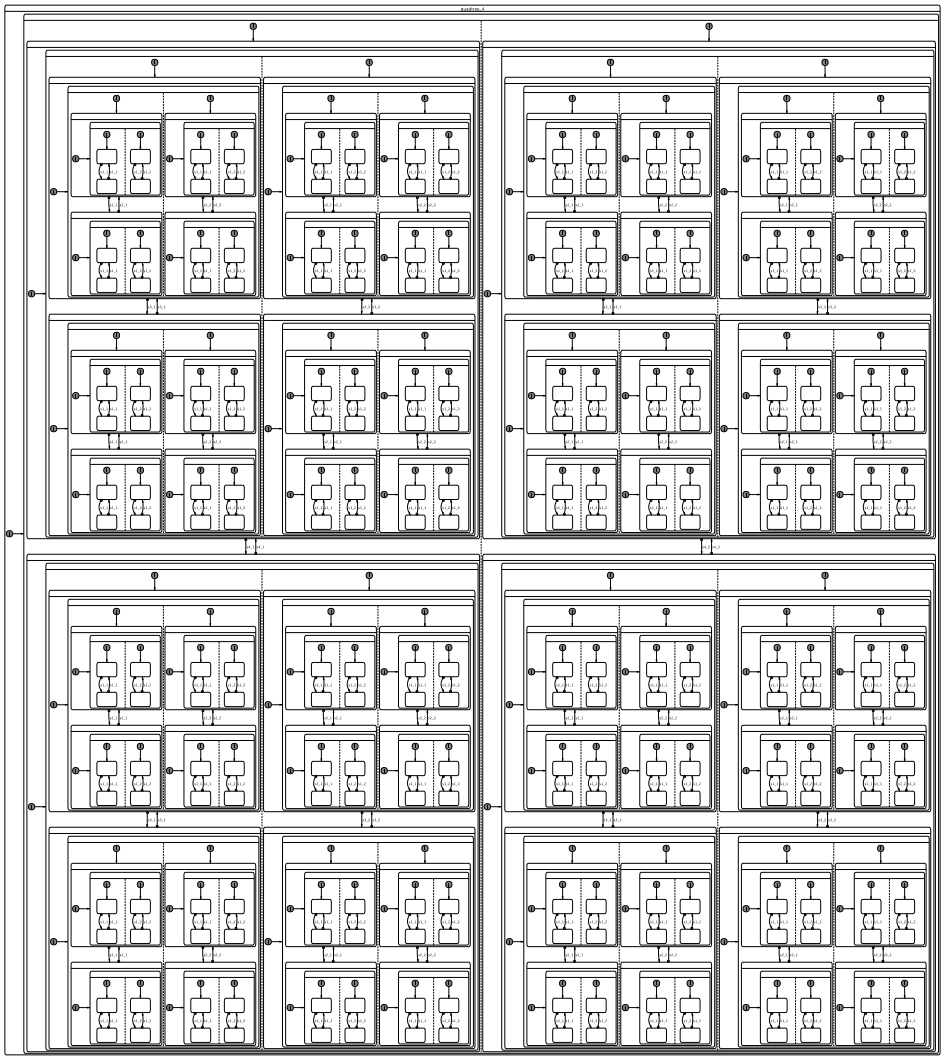


(a) quadtree-1 (b) quadtree-2



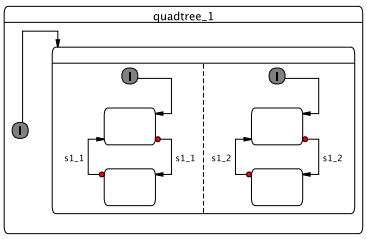
(c) quadtree-3

Figure 66: The generic Statechart model *quadtree* laid out according to the Alternating Dot Layout (ADL)

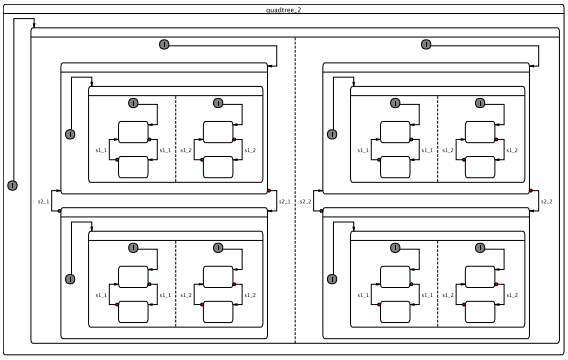


(d) quadtree-4

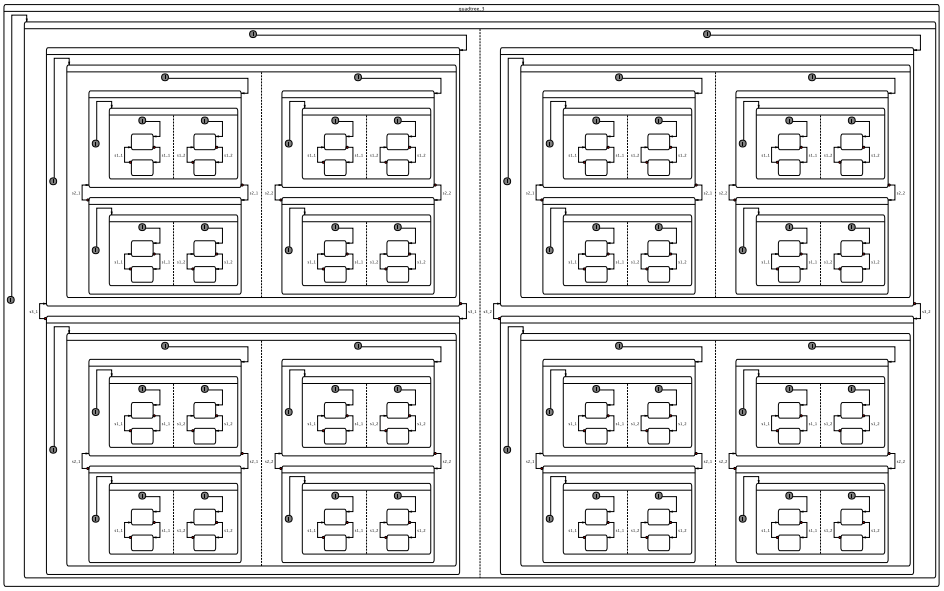
Figure 66: The generic Statechart model *quadtree* laid out according to the Alternating Dot Layout (ADL) (continued)



(e) quadtree-1

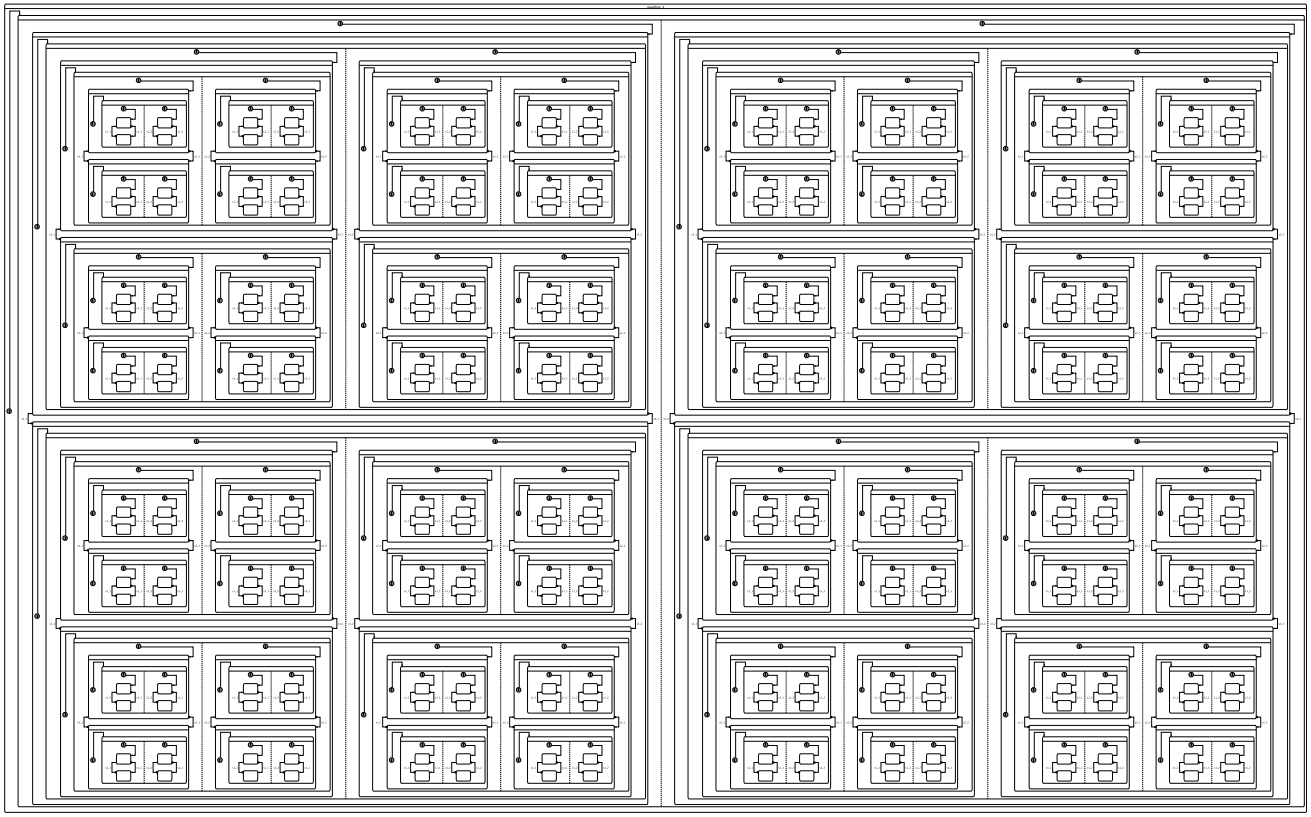


(f) quadtree-2



(g) quadtree-3

Figure 66: The generic Statechart model *quadtree* laid out according to the Linear Layer Layout (LLL)



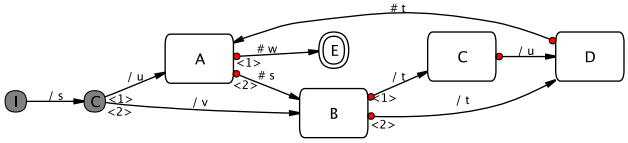
(h) quadtree-4

Figure 66: The generic Statechart model *quadtree* laid out according to the Linear Layer Layout (LLL) (continued)

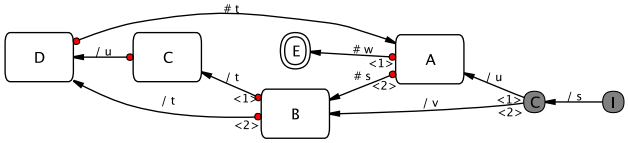
B

STATECHART LAYOUTS FROM EMPIRICAL STUDY

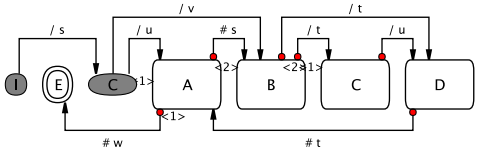
In this chapter we depict the Statechart layouts which have been presented to the subjects of the empirical study (see Section 7.1 on page 115). We here depict only 15 of 75 Statecharts of different complexity.



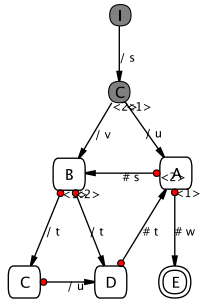
(a) Alternating Dot Layout (ADL)



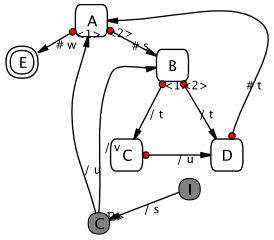
(b) Alternating Dot Backwards Layout (ADBL)



(c) Linear Layer Layout (LLL)

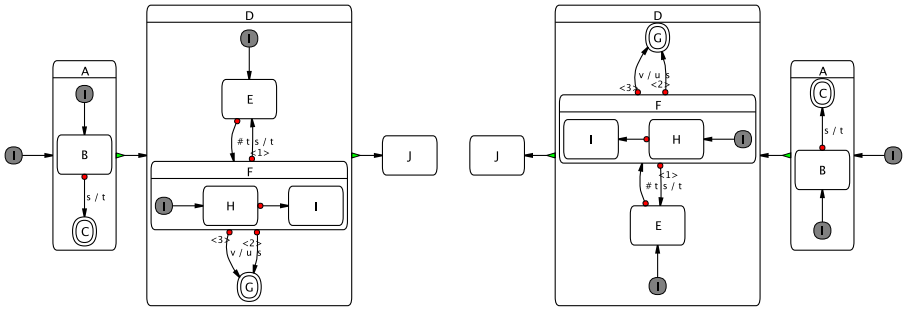


(d) Alternating Linear Layout (ALL)



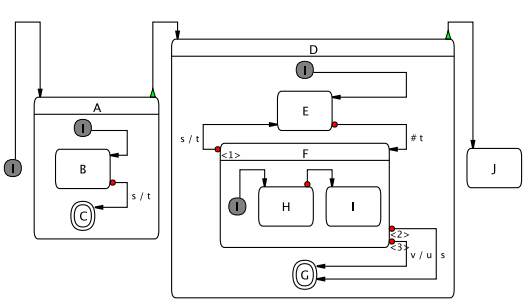
(e) Arbitrary Layout (AL)

Figure 67: Layouts of a Statechart model with simple states

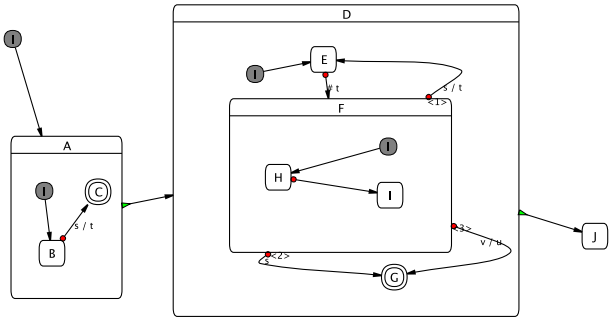


(f) Alternating Dot Layout (ADL)

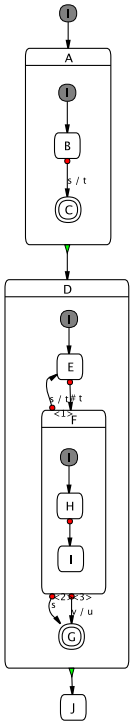
(g) Alternating Dot Backwards Layout (ADBL)



(h) Alternating Linear Layout (ALL)

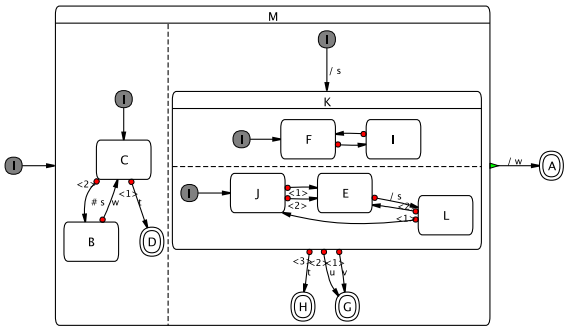


(j) Arbitrary Layout (AL)

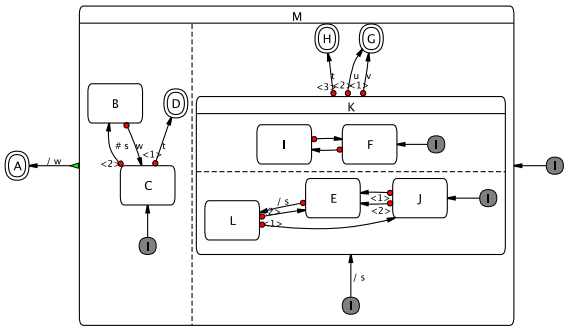


(i) Linear Layer Layout (LLL)

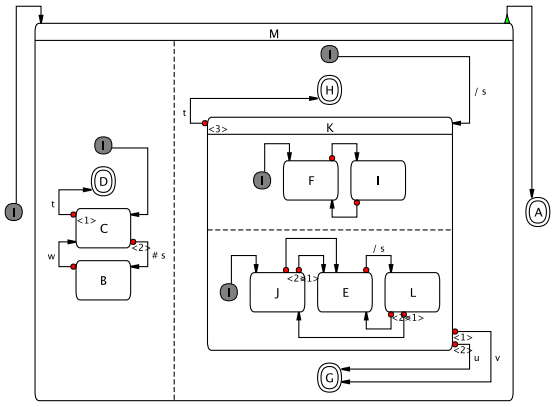
Figure 67: Layouts of a Statechart model with hierarchical states



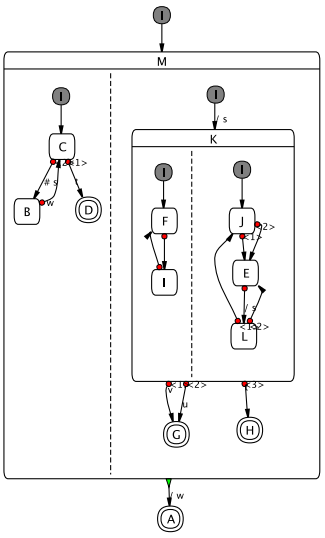
(k) Alternating Dot Layout (ADL)



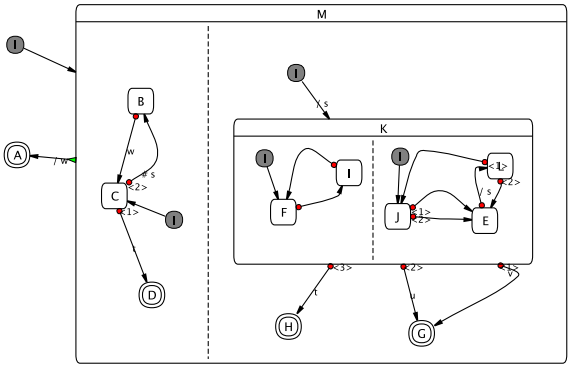
(l) Alternating Dot Backwards Layout (ADBL)



(m) Alternating Linear Layout (ALL)



(n) Linear Layer Layout (LLL)



(o) Arbitrary Layout (AL)

Figure 67: Layouts of a Statechart model with parallel states

C

WORKING DOCUMENTS FROM EMPIRICAL STUDY

In the following we depict documents which have been presented to subjects of the empirical study (see Section 7.1 on page 115). Section C.1 depicts one of the randomized data documents. It consists of thee parts: (1) In the first part, there are pairs of Statechart layouts depicted which had to be compared, (2) the second part contains Statecharts which had to be understood, and (3) the third part contains instructions to create and modify a Statechart using different modeling tools. In Section C.2 we present the document which had to be completed by subjects and in Section C.3 we present reference cards of the used modeling tools.

C.1 THE DATA DOCUMENT

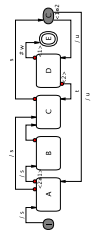
Untersuchung des Layouts und der Modellierung von Statecharts (II)

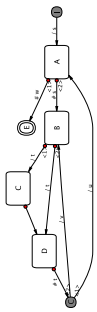
– Daten-Blattsammlung –
(Gruppe 1, Nummer 1)

1 Vergleichen verschiedener Statechart-Layouts

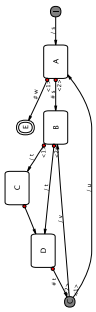
1.1 Statecharts mit einfachen Zuständen

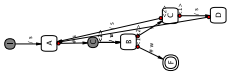
Vergleichen 1



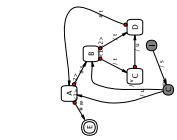


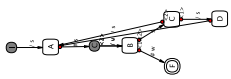
Vergleichen 2



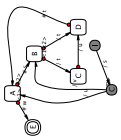


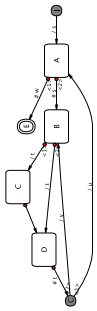
Vergleichen 3



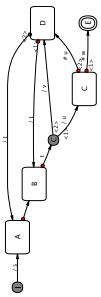


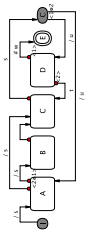
Vergleichen 4



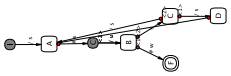


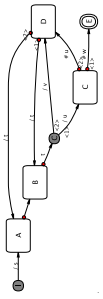
Vergleichen 5



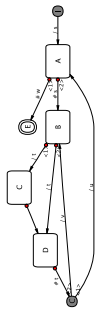


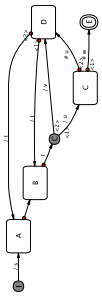
Vergleichen 6



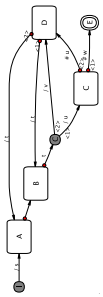


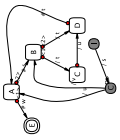
Vergleichen 7



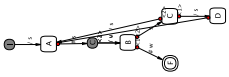


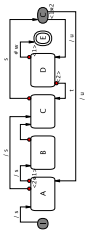
Vergleichen 8



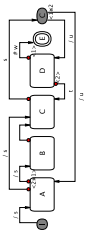


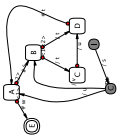
Vergleichen 9





Vergleichen 10

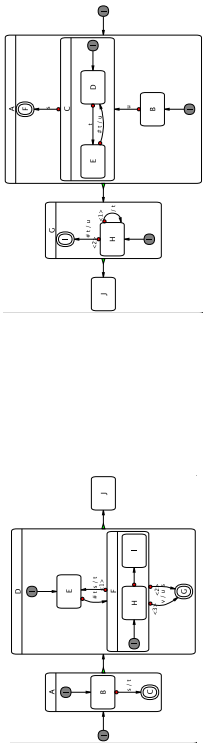




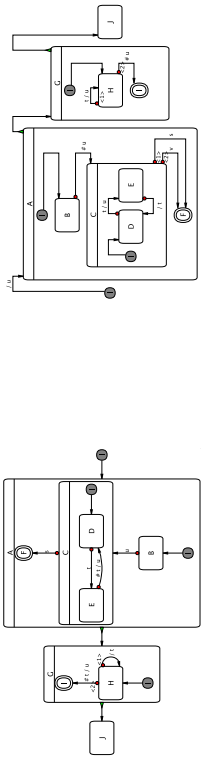
1.2 Statecharts mit hierarchischen Zuständen

13

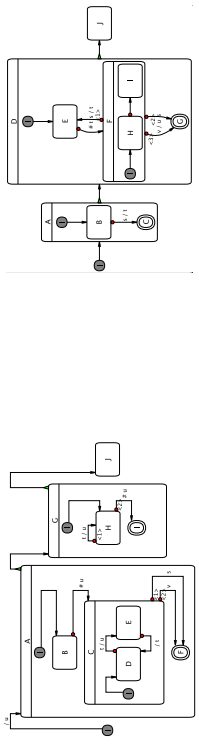
Vergleichen 11



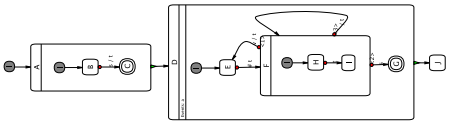
Vergleichen 12

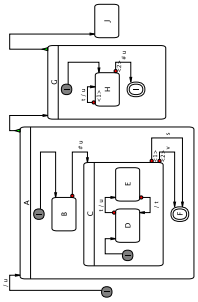


Vergleichen 13

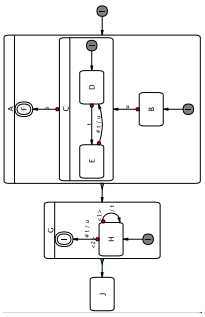


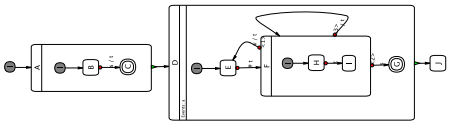
Vergleichen 14



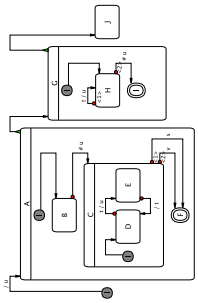


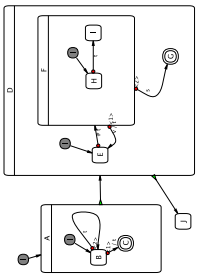
Vergleichen 15



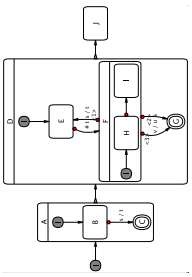


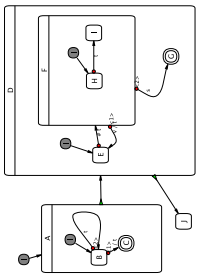
Vergleichen 16



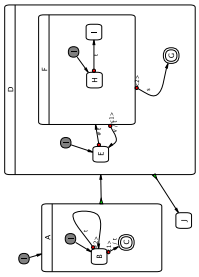


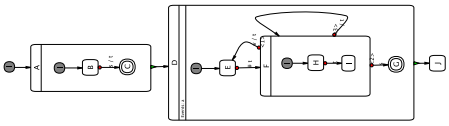
Vergleichen 17



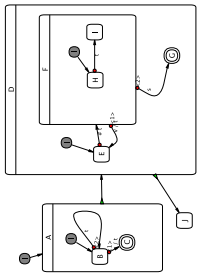


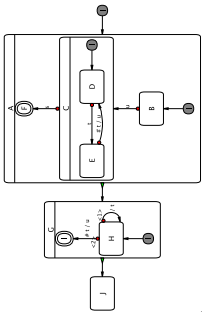
Vergleichen 18



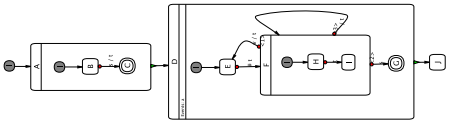


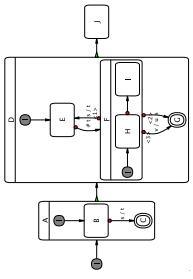
Vergleichen 19





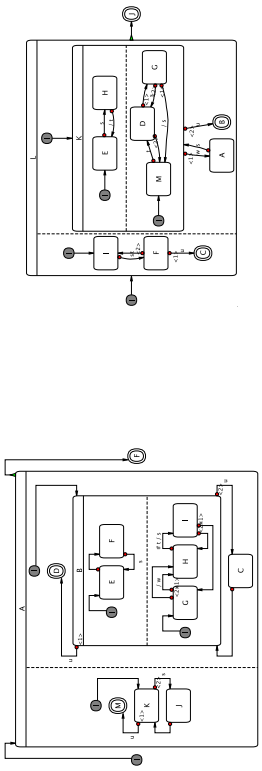
Vergleichen 20



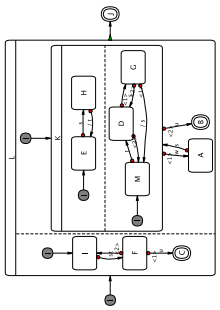


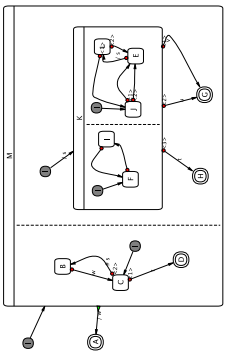
1.3 Statecharts mit parallelen Zuständen

Vergleichen 21

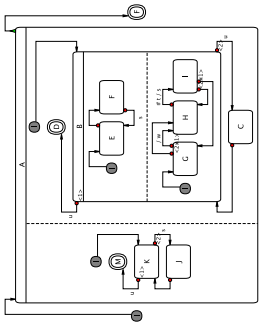


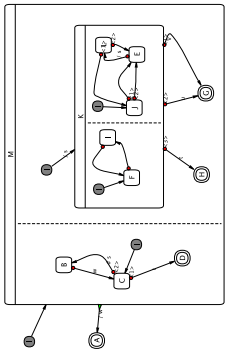
Vergleichen 22



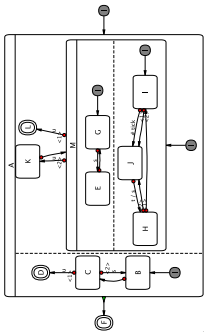


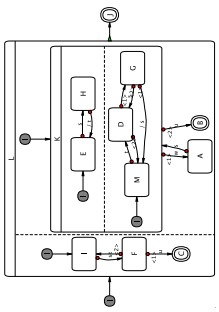
Vergleichen 23



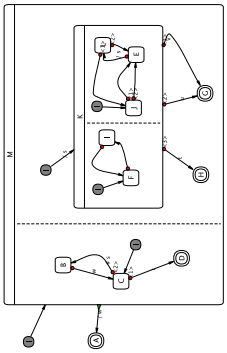


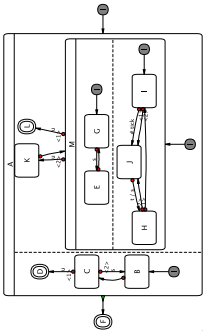
Vergleichen 24



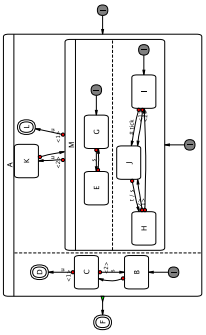


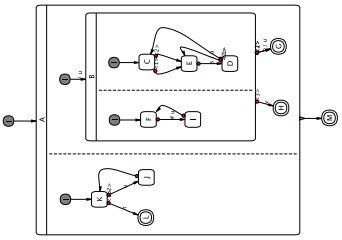
Vergleichen 25



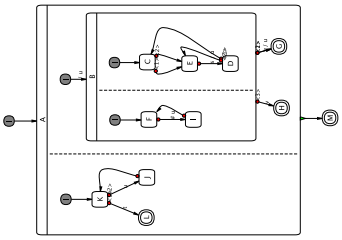


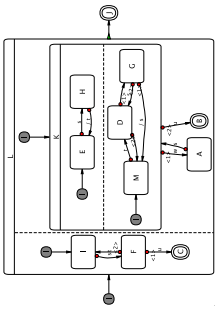
Vergleichen 26



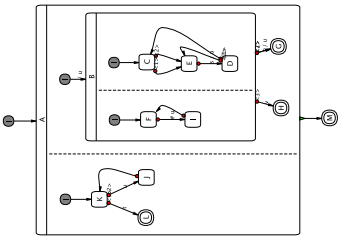


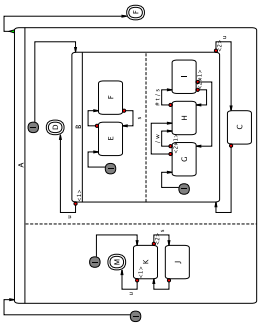
Vergleichen 27



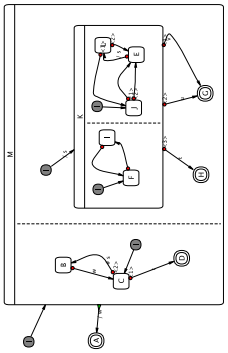


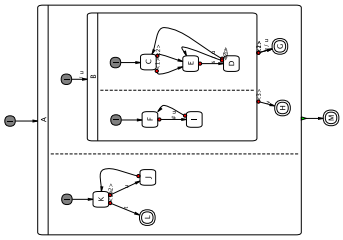
Vergleichen 28



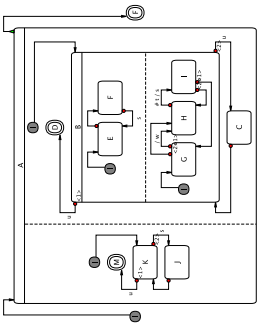


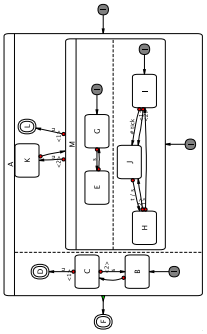
Vergleichen 29





Vergleichen 30

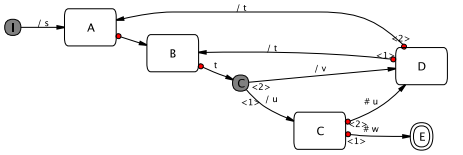


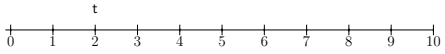


Verstehen 1

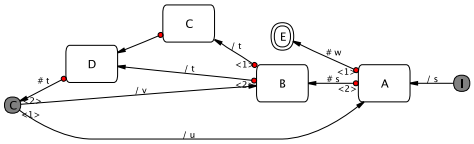
2 Verstehen von Statecharts

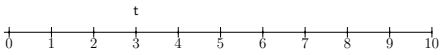
2.1 Statecharts mit einfachen Zuständen



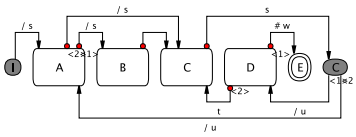


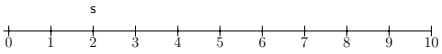
Verstehen 2



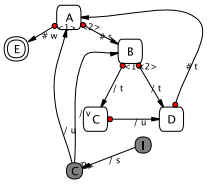


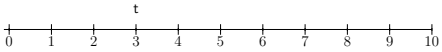
Verstehen 3



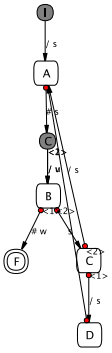


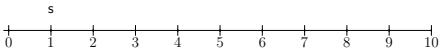
Verstehen 4





Verstehen 5

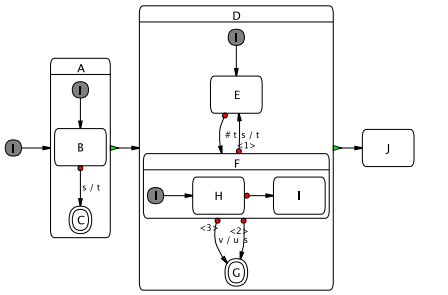


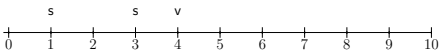


2.2 Statecharts mit hierarchischen Zuständen

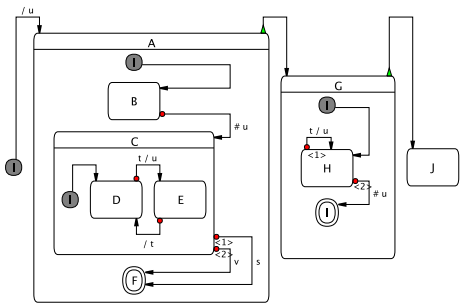
41

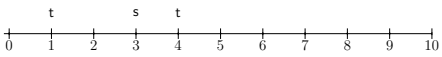
Verstehen 6



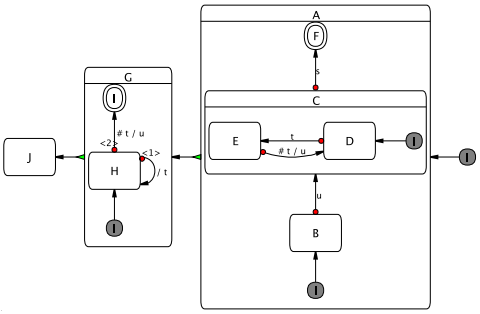


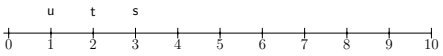
Verstehen 7



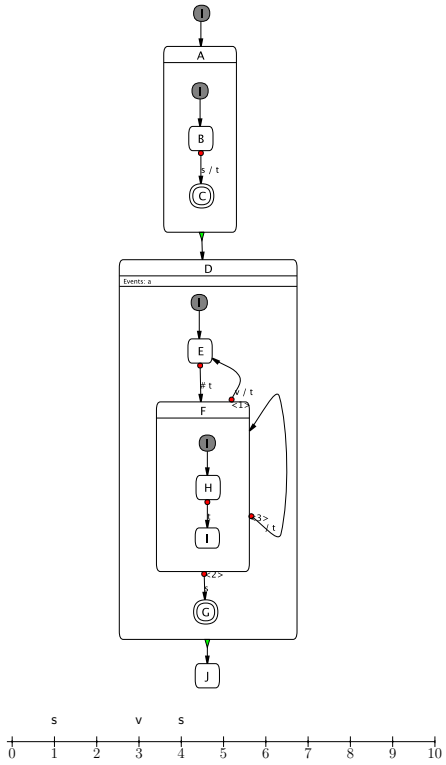


Verstehen 8

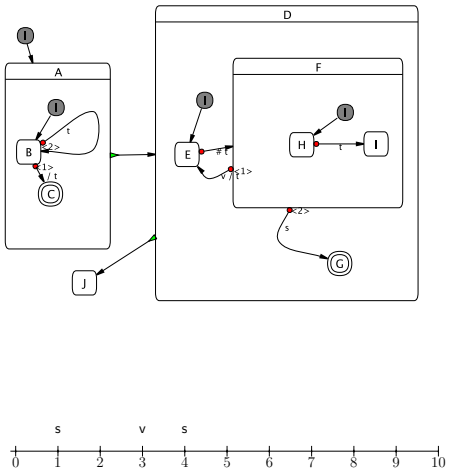




Verstehen 9

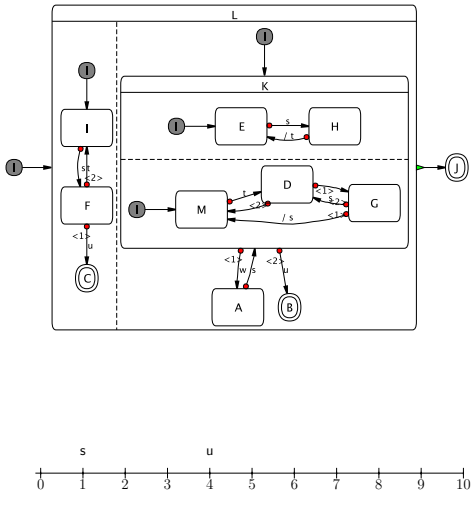


Verstehen 10

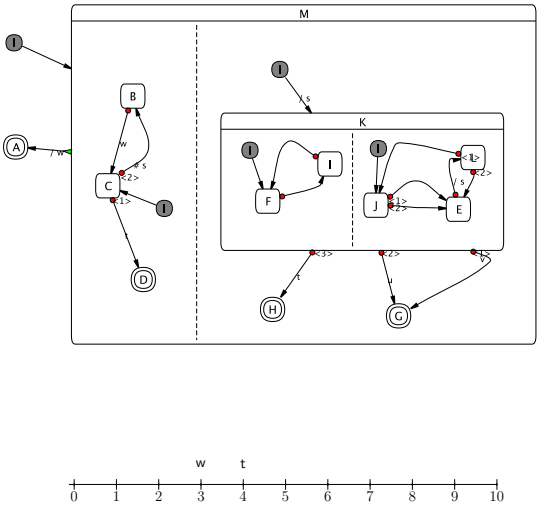


2.3 Statecharts mit parallelen Zuständen

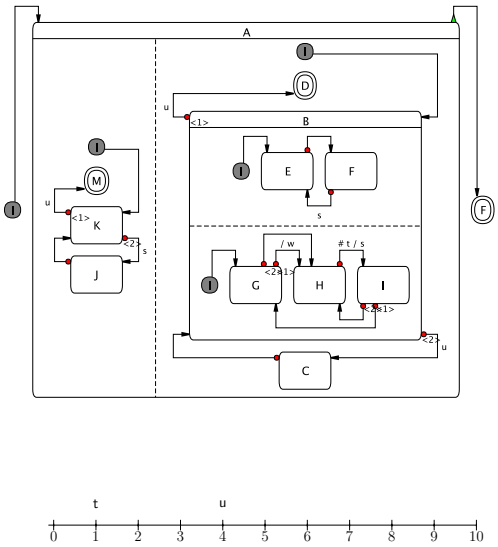
Verstehen 11



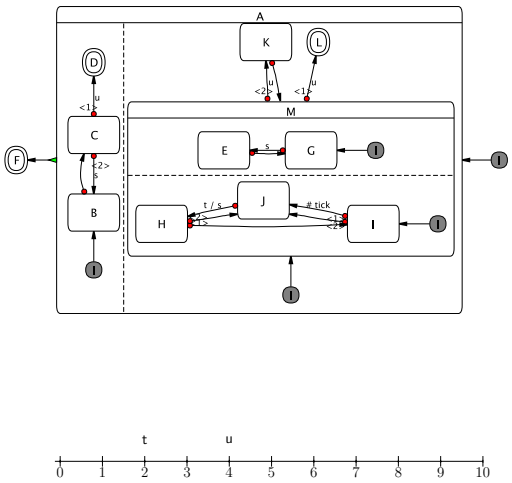
Verstehen 12



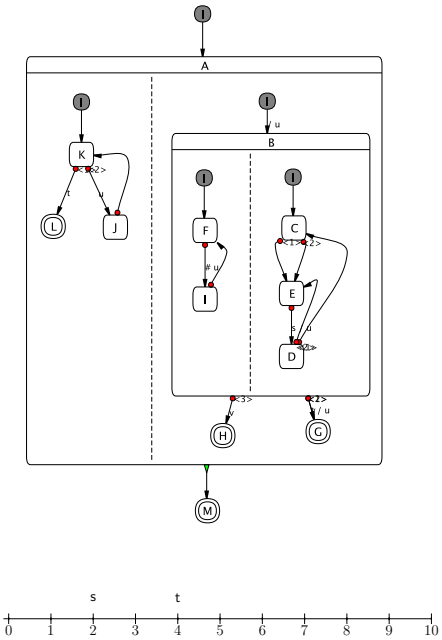
Verstehen 13



Verstehen 14



Verstehen 15



3 Modellieren von Statecharts

Neuerstellen eines Statecharts

Zu modellierendes Statechart:

Anmerkung I: Bevor Sie sich mit der Umsetzung des beschriebenen Statecharts in einem der drei Werkzeuge befassen, analysieren Sie die u. g. Beschreibung soweit, dass Sie diese gut verstanden haben. *Die Zeit für diese Analyse wird separat gemessen.*

Anmerkung II: Für ein korrektes Modell müssen die Signale, die das Modell von außen beeinflussen korrekt angelegt sein. Der Einfachheit halber verzichten wir bei diesem Experiment darauf, das Signal-Interface zu deklarieren. Tragen Sie deshalb einfach nur die Signale an den Transitionen ab.

$$
\begin{aligned}
S &= \{R_0, R, S, X_0, X, Y, Z\} \\
\Sigma &= \{r, x, y, z\} \\
\Gamma &: \{R_0 \rightarrow R, X_0 \rightarrow X, X \overset{x}{\bullet\!\!\rightarrow} Z, Z \overset{z}{\bullet\!\!\rightarrow} Y, Y \overset{y}{\bullet\!\!\rightarrow} Z, Y \overset{y}{\bullet\!\!\rightarrow} X, R \overset{r}{\bullet\!\!\rightarrow} S, \} \\
t &: \{R_0 \rightarrow \text{initial}, R \rightarrow \text{normal}, S \rightarrow \text{normal}, X_0 \rightarrow \text{initial}, X \rightarrow \text{normal}, \\
&\quad Y \rightarrow \text{normal}, Z \rightarrow \text{normal}\} \\
\leq &: \{X_0 \leq R, X \leq R, Y \leq R, Z \leq R\}
\end{aligned}
$$

Werkzeug-Reihenfolge:

1. Graphischer Statechart-Editor
2. Strukturbasierter Statechart-Editor
3. Textueller Statechart-Editor

Verändern eines Statecharts

Ergänzendes Statechart:

Anmerkung I: Bevor Sie sich mit der Umsetzung des beschriebenen Statecharts in einem der drei Werkzeuge befassen, analysieren Sie die u. g. Beschreibung soweit, dass Sie diese gut verstanden haben. *Die Zeit für diese Analyse wird separat gemessen.*

Anmerkung II: Hier wird nur der Teil des Statechart der gerade vollzogenen Übung dargestellt, der das von dort bekannte Modell verändert.

$$
\begin{aligned}
S' &= S \cup \{T_0, T, U\} \\
\Sigma' &= \Sigma \\
\Gamma' &: (\Gamma - \{Y \overset{y}{\bullet\!\!\rightarrow} Z\}) \cup \{Z \overset{y}{\bullet\!\!\rightarrow} Y, T_0 \rightarrow T, T \bullet\!\!\rightarrow U\} \\
t' &: t \cup \{T_0 \rightarrow \text{initial}, T \rightarrow \text{normal}, U \rightarrow \text{normal}\} \\
\leq' &: \leq \cup \{T_0 \leq X, T \leq X, U \leq X\}
\end{aligned}
$$

Unveränderte Werkzeug-Reihenfolge:

1. Graphischer Statechart-Editor
2. Strukturbasierter Statechart-Editor
3. Textueller Statechart-Editor

C.2 THE ANSWER TEMPLATE DOCUMENT

Untersuchung des Layouts und der Modellierung von Statecharts (II)

– Testbogen –

Name: _____

E-Mail: _____

Ich erkläre mich damit einverstanden, dass zum Zwecke der nachträglichen Analyse dieses Experimentes der Ablauf in Bild und Ton festgehalten wird.

Unterschrift: _____

Vorwort

Die folgenden Aufgaben wurden entsprechend dem zu erwartenden derzeitigen Kenntnisstand der Vorlesung „Modellbasierter Entwurf und Verteilte Echtzeitsysteme“ entworfen. Lesen Sie sich vor Beginn der Experimente die Versuchsanordnung genau durch und fragen Sie den Experimentleiter bei Unklarheiten möglichst noch vor dem Experiment. Bearbeiten Sie dann die Aufgaben gewissenhaft und möglichst zügig. Zur Experimentanalyse ist es wichtig, die Gedankengänge bei der Lösung der Aufgaben zu kennen. Sprechen Sie deshalb Ihre Überlegungen laut aus.

Danke für Ihre Teilnahme am Experiment
und gutes Gelingen bei der Lösung der Aufgaben!

1 Vergleichen verschiedener Statechart-Layouts

Im Folgenden werden Ihnen jeweils zwei Statecharts mit unterschiedlichem Aussehen vorgelegt. Entscheiden Sie jeweils spontan (nicht länger als 10 Sekunden), bei welchem sich der Statechart-Modellierer mehr Mühe gegeben hat, ein gut lesbares und verständliches Statechart zu erstellen. Sagen Sie zu jedem Paar die Nummer mit an. Stellen Sie jeweils mündlich die Unterschiede der Statecharts heraus und diskutieren Sie Vor- und Nachteile der Layouts. Tragen Sie dann im entsprechenden Bewertungsschema das Ergebnis Ihrer Einschätzung ein (bitte wenden).

1.1 Statecharts mit einfachen Zuständen

1.	○		○		○		○		○
	Layout I				gleich gut				Layout II
2.	○		○		○		○		○
	Layout I				gleich gut				Layout II
3.	○		○		○		○		○
	Layout I				gleich gut				Layout II
4.	○		○		○		○		○
	Layout I				gleich gut				Layout II
5.	○		○		○		○		○
	Layout I				gleich gut				Layout II
6.	○		○		○		○		○
	Layout I				gleich gut				Layout II
7.	○		○		○		○		○
	Layout I				gleich gut				Layout II
8.	○		○		○		○		○
	Layout I				gleich gut				Layout II
9.	○		○		○		○		○
	Layout I				gleich gut				Layout II
10.	○		○		○		○		○
	Layout I				gleich gut				Layout II

1.2 Statecharts mit hierarchischen Zuständen

11.	○ Layout I		○ gleich gut		○ Layout II				
12.	○ Layout I		○ gleich gut		○ Layout II				
13.	○ Layout I		○ gleich gut		○ Layout II				
14.	○ Layout I		○ gleich gut		○ Layout II				
15.	○ Layout I		○ gleich gut		○ Layout II				
16.	○ Layout I		○ gleich gut		○ Layout II				
17.	○ Layout I		○ gleich gut		○ Layout II				
18.	○ Layout I		○ gleich gut		○ Layout II				
19.	○ Layout I		○ gleich gut		○ Layout II				
20.	○ Layout I		○ gleich gut		○ Layout II				

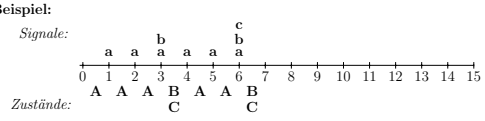
1.3 Statecharts mit parallelen Zuständen

21.	○ Layout I		○ gleich gut		○ Layout II				
22.	○ Layout I		○ gleich gut		○ Layout II				
23.	○ Layout I		○ gleich gut		○ Layout II				
24.	○ Layout I		○ gleich gut		○ Layout II				
25.	○ Layout I		○ gleich gut		○ Layout II				
26.	○ Layout I		○ gleich gut		○ Layout II				
27.	○ Layout I		○ gleich gut		○ Layout II				
28.	○ Layout I		○ gleich gut		○ Layout II				
29.	○ Layout I		○ gleich gut		○ Layout II				
30.	○ Layout I		○ gleich gut		○ Layout II				

2 Verstehen von Statecharts

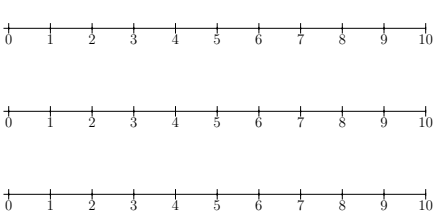
Aufgabe: Im Folgenden werden Ihnen Statecharts mit verschiedenem Aussehen vorgelegt. Leiten Sie aus jedem vorgelegten Statechart, entsprechend der in der Vorlesung vorgestellten Semantik für *Safe State Machines*, die Zustandsfolge und die Signalfolge ab. Tragen Sie Ihre Ergebnisse auf dem entsprechenden Zeitstrahl ab. Sollten mehrere Signale gleichzeitig *present* sein, tragen Sie diese übereinander auf dem Zeitstrahl ab. Sollten mehrere Zustände während einer Zeit-Instanz aktiv sein (z. B. durch Parallelität oder ein *immediate*-Signal), tragen Sie diese unterhalb des Zeitstrahls für diese Zeiteinheit ab. Verwenden Sie bei hierarchischen Zuständen dabei immer nur die inneren Zustände (*Simple States*). Knoten, die keine echten Zustände darstellen (z. B. *Initial-*, *Choice-*, *History-Connector*) notieren Sie bitte nicht.

Beispiel:

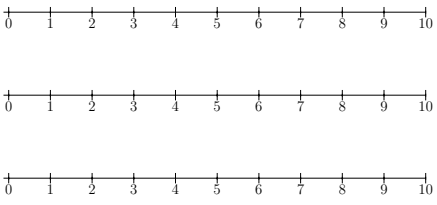


Ihre Lösung wird jeweils vom Experimentleiter auf Richtigkeit geprüft. Sollte die Lösung falsch sein, werden Sie gebeten, die Lösung zu überdenken und richtig zu lösen. Dafür stehen Ihnen pro Teilaufgabe drei Zeitstrahle zur Verfügung. Sollten Sie nach dem dritten Versuch keine richtige Lösung haben, gehen Sie zur nächsten Teilaufgabe über. Versuchen Sie jedoch möglichst schon im ersten Versuch zum Erfolg zu kommen.

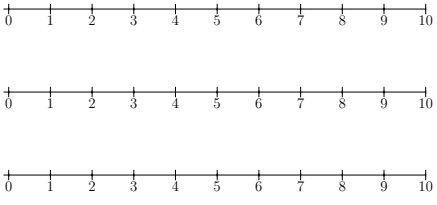
1.



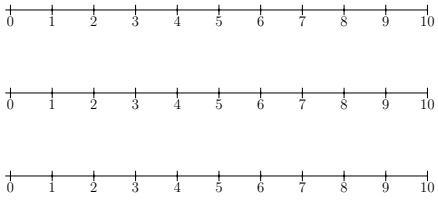
2.



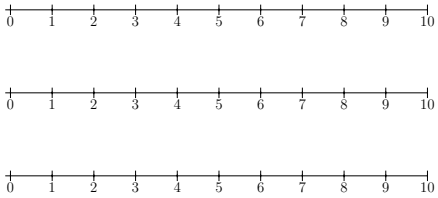
3.



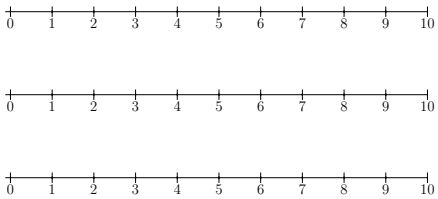
4.



5.

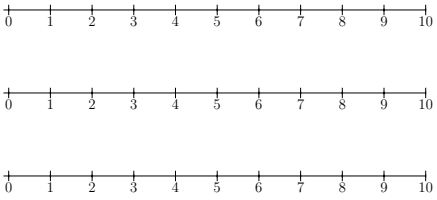


6.

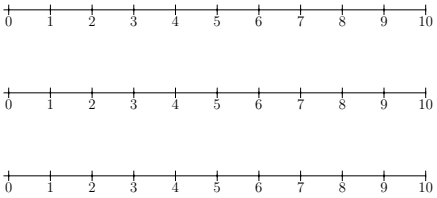


9

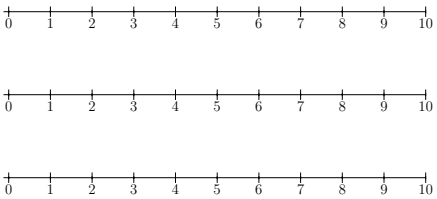
7.



8.

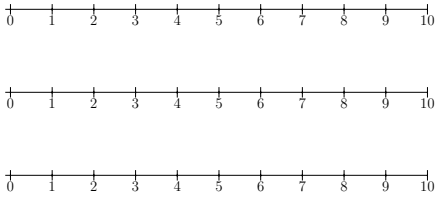


9.

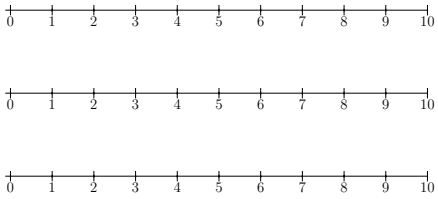


10

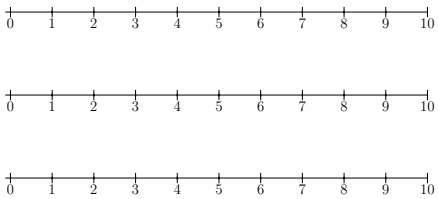
10.



11.

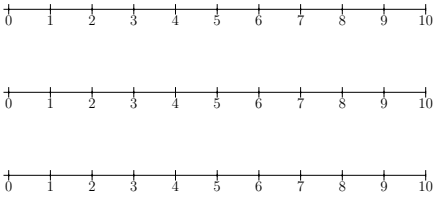


12.

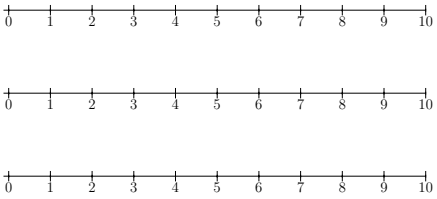


11

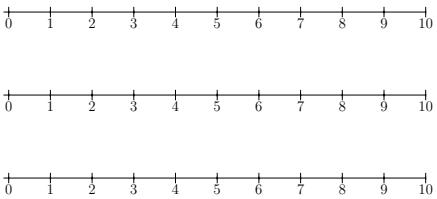
13.



14.



15.



12

3 Modellieren von Statecharts

Aufgabe:

Neuerstellen eines Statecharts

In dieser Aufgabe sollen Sie ein Ihnen vorgelegtes Statechart mit drei verschiedenen Werkzeugen erstellen. Benutzen Sie dabei die im Daten-Dokument angegebenen Werkzeuge unbedingt in der dort angegeben Reihenfolge. Das Statechart liegt in der formalen Notation vor, die Ihnen bereits aus der Übung zur Vorlesung „Modellbasierter Entwurf und Verteilte Echtzeitsysteme“ bekannt ist. Das so entworfene Statechart sollte am Ende de Übung leserlich sein, d. h. alle Elemente sollten klar erkennbar sein und eindeutig zugeordnet werden können. Das Statechart muss dabei aber nicht „schön“ sein.

Aufgabe:

Verändern eines Statecharts

Ergänzen Sie nun in jedem Werkzeug entsprechend der angebeben Reihenfolge der Werkzeuge die gerade erstellten Statecharts entsprechend der ergänzenden formalen Beschreibung. Das so entworfene Statechart sollte am Ende de Übung leserlich sein, d. h. alle Elemente sollten klar erkennbar sein und eindeutig zugeordnet werden können. Das Statechart muss dabei aber nicht „schön“ sein.

4 Fragebogen

Aufgabe: Auf den folgenden Seiten finden Sie einen Fragebogen. Füllen Sie diesen bitte aus!

1. Haben Sie sich nach dem ersten Experiment noch mit dem Verstehen von Statecharts beschäftigt?

   ○ ○ ○ ○ ○
   gar nicht — sehr intensiv

2. Haben Sie nach dem ersten Experiment noch verschiedene Statechart-Modellierungswerkzeuge benutzt?

   ○ ○ ○ ○ ○
   gar nicht — sehr intensiv

3. In welchem Rahmen habe Sie sich mit diesen Modellierungswerkzeugen beschäftigt?

   ○ Vorlesung
   ○ Übung
   ○ Hausaufgabe
   ○ private Zwecke

4. Mit welchen Werkzeugen haben Sie sich beschäftigt?

   ○ graphisches Werkzeug
   ○ Struktur-basiertes Werkzeug
   ○ textuelles Werkzeug

5. Wie schätzen Sie nach den Experimenten die Aussagekraft von Statecharts bzgl. der Spezifikation Eingebetteter Systeme ein?

   ○ ○ ○ ○ ○
   ohne Aussagekraft — starke Aussagekraft

6. Eingebettete Systeme können mit textuellen und visuelle Sprachen beschrieben werden. Welche der beiden Ansätze bevorzugen Sie?

   ○ ○ ○ ○ ○
   textuelle Sprachen — visuelle Sprachen

7. Wie wichtig ist Ihnen Sekundärnotation für textuelle Sprachen (d. h. z. B. Einrückungen bei Methoden, Zeilenumbrüche, etc.)?

○	○	○	○	○
unwichtig				sehr wichtig

8. Wie wichtig ist Ihnen Sekundärnotation für visuelle Sprachen (d., h. Anordnung der graphischen Elemente, Einhalten von Editier-Richtlinien, etc.)?

○	○	○	○	○
unwichtig				sehr wichtig

9. Sie haben in diesem Experiment drei verschiedene Werkzeuge benutzt. Welche Vor- bzw. Nachteile sehen Sie bei der Benutzung des graphischen Werkzeuges?

10. Welche Vor- bzw. Nachteile sehen Sie bei der Benutzung des Struktur-basierten Werkzeugs?

11. Welche Vor- bzw. Nachteile sehen Sie bei der Benutzung des textuellen Werkzeugs?

12. Was halten Sie von der Kombination des textuellen Werkzeugs mit der graphischen Darstellung?

13. Welches Werkzeug bevorzugen Sie?

- ◯ graphisches Werkzeug
- ◯ Struktur-basiertes Werkzeug
- ◯ textuelles Werkzeug (ohne graphischer Anzeige)
- ◯ textuelles Werkzeug (mit graphischer Anzeige)

Fragen zum Experiment II

14. Was war besonders schwierig, einfach, lästig, angenehm (etc.)?

15. Wie schätzen Sie die Geschwindigkeit Ihrer Lösung ein?

○	○	○	○	○
sehr langsam				sehr schnell

16. Wie schätzen Sie die Qualität Ihrer Lösung ein?

○	○	○	○	○
sehr schlecht				sehr gut

17. Sind Sie mit Leistung und Geschwindigkeit zufrieden?

○	○	○	○	○
überhaupt nicht zufrieden				sehr zufrieden

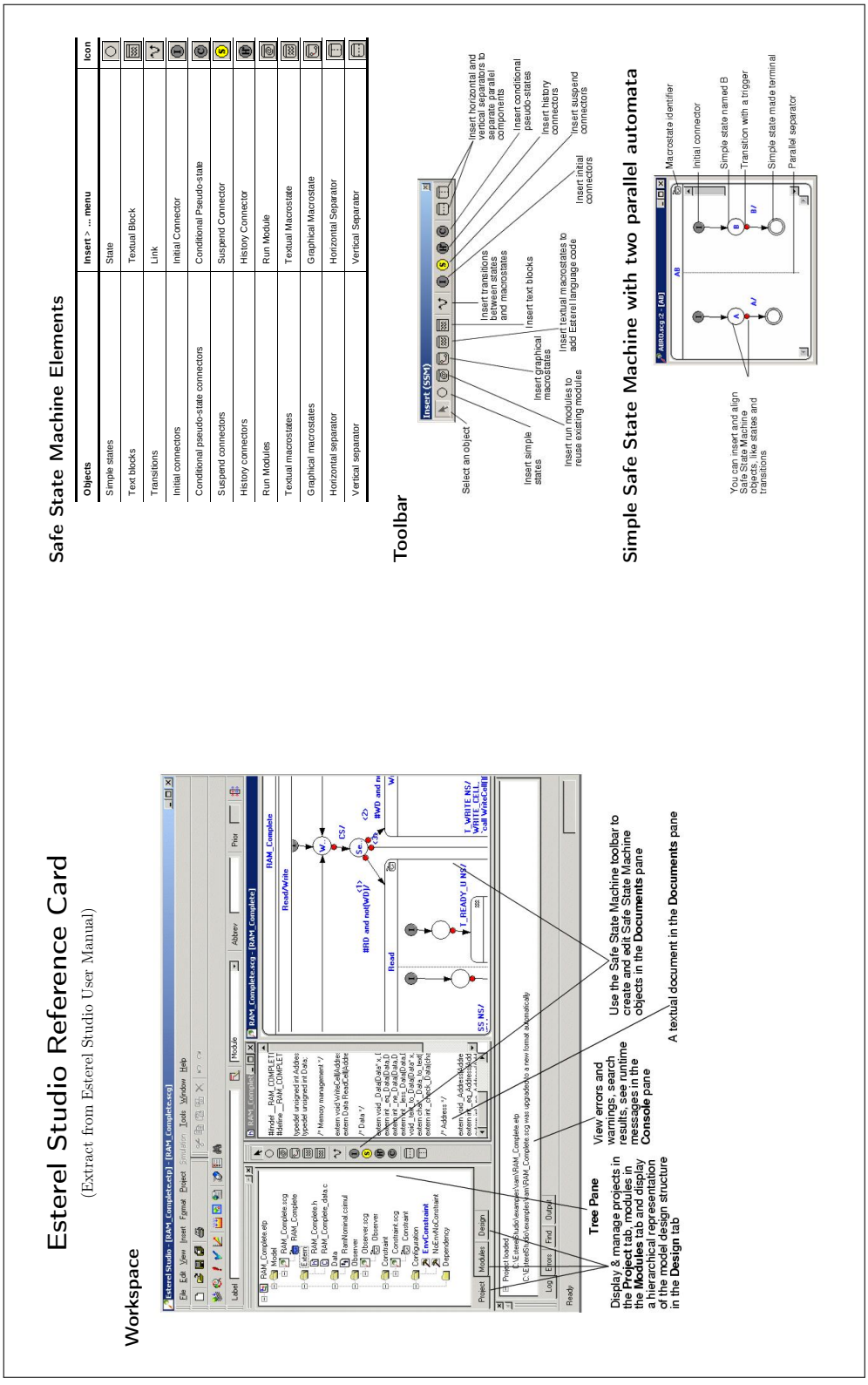
18. Warum?

19. Was hätte zur Verbesserung Ihrer Leistung beigetragen?

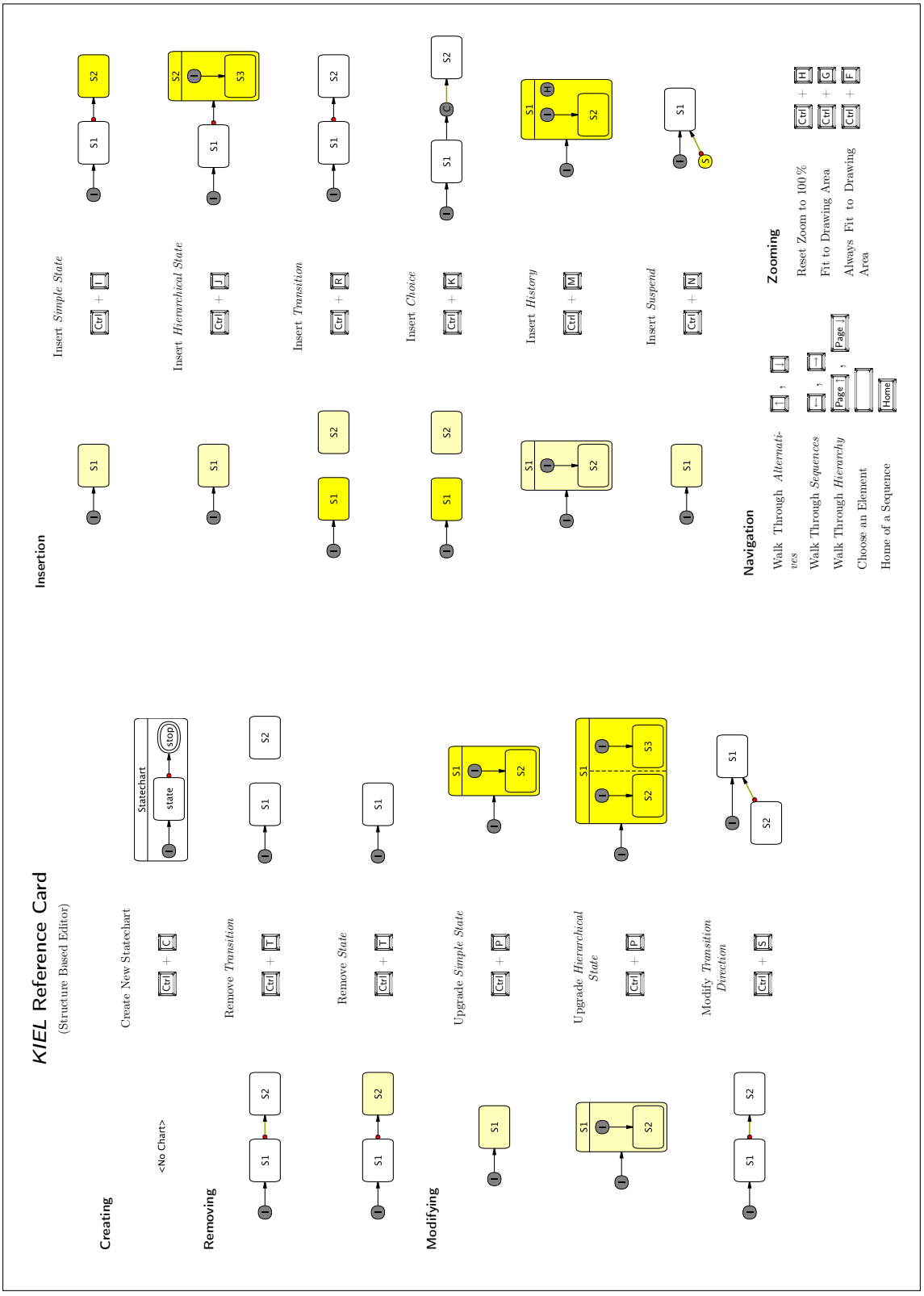
20. Welche Kommentare möchten Sie sonst noch zum Experiment machen?

C.3 THE TOOL REFERENCE CARDS

C.3.1 *Esterel Studio Reference Card*



C.3.2 *KIEL Macro Editor Reference Card*

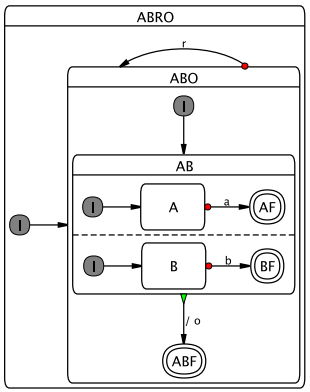


KIT Reference Card

(Textual Statechart Description Language)

Example ABRO

```
1  statechart ABRO[model="Esterel Studio";version="5.0"]{
2   input a;
3   input b;
4   input r;
5   output o;
6   {
7    ABO{
8     AB{
9       ->A;
10      A->AF[type=sa;label="a"];
11      AF[type=final];
12     ||
13      ->B;
14      B->BF[type=sa;label="b"];
15      BF[type=final];
16    };
17    ->AB;
18    AB->ABF[type=nt;label="/ o"];
19    ABF[type=final];
20   };
21   ->ABO;
22   ABO->ABO[type=sa;label="r"];
23  };
24 };
```



Statechart

Create a new Statechart

Ctrl + C

Statechart *Header* and *Body*

```
1  statechart ABRO[model="Esterel Studio";version="5.0"]{
   :
6  {
   :
23  };
24 };
```

Declarations

Signal

Types: `input|output`

```
2   input a;
3   input b;
4   input r;
5   output o;
```

States

Simple State

```
9     ->A;
```

Remark: The identifier `->` creates an initial state, which points to state `A` (see Section *State Types*).

Hierarchical State

```
7   ABO{
   :
16    };
```

Parallel State

```
8     AB{
   :
12     ||
   :
16    };
```

State *Label*

State labels are used to declare state names, which differ from the node name.

Example:
```
->MyState;
MyState->node;
node[label="AnOtherStateName"];
```

State *Type*

```
11     AF[type=final];
```

Types: `history|initial|choice|suspend|final`

Abbreviation: The initial doesn't need to be written. An `->A` creates an initial state with transition to state `A`.

Transition

Two states are connected by a transition denoted by the symbol "`->`".

Examples: `A->B`, `A->A` (Self-Loop)

```
10     A->AF[type=sa;label="a"];
```

Transition *Label*

The transition label follows the Statechart transition notation : `e` Event/Signal, `[c]` Guard/Condition, `/a` Action

Example: `A->B[label="e[c]/a"];`

Transition *Priority*

The priority is of the natural numbers ($\mathbb{N}$). A transition labeled with number "1" is of the highest priority relatively to other transitions coming from the same state. A single outgoing transition always has priority "1" and doesn't have to be written.

Example: `A->B[priority="1"];`

Transition *Type*

Types: `nt|sa|wa`
(normal termination, strong abortion, weak abortion)

Example: `A->AF[type=sa];`

BIBLIOGRAPHY

[1] Scott W. Ambler. *The Elements of UML 2.0 Style*. Cambridge University Press, New York, NY, USA, 2005. (Cited on page 86.)

[2] Charles André. Computing SyncCharts reactions. 88:3 – 19, 2004. ISSN 1571-0661. doi: DOI:10.1016/j.entcs.2003.05.007. URL URL: http://www.sciencedirect.com/science/article/B75H1-4DN4CWN-2/2/796eaef2f3793bf8e3b89bfb5226aed0. SLAP 2003: Synchronous Languages, Applications and Programming, A Satellite Workshop of ECRST 2003. (Cited on pages 10 and 24.)

[3] Charles André. Semantics of S.S.M (Safe State Machine). Technical report, Esterel Technologies, Sophia-Antipolis, France, April 2003. URL: http://www.esterel-technologies.com. (Cited on pages 2, 10, 28, 64, 65, and 104.)

[4] Charles André. SyncCharts: A visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France, Rev. April 1996. URL: http://www.i3s.unice.fr/~andre/CAPublis/SYNCCHARTS/SyncCharts.pdf. (Cited on page 24.)

[5] Charles André. Representation and analysis of reactive behaviors: A synchronous approach. In *Computational Engineering in Systems Applications (CESA)*, pages 19–29, Lille, France, July 1996. IEEE-SMC. URL: http://www.i3s.unice.fr/~andre/CAPublis/Cesa96/SyncCharts_Cesa96.pdf. (Cited on pages 10 and 24.)

[6] ArgoUML. Tigris.org. Open source software engineering tools, 2006. URL: http://argouml.tigris.org/. (Cited on pages 10 and 12.)

[7] Cyrille Artho. Jlint – find bugs in Java programs, 2006. URL: http://jlint.sourceforge.net/. (Cited on page 88.)

[8] Roswitha Bardohl. GenGEd – a visual environment for visual languages. *Science of Computer Programming, Special Issue of GraTra '00*, 2002. (Cited on page 16.)

[9] Clark Barrett, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The satisfiability modulo theories library (SMT-LIB), 2008. URL: `http://www.smt-lib.org/`. (Cited on page 113.)

[10] Clark W. Barrett and Sergey Berezin. CVC Lite: A new implementation of the Cooperating Validity Checker Category B. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of Computer Aided Verification: 16th International Conference, CAV 2004, Boston*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer, 2004. (Cited on pages 100 and 113.)

[11] Carlo Batini, Enrico Nardelli, and Roberto Tamassia. A layout algorithm for data flow diagrams. *IEEE Transactions on Software Engineering*, 12(4):538–546, 1986. ISSN 0098-5589. (Cited on page 17.)

[12] David M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the Fourth Annual USENIX Tcl/Tk Workshop*, pages 129–139, 1996. (Cited on page 113.)

[13] Benjamin B. Bederson and James D. Hollan. Pad++: A zooming graphical interface for exploring alternate interface physics. In *UIST '94: Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 17–26, New York, NY, USA, 1994. ACM Press. (Cited on page 22.)

[14] Michael von der Beeck. A comparison of statecharts variants. In H. Langmaack, W. P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. Springer-Verlag, 1994. (Cited on page 10.)

[15] Ken Bell. Überprüfung der Syntaktischen Robustheit von Statecharts auf der Basis von OCL. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, November 2006. (Cited on pages 28, 91, 92, 95, and 110.)

[16] Stefan Berner, Stefan Joos, Martin Glinz, and M. Arnold. A visualization concept for hierarchical object models. In *Automated Software Engineering*, pages 225–, 1998. (Cited on page 22.)

[17] Gérard Berry. The foundations of Esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte. (Cited on page 49.)

[18] Gérard Berry. *The Esterel v5 Language Primer, Version v5_91*. Centre de Mathématiques Appliquées Ecole des Mines and INRIA, 06565 Sophia-Antipolis, 2000. URL: `ftp://ftp-sop.inria.fr/esterel/pub/papers/primer.pdf`. (Cited on pages 42, 51, and 53.)

[19] Gérard Berry. *The Constructive Semantics of Pure Esterel*. Draft Book, 1999. URL: `ftp://ftp-sop.inria.fr/esterel/pub/papers/constructiveness3.ps`. (Cited on pages 51, 70, and 71.)

[20] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992. (Cited on pages 1 and 49.)

[21] Dag Björklund, Johan Lilius, and Ivan Porres. Towards efficient code synthesis from Statecharts. In A. Evans, R. France, A. Moreira, and B. Rumpe, editors, *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists*, volume P-7, Toronto, Canada, October 2001. Workshop of the pUML-Group held together with the UML 2001 Conference, Lecture Notes in Informatics (LNI). (Cited on page 1.)

[22] Dag Björklund, Johan Lilius, and Ivan Porres. A unified approach to code generation from behavioral diagrams. In *Languages for system specification: Selected contributions on UML, systemC, system Verilog, mixed-signal systems, and property specification from FDL '03*, pages 21–34, Norwell, MA, USA, 2004. Kluwer Academic Publishers. (Cited on page 1.)

[23] Christopher Brooks, Chih-Hong Patrick Cheng, Thomas Huining Feng, Edward A. Lee, and Reinhard

von Hanxleden. Model engineering using multimodeling. Technical Report UCB/EECS-2008-39, EECS Department, University of California, Berkeley, April 2008. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-39.html. (Cited on page 16.)

[24] Daniel Buck and Andreas Rau. On modelling guidelines: Flowchart patterns for Stateflow. *Softwaretechnik-Trends*, 21 (2):7–12, August 2001. (Cited on pages 6, 81, and 91.)

[25] Oliver Burn. Checkstyle, 2005. URL: http://checkstyle.sourceforge.net/. (Cited on page 88.)

[26] L. W. Cannon, R. A. Elliott, L. W. Kirchhoff, J. H. Miller, J. M. Milner, R. W. Mitze, E. P. Schan, N. O. Whittington, Henry Spencer, David Keppel, and Mark Brader. Recommended c style and coding standards. URL: http://www.chris-lott.org/resources/cstyle/indhill-cstyle.html. (Cited on page 34.)

[27] Rodolfo Castelló, Rym Mili, and Ioannis G. Tollis. A framework for the static and interactive visualization for statecharts. *Journal of Graph Algorithms and Applications*, 6(3):313–351, 2002. (Cited on pages 17, 18, 19, 24, and 34.)

[28] Rodolfo Castelló, Rym Mili, and Ioannis G. Tollis. ViSta. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing : 9th International Symposium, GD 2001*, volume 2265 of *Lecture Notes in Computer Science*, pages 481–482. Springer, 2002. (Cited on page 17.)

[29] Juan de Lara, Hans Vangheluwe, and Manuel Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. *Software and Systems Modeling (SoSyM)*, 3(3):194–209, August 2004. (Cited on page 16.)

[30] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: An annotated bibliography. *Computational Geometry: Theory and Applications*, 4:235–282, June 1994. (Cited on pages 17 and 102.)

[31] Stephan Diehl, Carsten Görg, and Andreas Kerren. Preserving the mental map using foresighted layout. In *Proceed-*

ings of Joint Eurographics—IEEE TCVG Symposium on Visualization, VisSym 2001, pages 175–184, Ascona, Switzerland, 2001. Springer Verlag, 2001. (Cited on pages 24 and 80.)

[32] Jerry Doland and Jon Valett. C style guide. Technical Report, Software Engineering Laboratory Series SEL-94-003, National Aeronautics and Space Administration (NASA), 1994. (Cited on pages 26 and 88.)

[33] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984. ISSN 0384-9864. (Cited on page 17.)

[34] Stephen A. Edwards. CEC: The Columbia Esterel Compiler, 2006. URL: http://www1.cs.columbia.edu/~sedwards/cec/. (Cited on pages 109 and 130.)

[35] Sol Efroni, David Harel, and Irun R. Cohen. Reactive animation: Realistic modeling of complex dynamic systems. *Computer*, 38(1):38–47, January 2005. (Cited on page 17.)

[36] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003. ISSN 0018-9219. doi: 10.1109/JPROC.2002.805829. (Cited on page 16.)

[37] Estbench Esterel Benchmark Suite. Estbench Esterel Benchmark Suite, 2007. URL: http://www1.cs.columbia.edu/~sedwards/software/estbench-1.0.tar.gz. (Cited on pages 58, 130, and 132.)

[38] *Esterel Studio User Manual*. Esterel Technologies, 5.2 edition, July 2004. (Cited on page 11.)

[39] Esterel Technologies, Inc. *Esterel Studio User Manual*. Esterel Technologies, Inc, 6.0 edition, December 2007. (Cited on page 10.)

[40] Esterel Technologies, Inc. SCADE Suite, last visited 05/2008. URL: http://www.esterel-technologies.com/products/scade-suite/. (Cited on pages 42 and 141.)

[41] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1): 42–51, 2002. (Cited on pages 85 and 88.)

[42] Thomas Huining Feng. An extended semantics for a Statechart Virtual Machine. In A. Bruzzone and Mhamed Itmi, editors, *Summer Computer Simulation Conference (SCSC 2003), Student Workshop*, pages 147–166. The Society for Computer Modelling and Simulation, July 2003. Montréal, Canada. (Cited on pages 25 and 47.)

[43] Rudolf Fleischer and Colin Hirsch. Graph drawing and its applications. In Michael Kaufmann and Dorothea Wagner, editors, *Drawing Graphs: Methods and Models*, number 2025, pages 1–22. Springer-Verlag, Berlin, Germany, 2001. (Cited on page 17.)

[44] Bastian Florentz, Martin Mutz, and Michaela Huhn. Avoiding unpredicted behaviour of large scale embedded systems by design and application of modelling rules. In *Proceedings of the 2004 First International Workshop on Model, Design and Validation*, November 2004. (Cited on page 25.)

[45] Ford Motor Company. *Structured Analysis Using Matlab/Simulink/Stateflow Modeling Style Guidelines*, 1999. URL: http://vehicle.berkeley.edu/mobies/papers/stylev242.pdf. (Cited on pages 26 and 94.)

[46] Bernd Freimut, Teade Punter, Stefan Biffl, and Marcus Ciolkowski. State-of-the-art in empirical studies. Technical report, ViSEK Technical Report 007/E, 2002. (Cited on page 115.)

[47] Manuel Freire and Pilar Rodríguez. A graph-based interface to complex hypermedia structure visualization. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, pages 163–166, New York, NY, USA, 2004. ACM. (Cited on page 24.)

[48] Manuel Freire and Pilar Rodríguez. Preserving the mental map in interactive graph interfaces. In *AVI '06: Proceedings of the working conference on Advanced visual interfaces*, pages 270–273, New York, NY, USA, 2006. ACM. (Cited on pages 23 and 24.)

[49] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software—Practice & Experience*, 21(11):1129–1164, 1991. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.4380211102. (Cited on page 138.)

[50] Hauke Fuhrmann and Reinhard von Hanxleden. The Kiel Integrated Environment for Layout for the Eclipse RichClientPlatform (KIELER) Homepage, 2009. URL: http://www.informatik.uni-kiel.de/rtsys/kieler/. (Cited on page 141.)

[51] Gerge W. Furnas. Generalized fisheye views. In *Human Factors in Computings Systems CHI '86 Conference Proceedings*, pages 16–23, 1986. (Cited on page 20.)

[52] Etienne Gagnon. SableCC: Java parser generator. URL: http://sablecc.org/. (Cited on page 107.)

[53] Etienne M. Gagnon and Laurie J. Hendren. SableCC, an object-oriented compiler framework. In *TOOLS (26)*, pages 140–154. IEEE Computer Society, 1998. URL URL: http://doi.ieeecomputersociety.org/10.1109/TOOLS.1998.711009. (Cited on page 107.)

[54] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Revised On-Line Version (2003), Philadelphia, PA, June 2003. URL: http://www.cis.upenn.edu/~jean/gbooks/logic.html. (Cited on page 113.)

[55] Emden Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with dot. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, February 2002. (Cited on pages 18, 20, and 48.)

[56] Emden R. Gansner. Drawing graphs with GraphViz. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, November 2004. (Cited on page 18.)

[57] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience*, 30(11):1203–1234, 2000. ISSN 00380644. (Cited on pages 18, 25, 28, 99, and 101.)

[58] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230, 1993. (Cited on page 102.)

[59] GenGED. Project homepage. URL: http://tfs.cs.tu-berlin.de/~genged/. (Cited on page 16.)

[60] Joseph Gil and Stuart Kent. Three dimensional software modelling. In *Proceedings of the 20th international conference on Software engineering*, pages 105–114. IEEE Computer Society, 1998. (Cited on page 22.)

[61] Stefania Gnesi and Franco Mazzanti. On the fly model checking of communicating UML State Machines. In *Second ACIS International Conference on Software Engineering Research Management and Applications (SERA2004)*, 2004. (Cited on page 47.)

[62] Martin Gogolla and Francesco Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In Manfred Broy, Derek Coleman, Tom S. E. Maibaum, and Bernhard Rumpe, editors, *Proceedings PSMT '98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998. (Cited on page 27.)

[63] GraphViz. Graphviz—graph drawing tools, 2007. URL: http://graphviz.org/. (Cited on page 18.)

[64] T. R. G. Green and M. Petre. When visual programs are harder to read than textual programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings ECCE-6 (6th European Conference Cognitive Ergonomics)*, 1992. (Cited on pages 16 and 17.)

[65] Jan Friso Groote and Frank van Ham. Large state space visualization. In Hubert Garavel and John Hatcliff, editors, *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003*, volume 2619 of *Lecture Notes in Computer Science*, pages 585–590. Springer, 2003. (Cited on pages 22 and 23.)

[66] Paul Le Guernic, Thierry Goutier, Michel Le Borgne, and Claude Le Maire. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9), September 1991. (Cited on page 1.)

[67] Corin Gurr. Aligning syntax and semantics in formalisations of visual languages. In *HCC '01: Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, page 60, Washington, DC, USA, 2001. IEEE Computer Society. (Cited on page 34.)

[68] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. (Cited on pages 1 and 141.)

[69] Reinhard von Hanxleden. Stateflow and STATEMATE – a comparison of statechart dialects. Project Report FT3/AS-1999-003, DaimlerChrysler, August 1999. (Cited on page 85.)

[70] Reinhard von Hanxleden, Kay Kossowan, and Horst Burmeister. Robustness analysis for statecharts. Presentation, 2000. URL: http://www.vmars.tuwien.ac.at/projects/setta/docs/Slides07_StateAnalyzerTalk.pdf. (Cited on page 85.)

[71] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987. (Cited on pages 1, 2, 9, 110, 111, and 133.)

[72] David Harel and Gregory Yashchin. An algorithm for blob hierarchy layout. *The Visual Computer*, 18:164–185, 2002. (Cited on pages 17 and 34.)

[73] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990. (Cited on pages 2, 3, and 132.)

[74] Jeanette Heidenberg, Andreas Nåls, and Ivan Porres. Statechart features and pre-release defects in software maintenance. In *2007 IEEE Symposium on Visual Languages*

and Human-Centric Computing, Coeur d'Aléne, Idaho, USA, September 2007. (Cited on pages 41 and 75.)

[75] Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-based requirements. *Software Engineering*, 22(6):363–377, 1996. (Cited on pages 25 and 27.)

[76] Constance Heitmeyer, Ralph Jeffords, and Bruce Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*. (Cited on page 27.)

[77] Karsten Heymann. Ein LaTeX-Style zur Benutzung von KIEL-Statecharts. Bachelor project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, 2006. (Cited on page 28.)

[78] Ralf Huuck. Sanity checks for Stateflow diagrams. In Stephen A. Edwards, Nicolas Halbwachs, Reinhard v. Hanxleden, and Thomas Stauner, editors, *Synchronous Programming – SYNCHRON '04*, number 04491 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany. (Cited on page 26.)

[79] Matthew S. Jaffe, Nancy G. Leveson, Mats P. E. Heimdahl, and Bonnie E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991. (Cited on page 24.)

[80] Stephen C. Johnson. Lint, a C program checker. In Ken Thompson and Dennis M. Ritchie, editors, *UNIX Programmer's Manual*. Bell Laboratories, seventh edition, 1979. (Cited on page 88.)

[81] Michael Jünger and Petra Mutzel. *Graph Drawing Software*. Springer, October 2003. (Cited on page 17.)

[82] Natalia Juristo and Ana M. Moreno. *Basics of Software Engineering Experimentation*. Springer, 2001. (Cited on page 116.)

[83] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989. ISSN 0020-0190. URL: http://dx.doi.org/10.1016/0020-0190(89)90102-6. (Cited on page 17.)

[84] Tobias Kloss. Flexibles und Automatisiertes Layout von Statecharts. Student resarch project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2003. (Cited on page 28.)

[85] Tobias Kloss. Automatisches Layout von Statecharts unter Verwendung von GraphViz. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2005. (Cited on page 28.)

[86] Hideki Koike and Hirotaka Yoshihara. Fractal approaches for visualizing huge hierarchies. In Ephraim P. Glinert and Kai A. Olsen, editors, *Proc. IEEE Symp. Visual Languages, VL*, pages 55–60. IEEE Computer Society, 1993. (Cited on page 22.)

[87] Kay Kossowan. Automatisierte Überprüfung semantischer Modellierungsrichtlinien für Statecharts. Diplomarbeit, Technische Universität Berlin, 2000. (Cited on pages 89, 95, and 96.)

[88] Oliver Köth and Mark Minas. Structure, Abstraction, and Direct Manipulation in Diagram Editors. In *DIAGRAMS '02: Proceedings of the Second International Conference on Diagrammatic Representation and Inference*, pages 290–304, London, UK, 2002. Springer-Verlag. (Cited on pages 15, 22, and 75.)

[89] Th. Kreppold. Modellierung mit Statemate MAGNUM und Rhapsody in Micro C. Berner & Mattner Systemtechnik GmbH, Otto-Hahn-Str. 34, 85521 Ottobrunn, Germany, Dok.-Nr.: BMS/QM/RL/STM, Version 1.4, August 2001. (Cited on page 26.)

[90] Lars Kühl. Transformation von Esterel nach Esterel Studio. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, September 2005. (Cited on pages 28, 51, 54, 56, and 71.)

[91] Leslie Lamport. *LaTeX – A Document Preparation System*. Adddison-Wesley, 1994. (Cited on page 39.)

[92] Brenda Laurel. *Design Principles for Human-Computer-Activity*, chapter 5. In Laurel [93], 1991. (Cited on page 36.)

[93] Brenda Laurel. *Computers as Theatre*. Addison-Wesley, 1991. (Cited on page 198.)

[94] Ying K. Leung and Mark D. Apperley. A review and taxonomy of distortion-oriented presentation techniques. *ACM Transactions on Computer-Human Interaction*, 1(2):126–160, June 1994. (Cited on page 20.)

[95] Jan Lukoschus. *Removing Cycles in Esterel Programs*. PhD thesis, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, July 2006. URL: http://eldiss.uni-kiel.de/macau/receive/dissertation_diss_2015. (Cited on page 45.)

[96] Paul Lukowicz, Ernst A. Heinz, Lutz Prechelt, and Walter F. Tichy. Experimental evaluation in computer science: A quantitative study. *Journal of Systems ans Software*, 28(1):9–18, January 1995. Also as TR 17/94 (August 1994), Fakultät für Informatik, Universtität Karlsruhe, Germany, ftp.ira.uka.de. (Cited on page 116.)

[97] Florian Lüpke. Implementierung eines Statechart-Editors mit layoutbasierten Bearbeitungshilfen. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, June 2005. (Cited on page 28.)

[98] F. Maraninchi and Y. Rémond. Argos: An automaton-based synchronous language. *Computer Languages*, 27(27):61–92, 2001. (Cited on pages 25 and 48.)

[99] Florence Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, October 1991. (Cited on page 25.)

[100] MathWorks Automotive Advisory Board (MAAB). *Controller Style Guidelines for Production Intent Using MATLAB, Simulink*

and Stateflow, April 2001. URL: http://www.mathworks.com/industries/auto/maab.html. (Cited on page 26.)

[101] Mathworks Inc. *Simulink – Simulation and Model-Based Design*. The Mathworks, Inc., Natick, MA, 2005. URL: http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sl_using.pdf. (Cited on pages 10 and 11.)

[102] Franco Mazzanti. *UMC User Guide (Version 2.5)*. Istituto di Scienza e Tecnologie dell'Informazione "Alessandro Faedo" (ISTI), Pisa, Italy, 2003. (Cited on page 25.)

[103] Linda McIver and Damian Conway. Seven deadly sins of introductory programming language design. *International Conference on Software Engineering: Education and Practice (SE:EP'96)*, 00:309, 1996. (Cited on pages 25 and 45.)

[104] Mark Minas. Specifying Statecharts with DiaGen. HCC '01 – 2001 IEEE Symposia on Human-Centric Computing Languages and Environments, Symposium on Visual Languages and Formal Methods, Statechart Modeling Contest, September 2001. (Cited on pages 15 and 22.)

[105] Mark Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming (SCP)*, 2001. (Cited on page 15.)

[106] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages & Computing*, 6(2):183–210, June 1995. (Cited on pages 22 and 23.)

[107] Motor Industry Software Reliability Association (MISRA). *MISRA-C:2004. Guidelines for the Use of the C Language in Critical Systems*. Motor Industry Research Association (MIRA), Nuneaton CV10 0TU, UK, 2004. (Cited on pages 26, 85, 86, and 88.)

[108] Miltiadis Moutos, Albrecht Korn, and Carsten Fisel. Guideline-Checker. Studienarbeit, University of Applied Sciences in Esslingen, June 2000. (Cited on page 27.)

[109] Martin Mutz. *Eine durchgängige modellbasierte Entwurfsmethodik für eingebettete Systeme im Automobilbereich*. Dissertation, Technische Universität Braunschweig, 2005. (Cited on pages 26, 27, 93, and 110.)

[110] Martin Mutz and Michaela Huhn. Automated statechart analysis for user-defined design rules. Technical report, Technische Universität Braunschweig, 2003. (Cited on pages 27, 89, 90, and 140.)

[111] Nabeel Al-Shamma and Robert Ayers and Richard Cohn and Jon Ferraiolo and Martin Newell and Roger K. de Bry and Kevin McCluskey and Jerry Evans. Precision graphics markup language (PGML). World Wide Web Consortium Note 10-April-1998. URL: http://www.w3.org/TR/1998/NOTE-PGML. (Cited on page 39.)

[112] National Instruments. LabVIEW, visited 03/2008. URL: http://www.ni.com/labview/. (Cited on page 16.)

[113] Object Management Group. Unified Modeling Language (UML) 1.3 specification, February 2000. URL: http://www.omg.org/cgi-bin/apps/doc?formal/00-03-01.pdf. (Cited on pages 26, 91, 92, and 100.)

[114] Object Management Group. Unified Modeling Lanugage—UML resource page, 2005. URL: http://www.uml.org. (Cited on pages 2 and 10.)

[115] Object Management Group. Unified Modeling Language: Superstructure, version 2.0, Aug 2005. URL: http://www.omg.org/docs/formal/05-07-04.pdf. (Cited on page 26.)

[116] Object Management Group, Inc. *Human-Usable Textual Notation (HUTN) Specification*, August 2004. URL: http://www.omg.org/technology/documents/formal/hutn.htm. (Cited on pages 25 and 46.)

[117] University of Chicago Press, editor. *The Chicago Manual of Style*. University of Chicago, 15th edition, 2003. (Cited on page 86.)

[118] André Ohlhoff. Simulating the Behavior of SyncCharts. Student research project, Christian-Albrechts-Universität zu Kiel,

Department of Computer Science, November 2004. (Cited on page 28.)

[119] Zsigmond Pap, Istvan Majzik, and Andras Pataricza. Checking general safety criteria on UML statecharts. *Lecture Notes in Computer Science*, 2187, 2001. (Cited on page 26.)

[120] David L. Parnas. Some theorems we should prove. In *HUG '93: Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 155–162, London, UK, 1994. Springer-Verlag. (Cited on page 81.)

[121] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987. (Cited on page 34.)

[122] Marian Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, June 1995. (Cited on pages 33 and 34.)

[123] Amir Pnueli and M. Shalev. What is in a step: On the semantics of Statecharts. In *TACS '91: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 244–264, London, UK, 1991. Springer-Verlag. (Cited on page 118.)

[124] Stuart Pook, Eric Lecolinet, Guy Vayssaix, and Emmanuel Barillot. *Context and Interaction in Zoomable User Interfaces*. ACM Press, 2000. (Cited on page 20.)

[125] Adrian Posor. Extension of KIEL by Stateflow charts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2005. (Cited on page 28.)

[126] Lutz Prechelt. *Kontrollierte Experimente in der Softwaretechnik: Potenzial und Methodik*. Springer, Berlin, 2001. (Cited on page 116.)

[127] Programming Research Ltd., 2005. URL: http://www.programmingresearch.com/. (Cited on page 88.)

[128] The GNU Project. GNU M4, last visited 04/2008. URL: http://www.gnu.org/software/m4/. (Cited on page 41.)

[129] The LaTeX project team. LaTeX – a document preparation system, visited 04/2007. URL: http://www.latex-project.org/. (Cited on page 39.)

[130] Helen C. Purchase, Matthew McGill, Linda Colpoys, and David Carrington. Graph drawing aesthetics and the comprehension of UML class diagrams: An empirical study. In *ACM International Conference Proceeding Series archive, Australian symposium on Information visualisation*, pages 129–137, 2001. (Cited on pages 16 and 17.)

[131] Thomas Pyrlik. Entwurf und Realisation eines OPC-Clients zur Steuerung redundanter PROFIBUS OPC-Server mit Fehlerüberwachung der PROFIBUS Peripherie. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2004. (Cited on pages 65 and 97.)

[132] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2006. URL: http://www.R-project.org. (Cited on page 120.)

[133] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:114–125, 1959. (Cited on page 9.)

[134] Rational Software. Rational Rose technical developer, 2006. URL: http://www-306.ibm.com/software/awdtools/developer/technical/. (Cited on pages 2 and 18.)

[135] E. M. Reingold and J. S. Tilford. Tidier drawing of trees. *IEEE Transactions on Software Engineering*, 7:223–228, mar 1981. (Cited on page 17.)

[136] Frederic Richard and Henry F. Ledgard. A reminder for language designers. *ACM SIGPLAN Notices*, 12(12):73–82, 1977. (Cited on pages 25 and 45.)

[137] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, University of Bremen, 2001. (Cited on pages 26 and 110.)

[138] Jason Robbins and David Redmiles. Cognitive support, UML adherence, and XMI interchange in Argo/UML. *Journal of Information and Software Technology. Special issue: The Best of COSET '99*, 42(2):79–89, 2000. (Cited on pages 12, 13, 14, and 17.)

[139] Jason Robbins, Michael Kantor, and David Redmiles. Sweeping away disorder with the broom alignment tool. In *Proceedings on Human Factors in Computing Systems (CHI '99)*, May 1999. (Cited on page 13.)

[140] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone trees: Animated 3d visualizations of hierarchical information. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 189–194. ACM Press, 1991. URL: http://doi.acm.org/10.1145/108844.108883. (Cited on page 22.)

[141] Pierre N. Robillard, Patrick d'Astous, Francoise D'tienne, and Willemien Visser. Measuring cognitive activities in software engineering. In *Proceedings of the 20th international conference on Software engineering*, pages 292–299. IEEE Computer Society, 1998. (Cited on page 17.)

[142] Manojit Sarkar and Marc H. Brown. Graphical fisheye views of graphs. In *Proceedings of the ACM SIGCHI 1992 Conference on Human Factors in Computing Systems*, pages 83–91, 1992. (Cited on pages 20 and 21.)

[143] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. Technical Report 2004-16, Verimag, Centre Équation, 38610 Gières, July 2004. URL: http://www-verimag.imag.fr/index.php?page=techrep-list. (Cited on pages 26 and 82.)

[144] Gunnar Schaefer. Statechart style checking – automated semantic robustness analysis of Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, June 2006. (Cited on pages 28, 84, 88, 89, 90, 91, 96, and 114.)

[145] Christian Scheidler. Systems engineering for time triggered architectures. SETTA Consortium, 2002. Deliverable D7.3 – Final Document. (Cited on pages 27 and 90.)

[146] K. Schneider. Embedding imperative synchronous languages in interactive theorem provers. In *Conference on Application of Concurrency to System Design (ACSD)*, pages 143–156, Newcastle upon Tyne, UK, June 2001. IEEE Computer Society. (Cited on pages 25 and 68.)

[147] M. Sheelagh, T. Carpendale, David J. Cowperthwaite, and F. David Fracchia. Extending distortion viewing from 2D to 3D. *IEEE Computer Graphics and Applications: Special Issue on Information Visualization*, 17(4):42–51, / 1997. (Cited on page 22.)

[148] F. G. Shi, J. M. Armstrong, and J. A. McDermid. A safe subset of Statecharts for safety-critical applications. Technical report, Department of Computer Science, University of York, 1996. (Cited on page 82.)

[149] Henry Spencer. The ten commandments for C programmers, 1992. URL: `http://www.lysator.liu.se/c/ten-commandments.html`. (Cited on page 88.)

[150] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11 (2):109–125, February 1981. (Cited on pages 17 and 18.)

[151] Sun Microsystems, Inc. Code conventions for the Java programming language, 1997. URL: `http://java.sun.com/docs/codeconv/`. (Cited on pages 26, 34, 85, and 88.)

[152] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man and Cybernetics*, 18(1):61–79, 1988. ISSN 0018-9472. (Cited on page 17.)

[153] Conrad Taylor. What has WYSIWYG done to us? *The Seybold Report on Publishing Systems*, 26(2), September 1996. (Cited on page 29.)

[154] The Mathworks. Stateflow—design and simulate state machines and control logic, 2008. URL: http://www.mathworks.com/products/stateflow/. (Cited on pages 2 and 10.)

[155] The Perl Foundation. The Perl directory, last visited 04/2008. URL: http://www.perl.org/. (Cited on page 41.)

[156] The Ricardo Company. Mint – a style checker for simulink and stateflow, 2006. URL: http://www.ricardo.com/engineeringservices/controlelectronics.aspx?page=mint. (Cited on page 27.)

[157] The University of Queensland. DSTC Pegamento project, last visited 03/2007. URL: http://www.dstc.edu.au/Research/Projects/Pegamento/hutn/. (Cited on page 25.)

[158] Jenifer Tidwell. Common ground: A pattern language for human-computer interface design, 1999. URL: http://www.mit.edu/~jtidwell/interaction_patterns.html. (Cited on page 38.)

[159] Dresden OCL Toolkit, 2006. URL: http://dresden-ocl.sourceforge.net/. (Cited on page 110.)

[160] Michael Tsai. WYSIWYG: Is it what you want? *About This Particular Macintosh (ATPM)*, December 1998. (Cited on page 29.)

[161] Dániel Varró, Gergely Varró, and András Pataricza. Visual graph transformation in system verification. In Elena Gramatova, Hans Manhaeve, and Adam Pawlak, editors, *Symposium on Design and Diagnostics of Electronic Systems (DDECS)*, pages 137–141, Bratislava, Slovakia, April 5-7 2000. (Cited on page 27.)

[162] Jonas Völcker. A quantitative analysis of Statechart aesthetics and Statechart development methods. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2008. URL: http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jovo-dt.pdf. (Cited on page 28.)

[163] Reinhard von Hanxleden. Lectures: Model-based design and distributed real-time systems, 2007. URL:

http://www.informatik.uni-kiel.de/rtsys/teaching/ws06-07/v-model/skript/. (Cited on page 116.)

[164] World Wide Web Consortium (W3C). URL: http://www.w3.org/html/. (Cited on page 39.)

[165] World Wide Web Consortium (W3C). W3C SVG homepage, 2005. URL: http://www.w3.org/Graphics/SVG/. (Cited on page 39.)

[166] Jos B. Warmer and Anneke G. Kleppe. *The object constraint language: Precise modeling with UML*. Addison-Wesley, Reading, Massachusetts, 1998. (Cited on page 81.)

[167] Mirko Wischer. Ein File-Interface für das KIEL Projekt – Import von Esterel-Studio-Dateien. Internship report, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, 2005. (Cited on page 28.)

[168] Mirko Wischer. Ein Browser für die Visualisierung dynamischer Sichten von Statecharts. Student research project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, August 2005. (Cited on page 28.)

[169] Mirko Wischer. Textuelle Darstellung und strukturbasiertes Editieren von Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, February 2006. (Cited on pages 28 and 49.)

[170] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Springer, 2000. (Cited on page 116.)

[171] World Wide Web Consortium (W3C). State Chart XML (SCXML): State Machine notation for control abstraction, February 2007. URL: http://www.w3.org/TR/scxml/. (Cited on pages 25 and 46.)

[172] World Wide Web Consortium (W3C). XML homepage. URL: http://www.w3.org/XML/. (Cited on page 39.)

COLOPHON

This thesis was typeset with LaTeX 2ε using André Miede's elegant *Classic Thesis* style, to suit scientific publication standards. The document text was typeset in Hermann Zapf's *Palatino* and *Euler* type faces. The listings are typeset in *Bera Mono*, originally developed by Bitstream, Inc. as "Bitstream Vera".