CHRISTIAN-ALBRECHTS-UNIVERSITÄT ZU KIEL

Diploma Thesis

Statechart Style Checking

Automated Semantic Robustness Analysis of Statecharts

Cand.-Ing. Gunnar Schaefer

June 9, 2006

Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

Advised by:

Dipl.-Inf. Steffen H. Prochnow

Eidesstattliche Erklärung

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

Kiel,

Abstract

Software for safety-critical systems is nowadays often *graphically modeled*, using what has come to be known as *statecharts*. In collaborative, complex software development projects, code efficiency, readability, and maintainability are particularly important issues. Especially, since humans tend to digress, err, and diversify, error prevention, as well as the provision of solutions for reoccurring tasks are of paramount importance. Hence, this diploma thesis focuses on the definition and automated compliance check of so-called *robustness rules* for statecharts. By these means, a *subset* of the statechart programming paradigm is specified, which is considered to be less error-prone.

Keywords: statecharts, error prevention, style checking, style guide, syntactic robustness, semantic robustness, automated theorem proving

Acknowledgements

First and foremost, I thank Steffen Prochnow, my thesis advisor and good friend. Although we have known each other for a long time, he never made it easy for me—and that's good. His sheer unlimited patience and the often long hours of advice and guidance are greatly appreciated.

Next, I would like to express my deep gratitude to Prof. Dr. Reinhard von Hanxleden for quickly developing the topic of this thesis at a time of need, when I was far away and very frustrated in India.

I would also like to acknowledge all members of the *Real-Time and Embedded Systems Group* at the University of Kiel, especially Ken Bell and Claus Traulsen.

Last, but definitely not least, I want to thank my fiancée Sylvia for standing by me all throughout the course of my graduate studies. During this time, she has not only become my partner for life, but also my greatest critic, constantly reminding me that stringent research methods are based in the philosophy of science, that meaningful empirical results need sound empirical experimentation, and also that there is more to life than working on a diploma thesis (late insights). She has accompanied me in this work in three different parts of the world—first India, then Germany, now Switzerland. Thank you!

Table of Contents

List of Tables

List of Figures

List of Acronyms and Abbreviations

API	application programming interface
ATP	automated theorem proving
BDD	binary decision diagram
CASE	computer-aided software engineering
CNF	conjunctive normal form
CVC	Cooperating Validity Checker
CVCL	Cooperating Validity Checker Lite
DNF	disjunctive normal form
DP	Davis-Putnam (algorithm)
DPLL	Davis-Logemann-Loveland (algorithm)
FSM	finite state machine
GUI	graphical user interface
IDE	integrated development environment
JAR	Java Archive
JNI	Java Native Interface
JVM	Java Virtual Machine
KIEL	Kiel Integrated Environment for Layout
MISRA	Motor Industry Software Reliability Association
NASA	National Aeronautics and Space Administration
OCL	Object Constraint Language
OMG	Object Management Group
SAT	boolean satisfiability problem
SMT	satisfiability modulo theories
SSM	Safe State Machines
SUD	system under development
SVC	Stanford Validity Checker
UML	Unified Modeling Language

VHDL	Very High-Level Design Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
w. r. t.	with respect to

Chapter 1

Introduction

Computer-controlled electronic devices, so-called embedded systems, have long become part of our everyday life. Their presence has evolved from abundance to ubiquity. Even many safety-critical systems—those where human life or at least great financial values are at stake—are controlled by software. It is, therefore, all the more important that this software does exactly what it is supposed to do. Prominent examples of safety-critical systems failure are the Therac-25 radiation therapy (1985–1987) [62], the Space Shuttle *Challenger* (1986) [105], Lufthansa Flight 2904 (1993) [60], and Ariane 5 Flight 501 (1996) [63].

The software for such safety-critical (embedded) systems is nowadays oftentimes realized without using "standard" programming languages such as C or Java. Instead, behavior is either expressed in a specialized programming language, such as Esterel [16], or graphically *modeled* at various levels of abstraction, using what has come to be known as *statecharts* [44]. According to Harel, their inventor, statecharts "constitute an extensive generalization of state-transition diagrams. They allow for multilevel states decomposed in an and/or fashion, and thus support economical specification of concurrency and encapsulation. They incorporate a broadcast communication mechanism, timeout and delay operators for specifying synchronization and timing information, and a means for specifying transitions that depend on the history of the system's behavior" [46, page 10]. With these specialized features—hierarchy, concurrency, broadcast-communication, and history—statecharts constitute a much more powerful and intuitive modeling paradigm than ordinary textual programming. Using a specialized modeling environment, *e. g.*, Esterel Studio or MATLAB Simulink/Stateflow, such a system model may be used to generate program code. Moreover, the modeled system may also be simulated by injecting events into the model, which then reacts according to its behavior.

Thus, statecharts in conjunction with a powerful modeling and simulation environment serve as specification and design language, graphical programming language, and test suite. Today, statecharts are used in the software development of such complex endeavors as the Airbus A380 and the Eurofighter.

Although utilizing statecharts for system development makes life easier, they are—by far—no magic bullet. When modeling systems graphically, certain aspects regarding quality need to be kept in mind, just as in textual computer programming:

> Good programming style begins with the effective organization of code. By using a clear and consistent organization of the components of your programs, you make them more efficient, readable, and maintainable.
>
> – Steve Oualline, *C Elements of Style* [85, page 1]

Code efficiency, readability, and maintainability are particularly important issues in collaborative, complex software development projects. Especially, since humans tend to digress, err, and diversify, common appearance and style, error prevention, as well as the provision of solutions for reoccurring tasks are of paramount importance in large programming projects. Hence, appropriate guidelines and an automated abidance check are indispensable. This diploma thesis focuses on the definition and automated check of so-called *robustness rules*, *i. e.*, rules that limit or even prevent the use of error-prone expressions.

The developed robustness checker is integrated in the statechart modeling and simulation tool *KIEL* [55]. A screenshot of *KIEL* is shown in Figure 1.1. In the broad context of computer aided software engineering (CASE), *KIEL* is currently being developed by the *Real-Time and Embedded Systems Group* at the Department of Computer Science, Christian-Albrechts-University of Kiel, Germany. Accordingly, the recursive acronym *KIEL* stands for "Kiel Integrated Environment for Layout." The tool's main goal is to enhance the intuitive comprehension of the behavior of the system under development (SUD). While most available statechart development tools merely offer a static view of the SUD during simulation, *KIEL* provides a more dynamic visualization [89]. At each simulation step, a new view of the chart is shown with the currently active states expanded and highlighted. The inner states of nonactive hierarchical states are collapsed to hide less important, potentially distracting details. Fundamental components of *KIEL* are a tree data structure that represents the topology of a statechart and an automatic layout mechanism that generates graphical representations of statecharts. *KIEL* also features import functionality for models of various statechart dialects, *e. g.*, Safe

State Machines and Stateflow, as well as for system descriptions written in Esterel via a transformation to Safe State Machines [90].

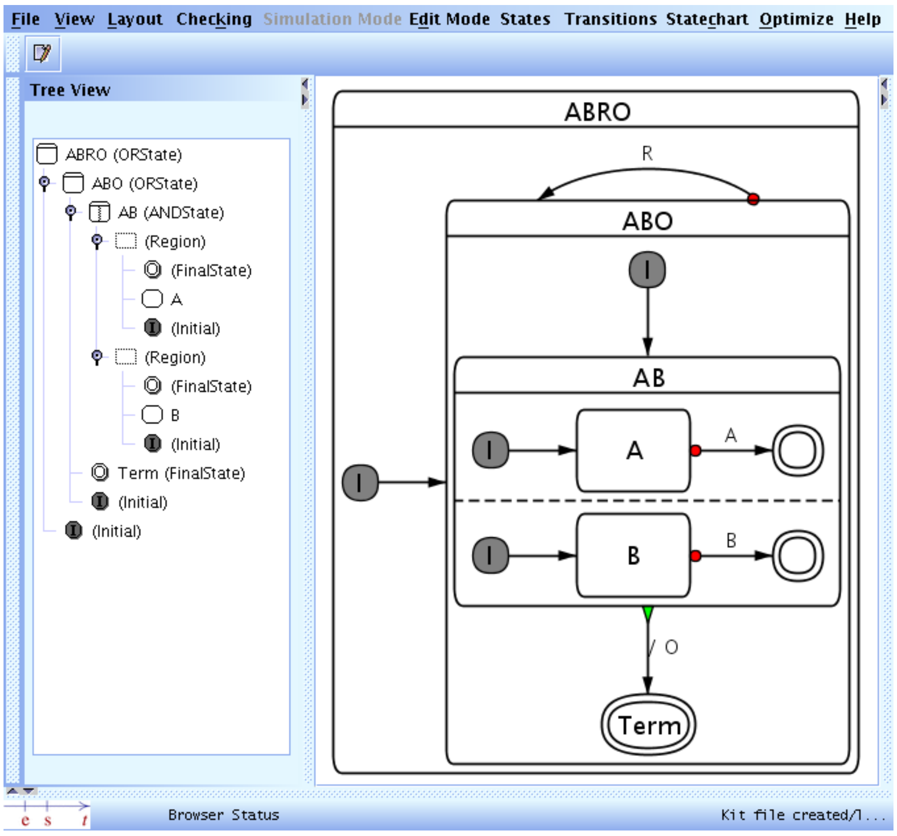


Figure 1.1: Screenshot of *KIEL* showing the example statechart ABRO. The tree data structure is shown on the left, the classical statechart view on the right.

This thesis is organized as follows: A thorough introduction of statecharts is given in Chapter 2. Thereafter, as the developed statechart checking framework is based on automated theorem proving, particularly the *satisfiability modulo theories* problem, Chapter 3 presents the respective theoretical background and an overview of available solving tools. Chapter 4 discusses automated software error prevention, focussing on the theoretical foundation and the practical state-of-the-art. In doing so, the particulars of style checking in textual computer programming (Section 4.2) as well as style checking in statecharts (Section 4.3) are addressed. Next, the definition of a comprehensive statechart style guide is presented in Chapter 5. Implementation details of the corresponding automatic checking framework as a plug-in for *KIEL* are described in Chapter 6. Finally, Chapter 7 concludes this thesis by critically discussing its results.

Chapter 2

Statecharts

Much has happened in the world of software and systems engineering since Harel introduced statecharts in 1987 in his seminal publication *Statecharts: A Visual Formalism for Complex Systems* [44]. Harel begins by recognizing “the existence of a major problem in the specification and design of large and complex reactive systems.” The problem, as he points out, originates from “the difficulty of describing reactive behavior in ways that are clear and realistic, and at the same time formal and rigorous” [44, page 231]. Although specialized programming languages, *e. g.*, Esterel, are forthcoming in the field of reactive systems, a gap between specification and design aims on one side and available methods on the other side exists. This gap stands in contrast to the development of transformational systems, for which excellent methods of decomposition, both coherent and rigorous have long existed.

Unlike a transformational system, which essentially computes a function, transforming input data into output data, a *reactive system* is characterized by continuously interacting with its environment, *i. e.*, responding to external and internal stimuli. In essence, Harel defines the behavior of a reactive system as “the set of allowed sequences of input and output events, conditions, and actions, perhaps with some additional information such as timing constraints” [44, page 232]. The fact that this set of sequences, which is often large and complex, could not be elegantly decomposed using traditional methods, hindered the systematic and efficient development of reactive systems. Here, statecharts, which are described in the following, using a bottom-up approach, enter the picture.

2.1 Emergence of Statecharts

A priori, the concept of states and events suggests itself as a rather natural medium for capturing the dynamic behavior of complex systems. A *state transition*, which takes the general form “when event *E* occurs while the system is in state *X*, if condition *C* is true at the time, the system transfers to state *Y*,” constitutes the basic element of this description. The systematic aggregation of such state-transitions forms a finite state machine.

2.1.1 Finite State Machines

A *finite state machine* (*FSM*), first introduced in the early 1950s by Rabin and Scott [92], is the oldest known formal, technical model of sequential behavior. It is based on the conjunction of Turing machines [106] and McCulloch’s and Pitts’ neural-network models [70]. Built on these concepts, FSMs are composed of states and transitions. “A state is a distinguishable, disjoint, orthogonal ontological condition that persists for a significant period of time” [30, page 50]. This means that a state can be clearly distinguished from other states, a system must be in exactly one state at all times, and states do not overlap. Transitions indicate a state change in response to an event of interest.

An FSM can be represented by a *state-transition diagram* (or *state diagram*, for short), as illustrated in Figure 2.1. State diagrams are directed graphs, with nodes representing states and edges labeled with *triggering* events and *guarding* conditions representing transitions.

A pure FSM, also called *acceptor* or *recognizer*, is defined as a quintuple $(Q, \Sigma, \delta, q_0, F)$, where

Q	is a finite set of *states*,
Σ	is the *input alphabet*,
$\delta : Q \times \Sigma \rightarrow Q$	is the *transition function*,
$q_0 \in Q$	is the *initial state*, and
$F \subseteq Q$	is the set of *final states*.

There are two basic ways to extend the above model into a *transducer* with an output capability:

According to the Moore model [73], output is associated with states. A *Moore machine* is a septuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0, F)$, where $Q, \Sigma, \delta, q_0, F$ remain as defined above,

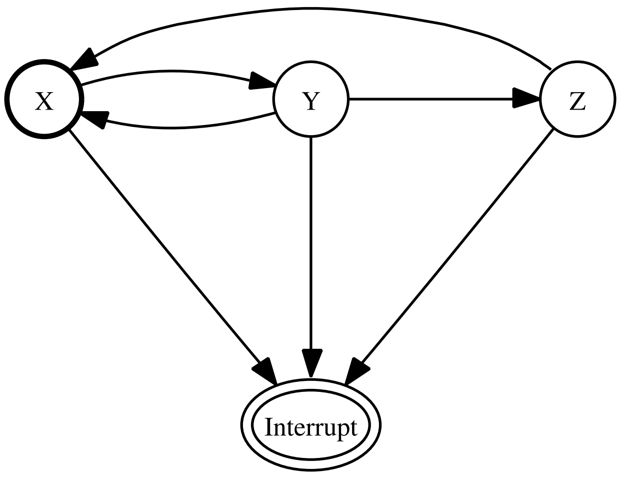


Figure 2.1: State transition diagram representing a simple finite state machine, with initial state "X" and final state "Interrupt."

$$\begin{array}{ll} \Delta & \text{is the } \textit{output alphabet}\text{, and} \\ \lambda : Q \to \Delta & \text{is the } \textit{output function}. \end{array}$$

According to the Mealy model [71], output is associated with transitions. A *Mealy machine* is also a septuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0, F)$, where $Q, \Sigma, \Delta, \delta, q_0, F$ are defined as for Moore machines, but

$$\lambda : Q \times \Sigma \to \Delta \quad \text{is the } \textit{output function}.$$

Output generally takes the form of an uninterruptible activity, called an action, to be performed before the system will react to any new events.

2.1.2 Application of FSMs: Shortcomings and Desiderata

Finite state machines, as presented above, have several critical shortcomings when employed for modeling complex reactive systems. Following a naïve approach of describing such systems, one would be faced with an unmanageable, exponentially growing multitude of states, as noted by Parnas as early as 1969 [86]. This is due to the "flat," unstratified nature of state diagrams, making them uneconomical concerning the number of transitions, *e. g.*, required for a high-level interrupt (cf. Figure 2.1), and even more uneconomical concerning the number of states w. r. t. parallel composition. Furthermore, state diagrams are inherently sequential, so that parallelism cannot be represented in a natural way [cf. 44].

To avoid the explicit representation of all combinations of states and transitions, a modular, hierarchical, and well-structured modeling strategy seems highly desirable. This strategy should provide more general and flexible concepts, including (1) a bottom-up aggregation of states into super-states as well as a top-down refinement of states into sub-states, which together results in a hierarchical organization of super-states, states, and sub-states; (2) the identification of independence or orthogonality of states; and (3) a generalization of single-event transitions into condition-guarded boolean multi-event ones.

To illustrate these concepts, consider the following example of a television set (numbers refer to the points mentioned above): (1) in all on-states (TV-program or videotext), when the power-button is pressed, the television switches off, and conversely, the television's on-state consists of TV-program and videotext; (2) the television's volume (or mute) is independent of the channel; (3) the device can be turned on by pressing either one of the channel buttons or the power button.

2.1.3 Harel Statecharts

Built on these desiderata, *i. e.*, enriching a combination of Moore and Mealy machines with hierarchy (depth) and orthogonality (parallelism), Harel developed an innovative visual formalism, introduced as statecharts [44]. *Statecharts* constitute an extension of conventional state diagrams by and/or decomposition of states, additionally including inter-level transitions and broadcast-communication between concurrent components, as well as encouraging 'zoom' capabilities for varying levels of abstraction—or, putting it in Harel's words,

> "statecharts = state-diagrams + depth + orthogonality +
> broadcast-communication" [44, page 233].

In addition, the modeled system may hold and modify data and also exchange this data with its environment. In statecharts, this data is represented by events and condition variables. Thus, Harel's original definition should be augmented to read

> statecharts = state-diagrams + depth + orthogonality +
> broadcast-communication + data [46].

The central components of this formalism are state-transitions, which model possible pathways to transfer from a current state into a next state under certain conditions, so-called transition predicates. There may be several outgoing transitions from one state, which are evoked in each case by their own predicates.

2.2 Statechart Dialects

Harel's original semantics is implemented in Statemate [45]. Besides, many different so called statechart *dialects* have evolved. In 1994, von der Beeck compared 21 of these dialects [12], each with somewhat different syntax and semantics. The Object Management Group (OMG) has incorporated an implementation-independent, overarching definition of statecharts into the UML specification [82]. In addition to this specification, the following three statechart implementations are considered in this thesis:

Statechart Dialect	Product Name	Product Vendor
Stateflow	MATLAB Simulink/Stateflow	The MathWorks
Statemate	Statemate	I-Logix
Safe State Machines	Esterel Studio	Esterel Technologies

All statechart dialects exhibit their own syntactic and semantic peculiarities of which the system modeler must be aware, especially since even identical syntax may have different semantics in different dialects. One of the most profound distinguishing characteristics of a particular statechart dialect is its notion of time. Essentially, two distinct models of time exist: The *synchronous* model of time assumes that a system executes a single step every instant or time unit, reacting to all external changes that have occurred since the completion of the previous step. The *asynchronous* model of time, on the other hand, assumes that a system reacts whenever an external change occurs, allowing several steps to take place within a single time unit [12].

The most prominent characteristics of the aforementioned dialects are described below:

UML state machines

- Currently, no immediate implementation of UML state machines exists. However, a growing group of UML modeling tools has committed itself to partial, but expanding implementations.

- In terms of syntactic scope, UML state machines represent a superset of a variety of statechart dialects. Conversely, many statechart implementations claim UML conformance. The semantics, on the other hand, are in many cases merely sketched in the UML standard.

Stateflow

- Stateflow charts are always embedded within a Simulink model. The charts may be run synchronously at the same rate as the surrounding Simulink model or asynchronously by selectively injecting events from Simulink when required [68].
- Stateflow provides a rudimentary static checking functionality.

Statemate

- Statemate was the first executable implementation of statecharts, adhering closely to Harel's original semantics.
- Statemate's micro-step semantics is a variant of the asynchronous model of time [45].

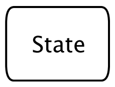
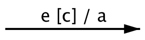
Safe State Machines

- Safe State Machines are built on a perfectly synchronous framework, *i. e.*, conceptually, concurrent processes perform computation and exchange information in zero time [14]. On the physical target-platform, this abstraction is translated into the guaranteed keeping of deadlines.
- Safe State Machines do not allow inter-level transitions, as they tend to disturb the desired encapsulation.
- Safe State Machines feature clear and explicitly defined semantics, facilitating strictly formal behavioral descriptions.
- A non-commercial, but slightly limited variant of Safe State Machines, named SyncCharts, has been developed by André [1].

2.3 Graphical Notation of Statecharts

So far, the theoretical concepts of statecharts have been introduced. To complete the formalism, we will now turn to the equally important aspect of visualization. In order

Table 2.1: Basic statechart elements: state and transition.

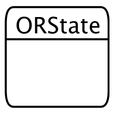
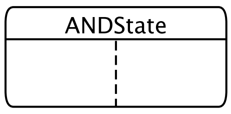
Statechart Element	Notation	Remarks
State		This most basic element, in unison with its name, represents the current state of the system. Moore-like actions, to be performed while entering, residing in, or exiting a state, may be specified.
Transition		A transition models a possible pathway to transfer the system from the current state into a next state, given the presence of an event of interest and the adherence to a certain condition. Taking the transition also triggers a specified action.

to avoid showing the graphical notation of all aforementioned statechart dialects, the notation of the UML specification [82] is shown. Additionally, the notation of Safe State Machines [35], which is used in the examples in Chapter 5, is given for elements that differ considerably from the UML standard.

Traditionally, trees and other line-graphs have been used for capturing depth and hierarchy. In these, however, the geometrical features of lines and points contribute no additional meaning. Drawing from the more general concepts of Euler circles, Venn diagrams, and hypergraphs, Harel suggested the use of rounded rectangles to represent states at any level, *i. e.*, (simple) states that have no sub-states as well as hierarchical (or composite) states [44]. The hierarchical relation is expressed using encapsulation of states within higher-level states. In Harel's notation, arrows, representing transitions, are allowed to originate and terminate at any level and are labeled with a triggering event and a parenthesized guarding condition. In most statechart dialects, both transition events and conditions are optional. Table 2.1 illustrates the notation of states and transitions.

As shown in Table 2.2, two distinct types of composite states, facilitating the specification of hierarchy, exist: XOR as well as AND. Within an XOR-state, exactly one of the contained states is active at any given time—still, such states are commonly referred to as OR-states. In contrast, within an AND-state, all orthogonal components are ac-

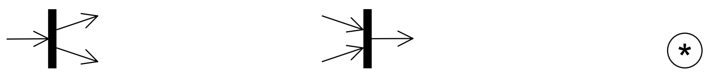
Table 2.2: Statechart elements: composite states.

Statechart Element	Notation	Remarks
OR-State		Exactly one of the contained states is active at any given time. In general, transitions originating from encapsulating states have higher priority.
AND-State		Each orthogonal component is called a region and has the semantics of an OR-state. All regions are active simultaneously.

tive simultaneously. AND-states constitute Harel's desired "natural way" to represent the decomposition of a system into its physical subsystems [44]. In general, transitions originating from encapsulating states have higher priority.

In addition to these basic statechart elements, final states and various pseudo-states are described in Table 2.3. Here, the notation of the UML standard as well as that of Safe State Machines (SSM) is shown. Pseudo-states are different from the previously introduced "normal" states in that a system can never actually reside in a pseudo-state.

The UML standard further defines Fork, Join, and Synch pseudo-states, shown in Figure 2.2. The exact semantics of these constructs are, however, beyond the scope of this paper.

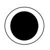
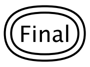
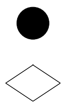
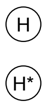
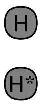


(a) Fork pseudo-state. **(b)** Join pseudo-state. **(c)** Synch pseudo-state.

Figure 2.2: Additional pseudo-states defined in the UML standard.

Moreover, as Safe State Machines do not allow inter-level transitions, alternative means for termination are required. These are illustrated in Table 2.4. For instance, an AND-state is exited by "normal termination" when all parallel components have reached their respective final states. Additionally, the relative priorities are given for the three types of transitions.

Table 2.3: Advanced statechart elements: final state and selected pseudo-states.

Statechart Element	Notation		Remarks
	UML	SSM	
Final State			Identifies the termination of the encapsulating composite state.
Initial Connector			Typically, when any composite state becomes active, the mandatory initial connector within is activated (see History Connector below for exceptions).
Choice Connector/ Conditional Connector/ Junction Connector			These pseudo-states are meant for uncluttering and economizing statecharts. Variying nomenclature, notation, and semantics exist in different statechart dialects.
History Connector			When a composite state is re-entered through its History Connector (H), the previous inner state is restored. Deep History Connectors (H*) even restore the states within all contained hierarchical levels.

2.4 Suitability of Statecharts for Modeling Reactive Systems

System modeling can be considered as a transition from ideas and informal descriptions to concrete descriptions that use fixed concepts and predefined terminology. Statecharts are embedded within such a description, capturing the system specification organized into three views, or projections, of the system [46]: (1) the *functional view*, encompassing the "what" of the system's capabilities, including its processes, inputs, and flow of information; (2) the *behavioral view* (statecharts), capturing the "when", *i. e.*, the system's behavior over time, including the control and timing behavior and thus providing

Table 2.4: Safe State Machines extensions of the statechart notation.

Statechart Element	Notation	Remarks
Normal Termination	▷——→	Lowest priority; automatically enabled when the originating composite state terminates. For an OR-state this is the case when an encapsulated final state is reached; for an AND-state it is the case when final states are reached in all encapsulated regions. Normal terminations do not support triggers or conditions.
Weak Abort	——→	Medium priority; when enabled, the originating state is left after it has completed all its activities associated with the current instant.
Strong Abort	●——→	Highest priority; when enabled, the originating state ist left immediately.
Suspend Connector	Ⓢ	"Always-active" pseudo-state; when one of the departing, actionless transitions is enabled, the composite target state is immediately (strongly) suspended, *i. e.*, execution within the state is paused for the current instant.

insight into causality, concurrency, and synchronization; and (3) the *structural view*, depicting the "how" of the system's structure in terms of subsystems, modules, or objects, and its communication links. While the former two views provide the *conceptual model* of the system, the structural view is considered to be its *physical model*, because it is concerned with the various aspects of the system's implementation. As a consequence, the conceptual model usually involves terms and notations borrowed from the problem domain, whereas the physical model draws more upon the solution domain.

Statecharts are well-fit for modeling reactive systems, as they emphasize not only "good" modeling in system development but also "easy" execution, debugging, analysis of the resulting models, and translation into working code for software and/or hardware [46]. With the support of a specialized modeling environment, such as Esterel Studio or MATLAB Simulink/Stateflow, a model can be executed, *i. e.*, simulated, by emulating the environment of the system under development and letting the model react accordingly. In particular, a model can be simulated interactively (step-by-step) or by batch execution. The modeling environment generates an animation of the diagrams by highlighting currently active states and activities. The possibility to simulate only part of a model allows for early testing. Furthermore, external code may be attached to a model to produce input stimuli or simply to augment the model. After a model has been constructed, successfully simulated, and satisfactorily tested, it can be translated automatically—with help of the development environment—into high-level programming language code (*e. g.*, C, Ada) or into a hardware description (*e. g.*, VHDL).

In summary, statecharts provide a powerful set of communication and abstraction mechanisms for structuring a reactive program as a well-scaling collection of interacting components [2]. In conjunction with a progressive development environment, they also permit to simulate the modeled system behavior. Due to the overarching development process as well as their strong intuitiveness, statecharts certainly surpass textual reactive system descriptions.

Chapter 3

Automated Theorem Proving

From the ancient Greek philosophers' syllogisms to present-day automated theorem proving, the field of logic and reasoning has undergone a significant development. Hence, the chapter is commenced by a historical introduction, which traces automated theorem proving back to its mathematical roots at the early beginning of the 20th century. This is followed by a detailed presentation of the validity problem and its dual, satisfiability. The satisfiability problem is then generalized to yield the satisfiability modulo theories (SMT) problem. After a presentation and characterization of several SMT solvers, the chapter is concluded by the selection of a tool, most suitable for solving SMT problems arising from the semantic robustness analysis of statecharts.

3.1 Historical and Introductory Remarks

As part of automated reasoning, the area of *automated theorem proving* (*ATP*) deals with the development of efficient algorithmic methods for finding proofs for mathematical formulae that are true and providing a counterexample for those that are not true. This problem of proving the *validity* of a theorem or formula traces back to Hilbert's *tenth of 23 mathematical problems*, which he originally presented at the International Congress of Mathematicians in Paris in 1900 [47]. The *tenth problem* addresses whether or not all Diophantine problems[1] have a solution. Seventy years later, Matiyasevich proved that this is not the case [69].

[1]A *Diophantine equation* is a polynomial equation that only allows integer-variables. *Diophantine problems* have fewer equations than unknown variables and involve finding integers that satisfy all equations.

Depending on the underlying logic, the problem of deciding the validity of a theorem varies from trivial to impossible. For propositional logic[2], the problem is decidable, but NP-complete, *i. e.*, only exponential-time algorithms are believed to exist. When dealing with first-order logic[3], there are two historical results of central importance: Gödel's completeness theorem [39] and Church's undecidability of validity [23].

Gödel's completeness theorem asserts that there are logical calculi in which every true formula is provable. An example of this is Presburger's first-order theory of the natural numbers[4], which was introduced in 1929. It is a model of arithmetic containing addition, but not multiplication [88]. Presburger-arithmetic is sound and complete, *i. e.*, it does not contain contradictions and every statement can either be proven or disproven. This is in contrast to, *e. g.*, Peano's undecidable first-order theory of the natural numbers, which also includes multiplication.

Church's undecidability of validity theorem was formulated in 1936 and solved the famous *Entscheidungsproblem*, a generalization of Hilbert's *tenth problem*. Church's theorem asserts that there is no decision procedure (that always terminates) for deciding whether a formula in first-order logic is valid. In the same year, Turing independently also proved the *Entscheidungsproblem* undecidable [106]. While Church's proof received less attention, Turing's approach, solving the halting problem for Turing machines along the way, became highly influential. Still, both proofs separately imply that the general first-order theory of the natural numbers expressed in Peano's axioms cannot be decided with any algorithm.

Combining Gödel's and Church's theorems, it follows that any valid first-order theorem can eventually be proven, given unbounded resources. Invalid statements, on the other hand, cannot always be recognized. In these cases, first-order theorem provers fail to terminate. Nevertheless, first-order theorem proving is one of the most mature subfields of

[2]Propositional logic is a logic in which the only objects are propositions, *i. e.*, objects which themselves have truth values. Variables represent propositions, and there are no relations, functions, or quantifiers except for the constants *true* and *false*. Typically, the connectives are $\wedge, \vee, \neg$, and $\rightarrow$ (representing negation, conjunction, disjunction, and implication, respectively) [38].

[3]First order logic, also referred to as predicate logic or predicate calculus, is an extension of propositional logic with separate symbols for predicates, subjects, and quantifiers ($\forall$ and $\exists$). For example, where propositional logic might assign a single symbol P to the proposition "All men are mortal," predicate logic can define the predicate $M(x)$, which asserts that the subject x is mortal and bind x with the universal quantifier: $\forall x, M(x)$.

[4]With the tools of first-order logic it is possible to formulate a number of theories, with explicit axioms or by rules of inference, that can themselves be treated as logical calculi. Arithmetic is the best known of these; another example is set theory.

automated theorem proving, as the logic is expressive enough to allow the specification of arbitrary problems, often in a reasonably natural and intuitive way. Despite its limited decidability, practical theorem provers can solve many hard problems in first-order logic. Moreover, fully automated systems can be constructed for a number of sound and complete calculi, *e. g.*, the abovementioned Presburger-arithmetic. However, rather than proving a formula's validity directly, the undermentioned most influential and powerful ATP algorithms have always sought after determining a formula's *satisfiability*—a dual problem [38].

Since the 1994 Pentium FDIV-bug[5], automated theorem proving has become a significant part of industrial integrated circuit design and verification. Industry leaders such as AMD, IBM, Intel, Motorola, and Sun use automated theorem proving to verify the operations of their modern microprocessors [54]. Program analysis through theorem proving has also become part of the development of large-scale safety-critical systems, *e. g.*, designed for the Airbus A380 or Ariane 5 [51].

3.2 Satisfiability: Theory and Algorithms

Two distinct forms of the satisfiability problem are presented below: the classic *boolean satisfiability* problem and the more recent and more general *satisfiability modulo theories* problem. In both cases, it is true that a proposition A is satisfiable, *iff* (if and only if) $\neg A$ is not valid. This was formally proven by Gallier [38, page 43].

3.2.1 Boolean Satisfiability Problem

As the dual of propositional theorem proving, the *boolean satisfiability* (*SAT*) problem is the simplest form of theorem proving. It holds an important place in various areas of computer science, including artificial intelligence, automatic verification of programs, and hardware circuit synthesis [110]. Alongside, SAT is the first problem which has been shown to be *NP*-complete [53, original proof in 24]

Boolean satisfiability is the problem of determining whether there is an assignment of truth values to the variables of a formula of propositional logic for which that formula

[5]FDIV is the x86 assembly language instruction for floating point division. Intel officially refers to the bug as the "Floating Point Flaw in the Pentium Processor" [see 50, the official Intel white paper on the "Statistical Analysis of Floating Point Flaw"].

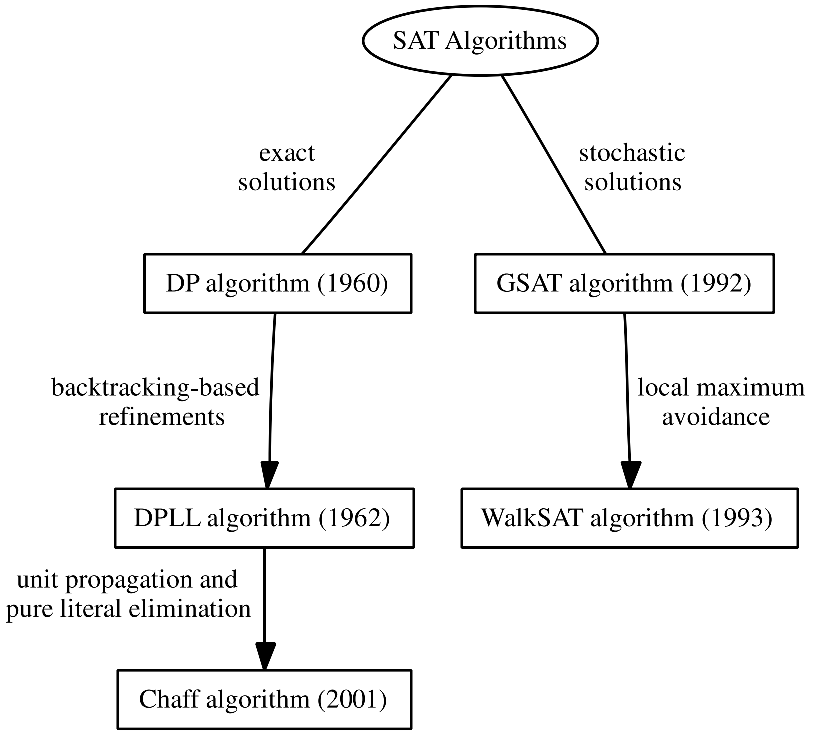


Figure 3.1: Historical overview of algorithms for solving the boolean satisfiability problem.

evaluates to *true*. More formally: Let P be a fixed finite set of propositional symbols; a truth assignment M is a set of literals such that $\forall p \in P, \{p, \neg p\} \subseteq M$. A clause C is true in M if at least one of its literals is in M. Then a formula F is satisfied or modeled by M, denoted by $M \models F$, if all of F's clauses are true in M. F is unsatisfiable *iff* it has no models [81]. Formulae are generally required to be in conjunctive normal form (CNF), *i. e.*, a conjunction of clauses, where a clause is a disjunction of literals[6].

As shown in Figure 3.1, modern-day exact algorithms for solving SAT problems can be traced back to the vintage 1960 Davis-Putnam (DP) [27] and 1962 Davis-Logemann-Loveland (DPLL) [28] algorithms. Over the years, a vast research community has streamlined and upgraded the DPLL algorithm to yield enormously powerful tools, *e. g.*, Chaff [74], capable of solving problems of tremendous size: more than one million variables and more than 10 million clauses. Moreover, stochastic algorithms, having the power to outperform Chaff by an order of magnitude on certain types of problems, have been developed in the 1990s [95, 96].

[6]A literal is an atom or the negation of an atom; atoms are the most elementary building blocks of formulae, with respect to the underlying logical calculus.

3.2.2 Satisfiability Modulo Theories Problem

Since many satisfiability problems arising from real-life applications cannot be easily expressed in propositional logic, a more universal, problem-oriented, though still decidable, underlying theory is required [81]. For example, the properties of timed automata are naturally expressed in difference logic, where formulae contain atoms of the form $a-b \leq k$, which may be interpreted with respect to a background theory T of real or integer linear arithmetic. Other common background theories, for which specialized decision procedures exist, are the theories of lists and arrays, bit-vectors, and the logic of equality and uninterpreted functions. Since arithmetic, lists, and arrays are ubiquitous in computer science, it seems very natural to consider satisfiability problems *modulo* the theory T of these data structures. Background theories are predominantly expressed in classical first-order logic with equality. Only those theories, for which satisfiability is actually decidable, are considered, though [5].

Problems may be composed of the conjunction of thousands of clauses like

$$\neg p \quad \vee \quad y = g(x-z) \quad \vee \quad send(s, g(x-7)) \leq a,$$

where $\neg p$, $y = g(x-z)$, and $send(s, g(x-7)) \leq a$ are literals. The *satisfiability modulo theories* (*SMT*) problem, the dual of first-order theorem proving, determines the satisfiability of such a formula F' with respect to a certain background theory T: Given F', determine whether F' is T-satisfiable, or in other words, whether there exists a model of T that is also a model of F'. Similar to the SAT problem above, F' is satisfied by a truth assignment M', denoted by $M' \models F'$, *i. e.*, M' is a T-model of F', if all clauses of F' are true in M'. F' is T-unsatisfiable *iff* it has no models in T. The set P' over which formulae are built is a fixed set of variable-free first-order atoms [81].

Two fundamentally different approaches for solving SMT problems exist today: the *eager* and the *lazy* approach. The *eager* approach converts the input formula within a single step into an equi-satisfiable propositional CNF formula. Although sophisticated ad-hoc translations exist for many theories, translators or SAT solvers run out of time or memory on many practical problems. If successful, this procedure is still several orders of magnitude slower than the undermentioned *lazy* approach to SMT [81]. The widely successful *lazy* approach to SMT was introduced by Armando and Giunchiglia in 1993 [3]. The basic idea is that of managing the search for a model via an existing efficient SAT algorithm, *e. g.*, the DPLL algorithm, and delegating first-order reasoning to an appropriate T-solver [4]. Accordingly, each atom of a formula F' to be checked for T-satisfiability is simply regarded as a propositional symbol, "forgetting" temporarily about the theory T. In this sense, the boolean abstraction of F' is checked for propositional satisfiability by a SAT solver. If the SAT solver reports unsatisfiability, then F' is T-unsatisfiable. Otherwise, the SAT solver returns a propositional model M^* of F'. This model, a conjunction of literals, is checked by a T-solver. If it is found T-satisfiable, then M^* is a T-model of F'. Otherwise, the T-solver builds a clause C which precludes M^*. The clause C could simply be the negation of M^*, but a smaller clause will lead to a more efficient algorithm. Subsequently, the SAT solver is started again with $F' \wedge C$ as input. This process is repeated until the SAT solver finds a T-satisfiable model or returns unsatisfiable [81].

3.3 Selection of an SMT Solver for Semantic Robustness Analysis of Statecharts

As mentioned in Chapter 1, the semantic robustness analysis of statecharts presented in Section 5.3, is based on automated theorem proving, particularly the satisfiability modulo theories problem. Accordingly, an SMT solver needs to be selected for this purpose. Therefore, a thorough survey of today's general-purpose SMT solvers—all currently under active development—was conducted. A brief characterization followed by a qualitative assessment of the considered tools is given below. On this basis, the most suitability SMT solver is finally chosen for integration into the *KIEL* framework.

3.3.1 Overview of SMT Solvers

Of the tools presented below, UCLID is the only SMT solver that has not participated in the first *Satisfiability Modulo Theories Competition* [10, 94], held in 2005, where SMT solvers competed against each other in seven categories covering different quantifier-free background theories. The categories are (1) uninterpreted functions, (2) real difference logic, (3) integer difference logic, (4) uninterpreted functions and integer difference logic, (5) linear real arithmetic, (6) linear integer arithmetic, and (7) arrays, uninterpreted functions and linear integer arithmetic. Not every tool necessarily competed in every category, though. According to the competition's organizers, each category comprised of approximately 50 SMT problems, ranging from "easy" to "hard." The results of the linear integer arithmetic category are shown in Figure 3.2 [11]. This category is of particular interest w. r. t. the semantic robustness analysis of statecharts.

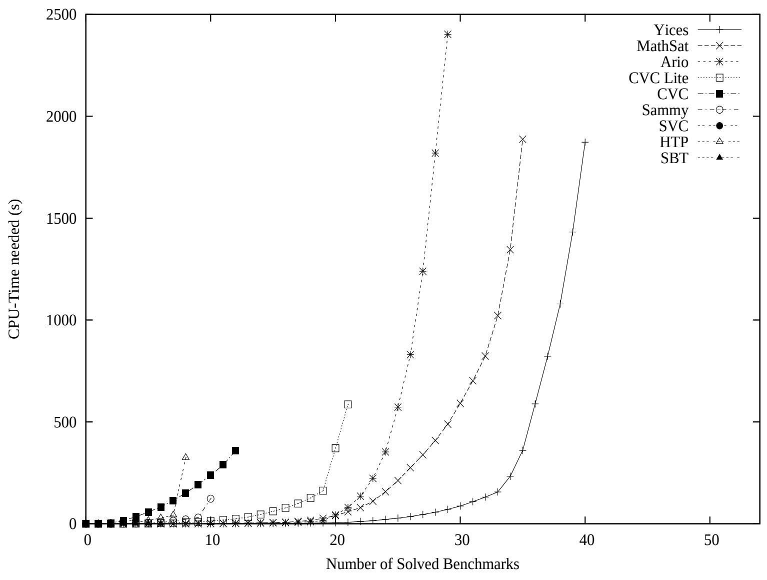


Figure 3.2: Results of the linear integer arithmetic category at the 2005 *Satisfiability Modulo Theories Competition* [11].

UCLID: Developed at Carnegie Mellon University, UCLID [18, 97] is the only (moderately) successful implementation of an SMT solver that follows the *eager* approach to SMT. The tool is capable of verifying formulae over equality and uninterpreted functions and integer linear arithmetic. UCLID is implemented in Moscow ML as a stand-alone decision procedure.

CVC Lite: In contrast to all other tools presented here, CVC Lite (CVCL) [8, 25] determines the validity of a formula, not its satisfiability. The open-source C++ implementation of decision procedures for a subset quantifier-free first-order logic is being developed by the Formal Verification Group at Stanford University and the Analysis of Computer Systems group at New York University. As a second reimplementation, CVC Lite supersedes the Cooperating Validity Checker (CVC) [99], which is in turn the successor the Stanford Validity Checker (SVC) [9]. The tool combines decision procedures for a multitude of theories including arrays, aggregate datatypes such as records and tuples, real and integer linear arithmetic, and bit-vectors. Uniquely, CVCL can be used in two modes: as a C/C++ library, and as a command-line executable (implemented as a command-line interface to the library). CVC Lite participated in all seven categories at the 2005 SMT competition [94]; scores were generally in the mid-field.

DPLL(T): As part of the *BarcelogicTools*, DPLL(T) [7, 81] is developed at the Technical University of Catalonia in Barcelona. The tool's modular implementation utilizes a general DPLL(X) engine—a slightly modified implementation of the DPLL algorithm [27, 28]—that is instantiated with a specialized theory solver T. DPLL(T) makes extensive use of the T-solver's capabilities, using it *a priori* to guide the SAT solver to a correct model. The highly efficient C-implementation currently supports difference logic over the integers and reals, equality and uninterpreted functions, as well as combinations of these theories. The tool won all four categories it participated in at the 2005 SMT competition [94].

MathSAT: MathSAT [17, 66], built on top of the MiniSAT [33] DPLL implementation, is a product of the Trentino Cultural Institute in Trento, Italy. The tool, programmed in C++, can decide the SMT problem for various background theories, including equality and uninterpreted functions, linear arithmetic over the integers and reals, as well as combinations of these. Contrary to most other SMT solvers, MathSAT accepts input formulae in non-clausal form, that may even include *if-then-else* constructs over non-boolean terms. A pre-processor translates the formula into conjunctive normal form in linear-time. At the 2005 SMT competition [94], MathSAT participated in six of the seven categories, receiving average scores.

Yices: One of several SMT solvers developed at the Computer Science Laboratory of SRI International (formerly the Stanford Research Institute), the prototype Yices [109] performed surprisingly well at the 2005 SMT competition [94], coming in first in two categories and second in the remaining five categories. Implemented in C++, the tool decides satisfiability for real and integer linear arithmetic, equality and uninterpreted functions, recursive datatypes, tuples and records, arrays, and lambda expressions.

3.3.2 Selection of SMT Solver

Table 3.1 provides an overview of the characteristics and availability of the aforementioned SMT solvers. In order to decide on the most suitable tool for solving the SMT problems arising in the context of semantic robustness analysis of statecharts, the following considerations were made: The selected tool must integrate smoothly into the *KIEL* framework, which implies executability on Linux as well as on Windows (cf. Section 6.1). This immediately eliminates all solvers except DPLL(T) and CVC Lite.

Table 3.1: Classification of SMT solvers

SMT Solver	Approach	Integer Arithmetic	Real Arithmetic	Linux		Windows		Open-Source
				Bin	Lib	Bin	Lib	
UCLID	*Eager*	+	–	+	–	–	–	–
CVC Lite	*Lazy*	+	+	+	+	+	+	+
DPLL(T)	*Lazy*	+	+	+*	–	+*	–	–
MathSAT	*Lazy*	+	+	+	–	–	–	–
Yices	*Lazy*	+	+	+	–	–	–	–

*The executable expires after approximately two months.

Moreover, both of these tools are able to handle the required SMT discipline of difference logic over the integers and reals. Judging by the results of the 2005 SMT competition [11, 94], it seems that DPLL(T), the tool that won all four categories it participated in, is the clear choice. On the other hand, the benchmark assessment of the SMT competition must not be mistaken for a universally valid metric for these tools, though. In general, the interpretation of benchmarking data is extraordinarily difficult. For example, in the case of the 2005 SMT competition, tools were ranked according to their performance in solving *difficult* problems. In their report, however, the organizers mention that in some categories the problems must have been easier than in others, considering the number of correct answers [10]. In judging the results, one should be aware of the general difficulty of defining and identifying “hard” problems [22, 72].

Irrespective of such intricacies, great numbers of very small and simple SMT problems must be solved in the present work. Accordingly, the call to the decision procedure must be fast. Due to the minimal size of the arising problems, the actual time to decide the satisfiability is almost negligible. Here, CVC Lite has the upper hand due to its availability as a C/C++ library, allowing for efficient access. As *KIEL* is implemented in Java, the library cannot be called directly, though, but only via the Java Native Interface (JNI) [101], as presented in Section 6.2.

The finally deciding aspect, however, was that DPLL(T) is available only as an expiring executable. Since new versions of the tool are subject to significant changes, maintenance far beyond the scope of this thesis would be necessary. Accordingly, CVC Lite emerges as the tool of choice, with the additional bonus of open-source availability.

Chapter 4

Automated Software Error Prevention

The software engineering dream of error-free software delivered on time and within budget is almost as old as software itself. In 2003, Hoare re-proposed the goal of automatically verified software as a grand scientific challenge[1] for computing, originally suggested in 1969, but abandoned in the 1970s [48]. As automated software error prevention is the main theme of the present thesis, this chapter is concerned with exactly that topic: First, an introduction and disambiguation of style checking terminology within the context of automated software error prevention are given. Following, the state-of-the-art of style checking in textual computer programming is described, before turning to style checking in statecharts.

4.1 Introduction and Disambiguation of Terminology

Software error prevention encompasses a number of different techniques designed to identify programming flaws. It has thus become an important part of the certification process required for (real-time) safety-critical systems, since the failure of such systems—attributable to programming flaws—can often cause loss of human life or property. Numerous times during such certification processes, it has been observed

[1]Grand challenges in science or engineering are proclaimed in order to distinguish certain projects from the many other kinds of short-term or long-term research problems. A grand challenge "represents a commitment by a significant section of the research community to work together towards a common goal, agreed to be valuable and achievable by a team effort within a predicted timescale. . . . A grand challenge may involve as much as a thousand man-years of research effort, drawn from many countries and spread over ten years or more" [48, page 63].

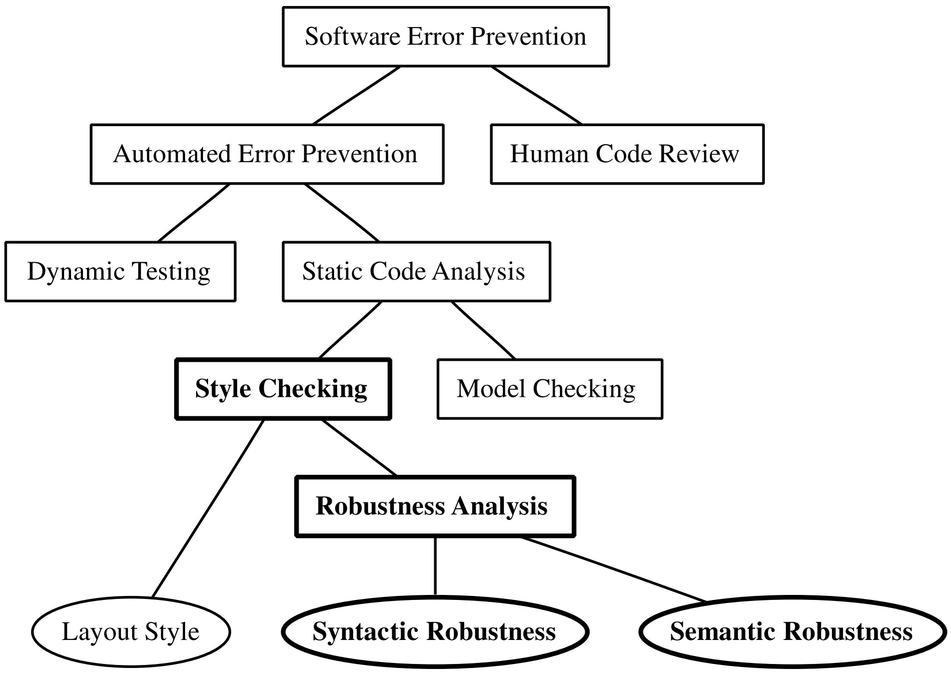


Figure 4.1: Software Error Prevention and its branches.

that “reviewers spent much too much of their time and energy checking for simple, application-independent properties, [distracting them] from the more difficult, safety relevant issues,” in this case stated by Parnas [87, page 155] about the certification of a nuclear power plant. The statement refers to human code reviews, which, though time-consuming and highly dependent upon expertise and concentration, may sometimes find conceptual problems that are impossible to detect automatically. 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.* Expert knowledge codified in tools is an additional benefit resulting from this automation.

Automated error prevention is commonly separated into dynamic and static methods [65], as outlined in Figure 4.1. *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. It thus allows for statements about all possible program executions rather than just one test-case execution. 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*. The primary application area of static code analysis is the verification of safety-critical computer systems. It is considered vital for defect detection during analysis and design. However, it should always be used in conjunction with (and not as a replacement for) software testing, *i. e.*, static code analysis is "necessary but not sufficient" [65, page 1]. It does certainly not replace runtime access controls, systematic testing, careful security assessments, and a solid high-level design [36].

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 later two are often collectively referred to as robustness analysis (see below). Layout style and robustness analysis are clearly distinguished within the realm of this thesis, as will be pointed out in the sections to follow. Style checking always requires the syntactic and semantic soundness of the code, as it typically contains guidelines on issues that do not directly affect correctness, but rather try to reveal common sources of errors: 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 [102], 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[2] *analysis*, as an important field of style checking, refers to "the process of checking a design against a [defined] set of design rules, with the objective of eliminating certain types of errors and enforcing sound engineering practices"[3] [40, page 17]. Robustness rules limit the general range of a given modeling/programming language, as they are entirely independent of what is being designed. 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 [42, 75]. Hence, robustness analysis is a preventive instrument and substantially draws from the knowledge and understanding of software implementation flaws [36].

[2]Robustness, from the Latin *robustus*, meaning oaken or strong, indicates a property of strength, vigor, or firmness, or being of strong form or construction [34].

[3]This use of the word robustness must be clearly distinguished from other connotations, *e. g.*, robust control or robust design. Robust control aims at designing controllers so that they function correctly not only for a well-known design model, but also for (real) systems, whose dynamics deviate within certain boundaries from the dynamics of the model. Robust design, where robust 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.

Coding standards, also called programming style guides, contain a list of rules used for style checking which have been gathered from common sense and collective engineering experience to describe conventions for writing source code in a certain programming language. A coding standard commonly consists of a set of programming rules (*e. g.*, “Do not compare two floating point numbers.”), naming conventions (*e. g.*, “Classes should start with capital C.”) and layout specifications (*e. g.*, “Indent 4 spaces.”). The development of coding standards is typically approached by first conducting a survey of available source material (*e. g.*, common programming experience), then filtering and adapting the compiled information into rules, and finally subjecting the resulting material to expert review and revision [cf. 76]. As rules are likely to originate from subjective preferences, some of the contents of a style guide may only be advisory, while some must be considered mandatory. Still, there are typically means by which the programmer can indicate the he/she is aware of the rule, but is infringing it intentionally.

To automate style checking, specialized *code checking tools* have been developed. Such tools analyze whether a piece of software adheres to a certain coding standard. Code checkers are commonly employed as the number of rules in a coding standard is usually too large for anybody to remember. Moreover, some context-sensitive rules demand inspection of several files at once and are thus very hard to check for humans. Although code checkers can reduce formal code review time considerably, it certainly does not mean that human code reviews are obsolete.

Several characteristics can be applied to distinguish available code checkers [104]:

Possibility to define new rules: Code checkers do not necessarily fully cover proprietary coding standards. As a consequence, it should be possible to define new rules, either via an API (complex to use, but powerful) or a graphical user interface (easy to use, but limited).

Integration in programming IDE: As most end-users of code checkers are software engineers, the code checker should fit smoothly into the used programming environment, preferably running in the background at all times.

Presence of command-line version: In order to be able to integrate the code checker into the software development process and/or into the programming IDE (if no immediate plug-in is available), the checker should be command-line based.

High performance: Code checkers inherently perform a thorough, hence time-consuming, syntactical and semantical analysis of source code. However, high performance is a key issue in end-user acceptance.

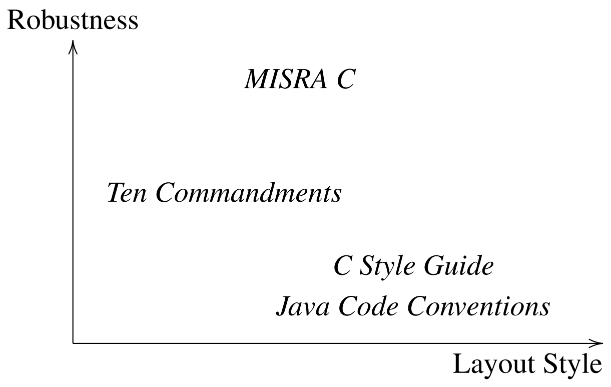


Figure 4.2: Classification of coding standards according to their emphasis of layout style vs. robustness.

4.2 Style Checking in Textual Computer Programming

In classical, *i. e.*, textual, computer programming, style checking has long found its way into the software development lifecycle. A multitude of coding standards and related code checking tools exits. Drawing from this large body of work, an exemplary selection of coding standards and checking tools is discussed in the following.

4.2.1 Coding Standards

Since programming style is often dependent on the actual choice of programming language, different coding standards exist for different programming languages. The most commonly applied coding standards for C include NASA's *C Style Guide* [29], MISRA's *Guidelines for the Use of the C Language in Critical Systems (MISRA-C:2004)* [75], and *The Ten Commandments for C Programmers* [98]; Java's dominant coding standard are Sun's *Code Conventions for the Java Programming Language* [102]. Figure 4.2 roughly classifies these standards according to their emphasis of layout style vs. robustness—a major distinction within style checking, as introduced in Section 4.1.

***C Style Guide*:** The *C Style Guide* [29] contains recommended practices and advice on style for programmers using the C language. These guidelines are based on generally recommended software engineering techniques, industry resources, and local conventions. They offer preferred solutions to common C programming issues, illustrated through examples of C code. The document recommends a style for writing C programs, where code with "good style" is defined as that which is "organized, easy to read, easy to understand, maintainable, [and] efficient" [29, page 1]. Guidelines for organizing the content of C programs, files, and functions are provided and the structure and placement of variables, statements, and comments is discussed.

***Guidelines for the Use of the C Language in Critical Systems*:** *MISRA C* [75], first published in 1998 with a strong focus on the automobile industry and later extended to cover a more general scope of safety-critical system development [75, 76], is the British Motor Industry Software Reliability Association's style guide for the C programming language, currently containing 121 required and 20 advisory rules on programming style, *i. e.*, forbidden layouts and allowed code structures, restrictions on which syntactic items of C should be used, stringent type checking rules, as strong typing is not part of C, but may help maintain a clearer design, and restrictions on ambiguous structures that can be subject to compiler interpretations and mistakes. Overall, these rules are aimed at three important aspects of the code, namely portability, reusability, and testability. Due to their comprehensiveness and maturity acquired over the past decade, the MISRA rules have become the established standard for the development of safety-critical systems all over the world.

***Ten Commandments for C Programmers*:** Even the hacker community has established its own set of coding conventions, the slightly quirky *Ten Commandments for C Programmers* [98]. In a list of ten rules, good programming standards in C are compiled, involving suggestions such as the usage of Lint (see below), avoiding null pointers, correct type casting, and so on. Albeit its queer appearance, the establishment of the standard illustrates the importance of sound programming practices even in the realm of freelance and open-source programming.

***Code Conventions for the Java Programming Language*:** Sun's *Code Conventions for the Java Programming Language* [102] are the conventions that the inventors of Java recommend for all programmers. The brief document contains rules on filenames, file organization, indentation, comments, declarations, statements, white space, naming conventions, programming practices, and also includes an example of "good" code. Although others exist, these conventions have established themselves over the years.

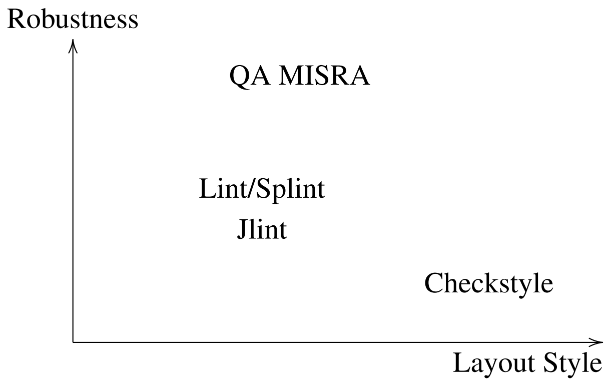


Figure 4.3: Classification of code checking tools according to their emphasis of layout style vs. robustness.

4.2.2 Code Checking Tools

Akin to coding standards, most code checking tools are programming language-specific. Available code checkers for C include Lint [52] (Sun Microsystems), its extension LCLint, now known as Splint [36] (University of Virginia), and QA MISRA [91] (Programming Research). Code checkers for Java are the open-source projects Jlint [6] and Checkstyle [20]. Figure 4.3 roughly classifies these code checkers according to their emphasis of layout style vs. robustness.

Lint and Splint: Originally developed and released in 1979 as part of Bell Labs' UNIX operating system, Lint [52] is nowadays distributed and maintained by Sun Microsystems. Lint extensively examines the source code of C programs for consistency and plausibility. According to the authors, Lint is intended to "encourage better code quality, clearer style, and may even point out bugs" [52, page 4]. Today, Lint has been superseded by is extension Splint [36], which comes with its own, extendable set of coding rules. Originally designed as a command-line tool, Splint can be efficiently integrated in state-of-the-art programming IDEs. In particular, the tool checks for unused variables, functions, function arguments, and return values, as well as uninitialized variables and unreachable fragments of code. It also enforces the type checking rules of C much more strictly and across file boundaries than most compilers.

Lint was introduced at a time when computing power was a rather limited resource. Compilers were designed with the supreme goal in mind to quickly produce fast code. The compilers were fast because they did not perform much checking, and the code was fast because the compilers did not generate many run time checks. Using Lint, programmers, for the first time, were able to protect themselves against producing error-prone code. Nowadays, many of the mentioned checks *can* be done by a modern compiler, if the user so desires, *e. g.*, `gcc -Wall` [37]. However, even today, a compiler is supposed to rapidly generate efficient code. Therefore, the compiler definitely should not do too much checking at compile time and should also not generate too many run-time checks. Furthermore, the compiler checks are neither customizable nor extendable. For example, `gcc -Wall` issues no warning for a statement such as `a[i] = b[i++]`, where the execution order is undefined. Rather, it is Lint's job to do so.

QA MISRA: As part of Programming Research's static analysis toolkit, QA MISRA [91] is a powerful, commercially developed, and industrially adopted solution for MISRA C compliance checking. The tool analyzes source code and detects constructs, which do not comply with the MISRA C rules. Rule violations are clearly identified in annotated source code. Warning messages usually include a suggestion of alternative MISRA compliant code as well as a cross-reference to the definition of the infringed rule. QA MISRA further allows users to tailor or add checks appropriate to individual company standards or other conventions. The tool, however, cannot be used outside of Programming Research's proprietary IDE.

Jlint: Jlint [6] performs syntactic checks and dataflow analysis on Java bytecode. The tool also builds a lock graph in order to find possible deadlocks in multithreaded applications. It is easy to run from the command-line or from within many programming IDEs, and it is very fast to execute. Jlint's downside is that its rule-set cannot easily be expanded.

Checkstyle: The extremely adaptable tool Checkstyle [20] efficiently verifies Java source code against any given or personal coding standard, naturally including Sun's Java code conventions. New rules can easily be added and flexibly managed. A command-line version as well as plug-ins for all common programming IDEs are available.

4.3 Style Checking in Statecharts

Statechart style checking is much less developed and less sophisticated as compared to style checking in textual computer programming. In modeling and analysis of the dynamics of (safety-critical) reactive systems, it is, however, all the more important that models are designed according to approved rules. This achieves not only a homogeneous modeling and layout of statecharts within projects, but also ensures 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 CASE tools can be simplified [43, 58, 59, 78, 80].

Although the commercial statechart modeling tools, Esterel Studio, MATLAB Simulink/Stateflow, and Statemate, have been supplemented with a number of consistency checks, Mutz and Huhn [80], von Hanxleden [43], and Kossowan [58] point to the following main deficiencies in style checking capabilities of statechart modeling tools: (1) Analyses are tool-specific. (2) Analyses are inalterable, *i. e.*, rules are neither extendable, individually selectable, nor parametrizable. Thus, the user has to adhere to the set of rules as implemented in the tool. (3) The scope of the rules checked by tools varies from rudimentary analysis to advanced model checking. (4) In Esterel Studio, for example, 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 actually reached by the control flow of the current simulation (non-static analysis). These shortcomings motivated many large industrial companies, where statecharts have long become an integral part of software development, to devise team- and domain-specific pattern collections in an effort to harmonize solutions for prevalent design problems [80]. Three such collections and four appendant checking tools are briefly characterized below. These checking tools as well as the style checker developed within the context of this thesis, the *KIEL Checking* plug-in, are roughly classified according to their emphasis of layout style vs. robustness in Figure 4.4.

Mint: The commercial MATLAB Simulink/Stateflow add-on Mint [93], distributed by the Ricardo Company, checks the style guide of the MathWorks Automotive Advisory Board (MAAB) [67]. The more than 100 MAAB rules, developed by Ford, Daimler Benz (now DaimlerChrysler), and Toyota in the late 1990s, primarily aim at achieving a consistent look-and-feel, enhancing readability, and avoiding common modeling errors. The rules go as far as specifying a certain background color for the Simulink environment and even give advice on project and model filenames. A few simple robustness rules regarding, *e. g.*, the scope of events, are also part of the checks. Mint features an interactive GUI frontend as well as an open API which allows for additional custom checks. The MAAB rules as well as Mint are specific to the MathWorks tool suite.

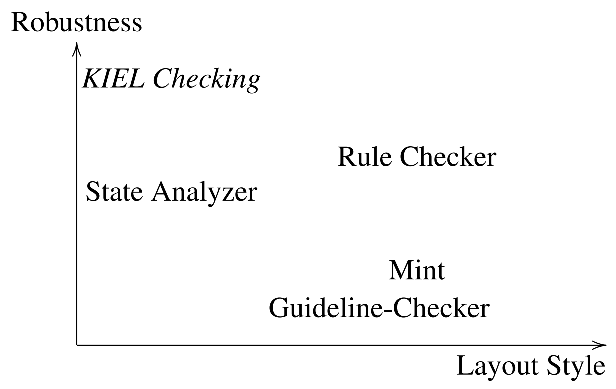


Figure 4.4: Classification of statechart style checking tools according to their emphasis of layout style vs. robustness.

Guideline-Checker: Intended as a no-cost/academic alternative to Mint, the Guideline-Checker results from a cooperation between DaimlerChrysler and the University of Applied Sciences in Esslingen [77]. The checking tool, which is based on the MAAB rules and coded in MATLAB, emphasizes modifiability and extensibility of the rule set. However, existing checks can only be modified and new ones added by directly manipulating the MATLAB code. Although the checked rules are individually selectable, they cannot be parametrized at runtime. 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” [77, page 26]. Sightly more complicated rules, *e. g.*, the uniqueness of state names, cannot be checked automatically. It has been further criticized that the tool only applies to the MathWorks family, with function queries and access to the internal representation mixed up [80].

State Analyzer: The State Analyzer, also developed within DaimlerChrysler’s R&D, is a prototypical software tool to check the “determinism” of statecharts [41, 58]. Performing an automated robustness analysis of requirements specifications, the tool verifies that for every state, the predicates (trigger and condition) of multiple outgoing transitions are pairwise disjoint. The approach for detecting non-determinism employs automated theorem proving (see Chapter 3), *i. e.*, proving the satisfiability of a formula consisting of the conjunction of each pair of transition predicates. More specifically, the tool tests whether there is a pair of transitions which may be enabled simultaneously. For this automated consistency analysis, the Stanford Validity Checker (SVC), an off-the-shelf theorem prover (as discussed in Section 3.3), is employed. The consistency check, thus, necessitates that the Statemate transition labels are transformed to SVC syntax. The tool is described to require only minimal effort to get familiar with, to scale well to large models, and its analysis results to be compositional, as results from the analysis of one subsystem are not sensitive to modifications of other subsystems.

Rule Checker: At the Technical University Braunschweig, Mutz and Huhn have developed a tool for the automated analysis of user-defined design rules for UML state machines, which can handle statechart models created by various UML modeling tools [79, 80]. They pursue an analysis based on the Object Constraint Language (OCL) [82], which they use to formulate rules and queries independent of the underlying data structure. This advantage is, however, bought at the cost of not being able to address graphical information. Hence, using OCL, it cannot be analyzed, *e. g.*, whether the initial state is located in the upper left corner of a statechart. Rules pertaining to graphical information must, therefore, be implemented directly in Java. Specifically, the tool is capable of checking rules from the following four areas: (1) consistency rules, *i. e.*, rules that verify the syntactic and semantic correctness of statecharts; (2) modeling rules, *i. e.*, rules that address the positioning of states and labels, the layout and direction of transitions, and other aspects to improve homogeneity and readability; (3) compatibility rules, *i. e.*, constraints that secure the exchange of models between different development environments; and (4) dialect-specific rules, as some dialects support intrinsic modeling constructs.

Due to the UML standard, the tool allows to import and export statecharts in the XML Metadata Interchange (XMI) format [83]. The user can configure the rule checker according to his/her needs by activating, deactivating, and parametrizing nearly 100 design rules. The tool then informs about problems and rule violations. New rules can be added either in Java or OCL, allowing for checks of widely accepted "best practices" as well as individual preferences on different statechart notations. Models can be analyzed with respect to their conformance to a standard and their compatibility to other CASE tools. Additional functions of Mutz and Huhn's tool include a tree-like representation of statecharts, advanced error and exception handling, automated and configurable report generation, a dynamic manual, and mechanisms for rule declaration and parametrization.

In summary, none of the presented tools entirely lives up to expectations. 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, but not profound semantic checks which require automated theorem proving as realized in the State Analyzer. However, semantic checks are particulary important since they eliminate possible non-trivial sources of error, which are very hard to discern for humans. Thus, the semantic checks, put forth in the next chapter of this thesis, aspire to fill the gap. The positioning of the *KIEL Checking* plug-in within Figure 4.4 further emphasizes this intention.

Chapter 5

A Proposal for Style Checking in Statecharts

Building on the aforementioned theoretical foundation, practical know-how, and available prototypes, this thesis set out to define a comprehensive statechart style guide, striving at applicability to all statechart dialects, present and future, within the limits of the UML state machines specification. Further, a corresponding automatic checking framework was implemented as part of the *KIEL* [55] statechart modeling tool, assuring flexibility and independence from any particular modeling tool.

As Buck and Rau correctly notice, it is essential that software developers recognize a style guide “as a means of support rather than a burden or evil management scheme to limit their freedom and suffocate their creativity” [19, page 22]. In order to promote acceptance, Buck and Rau give the following recommendations on formulating style guide rules:

Clarity: The meaning and motivation of all guidelines must be understandable for all developers.

Minimality: Each guideline should only cover one aspect to facilitate conformance testing and limit the rippling effect of guideline-changes.

Consistency: The whole set of guidelines must be consistent, *i. e.*, adhering to guideline *A* must not lead to breaking guideline *B*. Where mutually exclusive guidelines exist they should be clearly marked as such.

Consensus: Guidelines must not be imposed as laws from upper management. Instead, the developers should be involved in their definition and/or the tailoring process (see “Adaptability”).

Flexibility: Guidelines can only try to capture “best practices” that apply to the majority of cases. Thus, it must be possible to deviate from them in well-justified cases.

Adaptability: Every project has slightly different needs. Therefore, it must be possible to tailor project specific guidelines from a generic set. To facilitate this, the scope, priority and dependencies between guidelines must be documented.

Stability: Guidelines must not be changed frequently. Otherwise, developers will need to spend excessive time to keep up with the changes and update their work and eventually become frustrated.

Testability: Like requirements, guidelines are useless if conformance with them cannot be tested, either manually in a review or automatically using tools.

– Buck and Rau [19, page 22f.]

This advice was followed in formulating the robustness rules for statecharts described below. First, however, a well-structured taxonomy for style checking in statecharts is presented.

5.1 Taxonomy for Style Checking in Statecharts

In the general context of static code analysis, a clear and clean separation must be made between syntactic and semantic correctness on the one hand and style checking on the other hand. Figure 5.1 illustrates this contrast, which holds true not only for statecharts, but for all areas of computer programming. On this foundation, as a first step toward systematically devising an extensive style guide for statecharts, the following taxonomy, also depicted in Figure 5.2, was laid down:

1 **Syntactic Analysis:** The enforcement of syntax-related rules does, in general, not necessitate knowledge of model semantics. However, deriving the rule set may require limited knowledge of model semantics, as is the case for items 1.2 and 1.3.

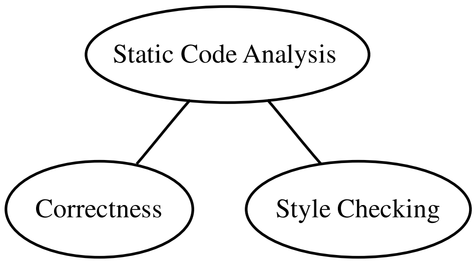


Figure 5.1: Distinction between correctness and style checking in the context of static code analysis.

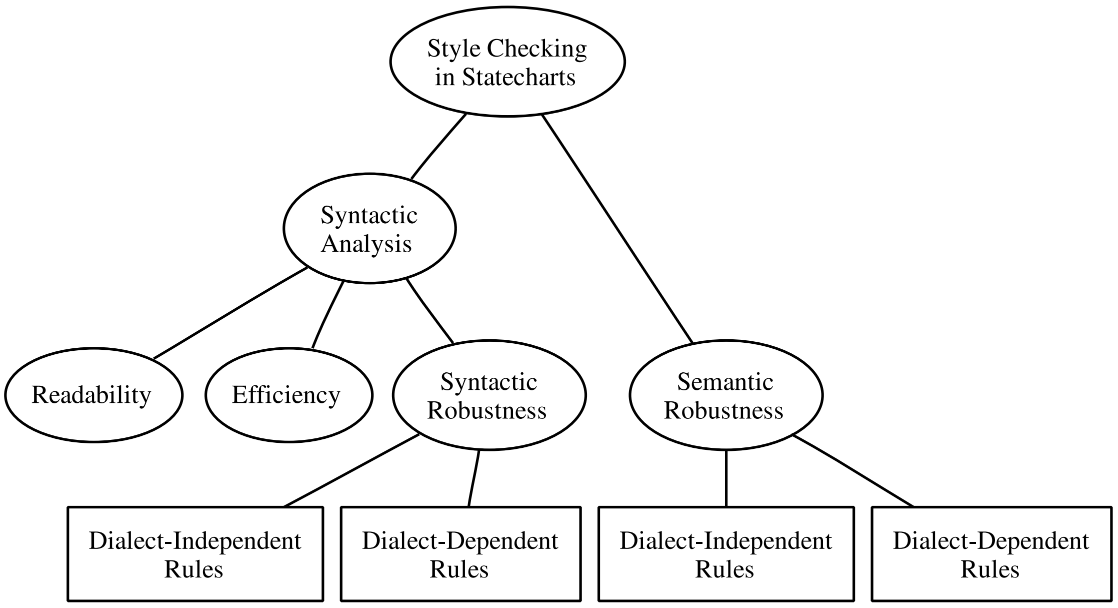


Figure 5.2: Taxonomy for style checking in statecharts.

1.1 Readability[1]**:** Readability (or layout style) is the domain of classic style guides. It aims at a graphical normal form, *e. g.*, transitions connect states in a clockwise direction, charts contain a limited number of states, transitions do not cross each other, or transition labels must be placed so that the relation between label and transition is clear and unambiguous [59].

1.2 Efficiency: Efficiency (or compactness, simplicity) eliminates superfluous and redundant elements from the statechart model. This can be achieved by applying optimizers to the model. A meta-language for formulating the rules may be beneficial.

1.3 Syntactic Robustness: Syntactic Robustness aims at reducing errors due to inadvertence and enhancing maintainability. Statechart dialect-dependent and dialect-independent rules are differentiated; rules may be formulated in OCL, as elaborated below.

2 Semantic Robustness: Deriving and enforcing semantic robustness rules requires knowledge of specific aspects of the model semantics. In this case, a descriptive language for formulating the rules, *e. g.*, OCL, is less practicable. Exact analysis typically requires the use of verification tools, *e. g.*, reachability, disjunction of transition predicates, shadowing of (less-prioritized) transitions. Structure-based heuristics may be beneficial as well, *e. g.*, in checking for race conditions. Statechart dialect-dependent and dialect-independent rules are differentiated.

As a whole, this taxonomy is suggested as a productive groundwork for systematically devising an extensive style guide for statecharts. From hereon, the discussion focuses exclusively on statechart robustness analysis, syntactic as well as semantic.

5.2 Robustness Rules for Statecharts

As mentioned before, robustness rules typically stem from collective engineering experience and common sense. Therefore, in the style of the *MISRA C* rules development process [75], all available “raw” information, stemming from documented best-practices and collective engineering experience, was filtered and adapted into rules; next, the resulting rules were subject to peer and expert review and revision. In this sense, the

[1]Within the *KIEL* [55] statechart modeling framework, briefly introduced in Chapter 1, a readability style guide and style check exist implicitly as part of the automatic layout functionality [57].

bibliographic references mentioned in the finally evolved set of rules presented below should be understood as mere sources of inspiration. Robustness rules are always built on the assumption of syntactic and semantic soundness. Hence, rules regarding the correctness of statecharts, *e. g.*, the presence of exactly one initial pseudo-state in each OR-state, are certainly not part of this set. Such rules are, however, often part of the tools discussed in the previous chapter, which partly explains their self-praised sizable number of rules. Moreover, robustness rules are divided into tool-independent and tool-dependent rules. Tool-dependent semantic robustness rules do not exist at the moment since the three statechart dialects considered in this thesis have rather closely related semantics.

I. Syntactic Robustness Rules

Tool-Independent Rules

1 **Departability:** All non-final states should have at least one departing transition [78].

2 **Non-Emptiness:** There should not be "empty" transitions, *i. e.*, transitions without trigger and condition, leading directly from one state to another state [78]. If a statechart dialect contains a default signal that is present at every instant, default transitions should be labeled with that signal. If no such default signal exists, one should be defined.

3 **Parsimony:** All regions of an AND-state should contain at least one sub-state in addition to the initial pseudo-state, and all OR-states should contain at least two sub-states in addition to the initial pseudo-state [78]. Otherwise, such constructs merely introduce unnecessary levels of hierarchy, spawning additional sources of error.

4 **Reachability:** There should not exist any isolated states, *i. e.*, states that have no incoming transitions [78]. Such states are unreachable and should therefore be removed.

5 **Uniqueness:** The labels of all states should be unique [78]. Comprehension may be impaired, if sub-states of two different composite states carry the same label.

6 **Variable Initialization:** System variables should be properly initialized [43, 58, 59, 78]. As the environment of a system will continue to change when the system is off-line, it is crucial that all variables are initialized at system start-up. To illustrate the relevance of this rule,

Leveson [61] describes a system controlling a chemical reactor that is taken off-line for an update during the charging of the reactor. When restarted, the system was not properly re-initialized and "assumed" the reactor to still be in its last known state. This eventually led to an over-charging of the reactor.

Tool-Dependent Rules

1 **Connective Junctions I:** In Stateflow, transition segments that lead to a connective junction, should not have any condition actions, or the connective junction must have a departing default transition [40]. Such condition actions are evaluated even if none of the following transition segments become active [68, pages 3-49f.].

2 **Connective Junctions II:** In Stateflow, Connective Junctions should not be used as they constitute an error-prone concept [40]. This rule, of course, cancels out the previous rule.

II. Semantic Robustness Rules

Tool-Independent Rules

1 **Dwelling:** The predicates of incoming and outgoing transitions of a state should be pairwise disjoint, or at least not completely overlapping [43, 58]. This rule ensures that the system pauses at every state it reaches. A state that is passed without stopping might be superfluous.

2 **Transition Overlap:** All transitions (directly or indirectly) outgoing from a state should have disjoint predicates [43, 58]. If the predicates overlap, a transition might be shadowed, and its target state might therefore be unreachable.

3 **Write/Read Race Conditions, "Dirty Read":** In parallel states, there should not be concurrent reading and writing of a variable [43, 58]. Unfortunately, Write/Read-Races are not detectable, in general. A very conservative approximation is implemented in Esterel Studio: "If a variable is written in a thread, then it can be neither read nor written in any concurrent thread" [15, page 52].

4 **Write/Write Race Conditions, "Lost Update":** In parallel states, there should not be concurrent writing of a variable [43, 58]. Here, the same as stated above for Write/Read Race Conditions holds.

As a next step toward an automatic checking framework, the rules were divided into two sets: those rules that can be formally defined as constraints in OMG's Object Constraint Language (OCL)—to be subsequently translated to Java code, utilizing the *Dresden OCL Toolkit* [31]—and those rules that cannot. The former set includes all syntactic rules as these can easily be expressed as OCL constraints. On the other hand, semantic robustness rules, in general, cannot be expressed entirely as OCL constraints because checking a statechart w. r. t. these rules typically requires extensive knowledge of the model semantics and even automated theorem proving, as explained below. In particular, the *Dwelling* check and the *Transition Overlap* check cannot be specified using OCL constraints alone; additional Java code is needed for formulating the required theorem-proving queries and sending them to an outside tool for analysis. The premise of using OCL in conjunction with Java, however, entails many complications, outweighing the gains of using OCL in the first place. Moreover, the intricate, global scoping of the *Race Conditions* check, makes it impossible to formulate this check in OCL, where a clearly defined scope is indispensable. Accordingly, these three checks are implemented directly in Java code.

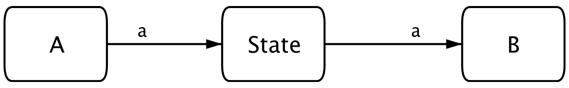
The author is well aware that a twofold rule-handling—Java code on the one hand and OCL constraints on the other hand—leads to drawbacks in maintainability and uniform extendability of the checking framework. However, accepting these deficits, facilitates a vast range of automatic checks, while at the same time achieving a highest-possible level of abstraction in implementing the checks. Whereas the implementation of the syntactic checks is subject of a separate, but closely related diploma thesis [13], the specifics of automated semantic robustness analysis of statecharts are hereafter illuminated in detail.

5.3 Semantic Robustness Rules for Statecharts

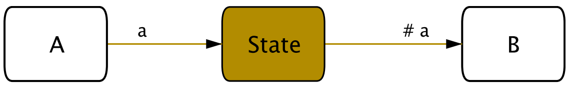
As introduced in Section 4.1, robustness rules for statecharts seek the definition of a less error-prone subset of this powerful modeling paradigm. This section provides a semi-formal specification and in-depth description of the semantic robustness rules put forward in the previous section. All examples depicting partial statecharts have been modeled and laid out using *KIEL*. For these particular rules, transitions are always considered pairwise. Thus, let $trans_1$ and $trans_2$ be the two transitions under investigation. The label of $trans_i$ is l_i, which consists of an event expressions e_i, a condition expressions c_i, and action expression a_i, where $i \in \{1,2\}$.

5.3.1 Dwelling

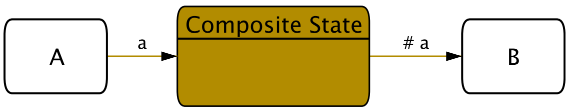
The *Dwelling* rule ensures a system's capability to pause at every state it reaches. A state in which the system cannot pause is not in accordance with the definition of a state as given in Section 2.1. Reaching a state due to the occurrence of an event and later leaving the state due to another occurrence of the same event certainly represents impeccable behavior (Figure 5.3a). However, reaching a state and instantly leaving it due to a single occurrence of an event constitutes a potential modeling error (Figures 5.3b and 5.3c).



(a) Non-transient state.



(b) Transient state.



(c) Transient composite state.

Figure 5.3: Semantic robustness rule *Dwelling*.

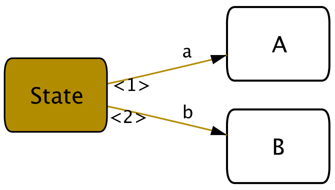
In order to determine if a *Dwelling* violation exists at a state, all incoming transitions and all "immediate" outgoing transitions (in Figure 5.3 denoted by Esterel Studio's #) are evaluated pairwise. A *Dwelling* violation exists if the event expression and condition expression of incoming transition $trans_1$ are disjoint from the respective expressions of outgoing transition $trans_2$. Technically, the satisfiability w. r. t. the theory of booleans in combination with linear integer and real arithmetic of the formula

$$\Big((e_1 \wedge c_1) \wedge (e_2 \wedge c_2)\Big) \tag{1}$$

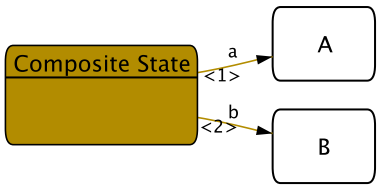
must be determined. Satisfiability of the formula implies that the predicates of $trans_1$ and $trans_2$ are not disjoint, *i. e.*, a *Dwelling* violation is on hand. Such a violation may be eliminated by simply adding the negation of the predicates of one transition to the predicates of the other transition, *e. g.*, $trans_2$ remains unchanged, but the predicates of $trans_1$ are augmented by $\neg e_2$ and $\neg c_2$ to yield $(e_1 \wedge \neg e_2)$ for the event expression and $(c_1 \wedge \neg c_2)$ for the condition expression. Other solutions are certainly conceivable. Ultimately, the modeler must decide if and how to resolve the situation.

5.3.2 Transition Overlap

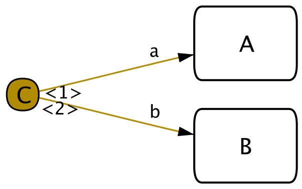
Ensuring that the predicates of two transitions departing from a state have disjoint predicates warrants that at most one transition is enabled at any time. This leads to guaranteed deterministic behavior independent of transition priorities, thus reducing the risk of errors otherwise easily introduced as part of an incremental modeling or remodeling process. The rule applies to simple and composite states as well as all pseudo-states with more than one departing transition, as shown in Figure 5.4. Although requiring considerable computational complexity, outgoing transitions must be considered pairwise, in order to identify two overlapping ones. Technically, checking this rule is similar to checking the previous rule. In this case, the satisfiability of Formula (1) indicates the overlapping of the predicates of transitions $trans_1$ and $trans_2$. As above, a possible solution may be obtained by adding the negation of the predicates of one transition to the predicates of the other transition. Again, the modeler must decide on the correct resolution on a case-by-case basis.



(a) Simple state with overlapping transitions.



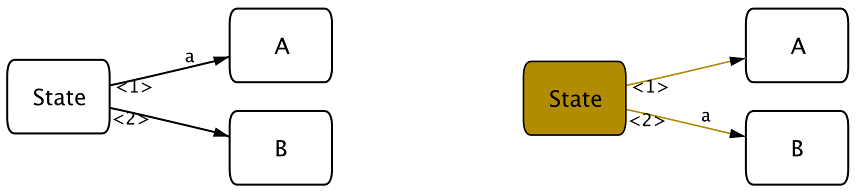
(b) Composite state with overlapping transitions.



(c) Conditional pseudo-state with overlapping transitions.

Figure 5.4: Semantic robustness rule *Transition Overlap*.

The strict enforcement of this rule, however, has the side-effect of eliminating the concept of a default transition that is always enabled, but only taken when no other transition is enabled. This occurs because a transition with empty predicates always overlaps with any other transition. Therefore, as shown in Figure 5.5, a single default transition without trigger and condition is allowed for every state and pseudo-state, but only if this transition has the lowest priority of all transitions.



(a) Unlabeled transitions with lower priority are allowed.

(b) Unlabeled transitions with higher priority should be avoided.

Figure 5.5: Semantic robustness rule *Transition Overlap*: applicability to default transitions.

In addition to transitions departing directly from a state, transitions departing from an enclosing state may also be enabled. Depending on the statechart dialect, such indirectly departing transitions have higher or lower priority than directly departing transitions. In any case, the situation depicted in Figure 5.6 must be regarded as an infringement of the *Transition Overlap* rule. However, this strict interpretation of the rule might disturb the "abort" mechanism represented by indirectly departing transitions with higher priority. Hence, whether such transitions are considered in the automatic checking of this rule is configurable.

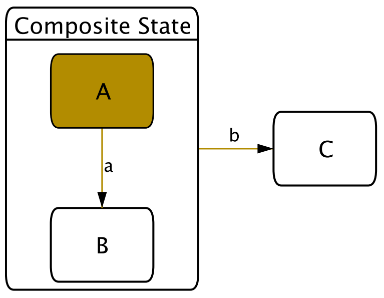


Figure 5.6: Semantic robustness rule *Transition Overlap*: applicability to "indirectly" outgoing transitions.

Transitions ending in a conditional connector, as shown in Figure 5.7, present another special case. Such transitions must be pursued in all possible directions until a regular state is reached. Accordingly, the satisfiability of the formula

$$\Big(\big((e_1 \wedge c_1) \wedge (e_c \wedge c_c)\big) \wedge (e_2 \wedge c_2)\Big) \tag{2}$$

must be determined, where e_c and c_c represent the event and condition expressions, respectively, of the transition segment departing from the conditional connector.

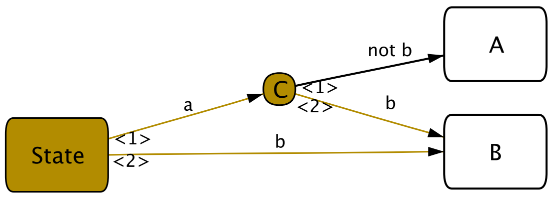
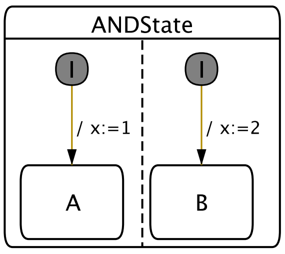


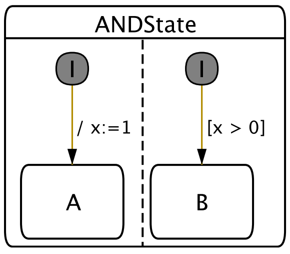
Figure 5.7: Semantic robustness rule *Transition Overlap*: applicability to transitions ending in a conditional.

5.3.3 Race Conditions

As in any other programming language, race conditions also pose a potential source of errors in statecharts. Two different forms of race conditions exist: write/read races, so called "dirty reads", and write/write races, so called "lost updates." Since, in general, neither of these are detectable, the following conservative approximation, as implemented in Esterel Studio, is adopted here: "If a variable is written in a thread, then it can be neither read nor written in any concurrent thread" [15, page 52]. Such behavior is shown in Figure 5.8.



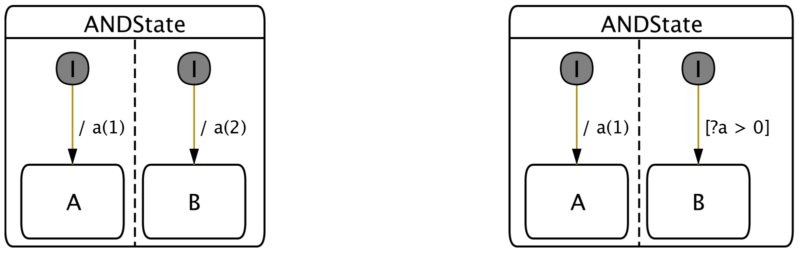
(a) Write/Write race condition.



(b) Write/Read race condition.

Figure 5.8: Semantic robustness rule *Race Conditions*: applicability to variables.

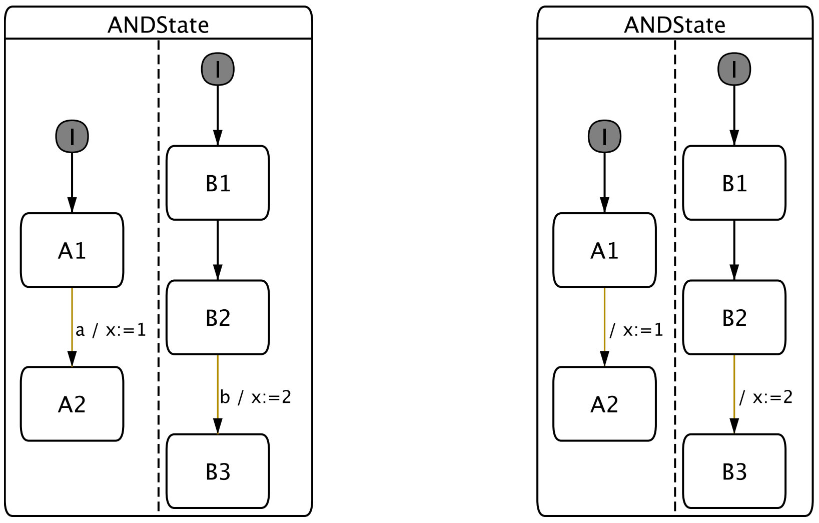
For the valued signals of Safe State Machines, circumstances are similar, as depicted in Figure 5.9. However, if a valued signal is declared as a combine signal, the situation in Figure 5.9a does not constitute a "lost update." The value of a combine signal is much rather updated according to a declared associative combine operation (addition, multiplication, or any associative user-defined operation). For the example in Figure 5.9a and addition as the combine operation, this means that signal *a* is emitted with the value 3, not 1 or 2.



(a) Write/Write race condition. **(b)** Write/Read race condition.

Figure 5.9: Semantic robustness rule *Race Conditions*: applicability to valued signals of Safe State Machines.

In the examples shown in Figures 5.8 and 5.9, the existence of a race condition is always clear and unambiguous. Due to the aforementioned approach, however, also cases where it is unclear if a race condition is on hand, as in Figure 5.10a, and even certain race-free cases, as in Figure 5.10b, are flagged. This is a result of the conservative structure-based heuristic of this robustness rule.



(a) Ambiguous case of a write/write race condition. **(b)** Race-free case that is marked to have a write/write race-condition.

Figure 5.10: Semantic robustness rule *Race Conditions*: applicability to ambiguous cases of parallelism.

Finally, the check is not only applicable to transitions that are directly parallel, *i. e.*, within neighboring regions of the same AND-state. Ascending and descending to all levels of hierarchy, the check also applies to cases such as the one depicted in Figure 5.11.

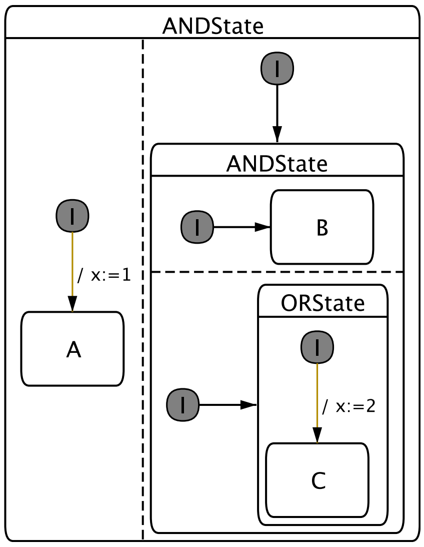


Figure 5.11: Semantic robustness rule *Race Conditions*: applicability to parallel transitions at all levels of hierarchy.

Chapter 6

The Implementation

The objective of this thesis was the development and implementation of an automated semantic robustness analysis of statecharts. After establishing the necessary theoretical ground, Chapter 5 provided a thorough definition of statechart robustness rules. Subsequently, this chapter presents the implementation of a tool for performing an automated semantic robustness check on statecharts. The tool is designed to adhere to the following constraints: modularity, extendability, and customizability of the rule set; and clear distinction of semantic robustness rules from correctness rules and syntactic robustness rules. Before turning to this, however, a brief overview of the *KIEL* framework is given.

6.1 The *KIEL* Framework

As already outlined in Chapter 1, *KIEL* is a software for the modeling and simulation of statecharts. The tool's main goal is to enhance the intuitive comprehension of the behavior of the system under development by providing an innovative dynamic visualization of the statechart simulation [89].

KIEL is implemented in Java (Standard Edition, Version 1.4.2 [100]); it can be compiled and executed both on Linux and on Windows XP. As shown in Figure 6.1 and briefly characterized below, *KIEL* consists of several independent components with well defined interfaces [56], which have largely been developed within the scope of student projects and diploma theses.

`DataStructure`: Represents a statechart's topology.

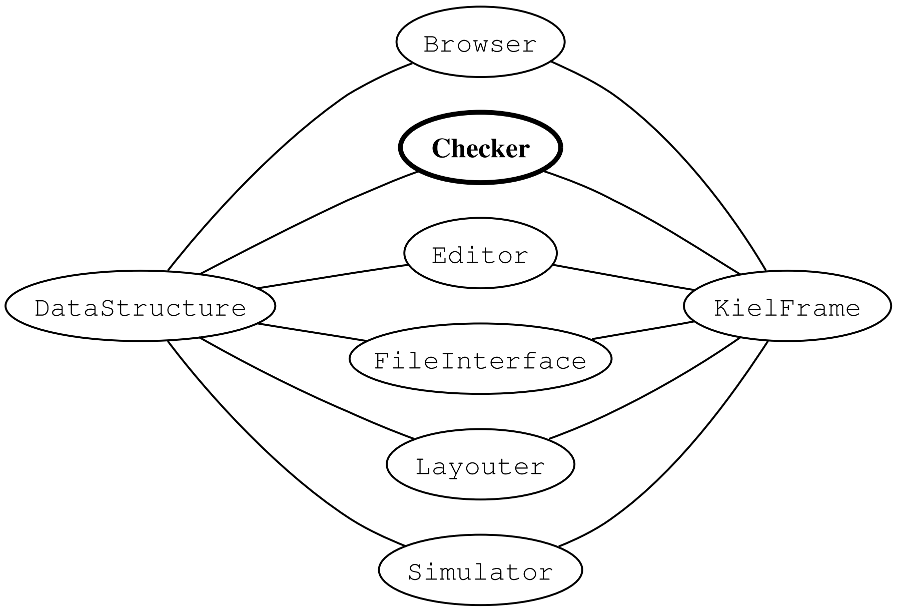


Figure 6.1: The components of *KIEL* and their conceptual interaction.

`Browser:` Controls the simulator and displays the statechart simulation [108].

`Checker:` Checks statecharts w. r. t. a modeling style guide—the focus of this thesis.

`Editor:` Enables the user to create new charts and modify existing ones [64].

`FileInterface:` Enables import and export of statecharts and simulation input data of several modeling tools [107].

`Layouter:` Provides different layout algorithms for the automatic layout of statecharts [57].

`Simulator:` Performs the simulation of models of various statechart dialects [84], currently Esterel Studio and Stateflow.

`KielFrame:` Provides *KIEL*'s main window and separates all components of the program from each other to warrant modularity. `KielFrame` encapsulates the `Checker`, the `FileInterface`, the `Layouter`, and the `Simulator`. No module calls methods of another module directly; `KielFrame` methods are called instead.

The tree data structure that represents a statechart's topology and the automatic layout mechanism that generates graphical views of the statechart form the foundation of *KIEL*.

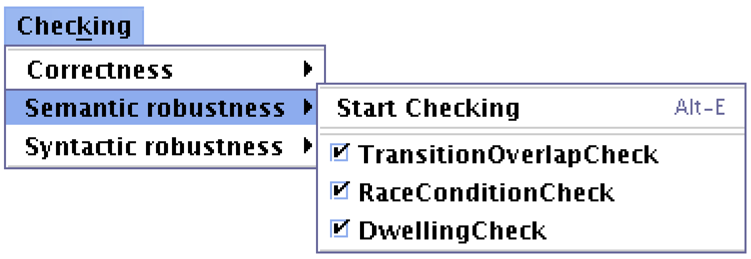


Figure 6.2: Screenshot of *KIEL*'s *Checking* menu.

The strict separation of topological and graphical information is the key to *KIEL*'s automatic layout capability and its support of multiple different layout algorithms. This powerful feature, in turn, facilitates a dynamic focus and context during statechart simulation as well as the possibility to edit statecharts based on their topology. The automatic layout further eliminates the need for an explicit layout style guide for statecharts. The *Checking* plug-in developed within this thesis operates directly on the data structure and sends its results to `KielFrame` for display. In addition to a graphical user interface (GUI), a command line version of *KIEL* exists, capable of rendering and exporting, optimizing, and checking statecharts.

6.2 The *KIEL Checking* Plug-In

The semantic robustness checks are realized as part of the *Checking* plug-in for the *KIEL* framework. The *Checking* plug-in provides the underlying infrastructure and utility methods needed for checking a statechart. Special care was taken to lay the ground for an arbitrary number of further checking packages, such as syntactic robustness checks or correctness checks. The individual checks are implemented as separate Java classes (Appendix A.3), extending a common base class (Appendix A.2.1). Checks are compiled into Java Archives (JARs), one JAR for each package of checks. In turn, the checks are dynamically loaded from the JARs when *KIEL* is started (Appendix A.2.2). Following the *Visitor* design pattern, all checks of a package are carried out on demand in a single depth-first traversal (Appendix A.2.3) of the abovementioned *KIEL* tree data structure. As show in Figure 6.2, checks may be individually enabled and disabled. Moreover, for each statechart dialect, a user-configurable default checking profile exists, specifying which checks are initially enabled and which are disabled.

As outlined in Section 5.3, an automated theorem prover is required for performing the semantic robustness checks. The open-source tool CVC Lite has been selected for this

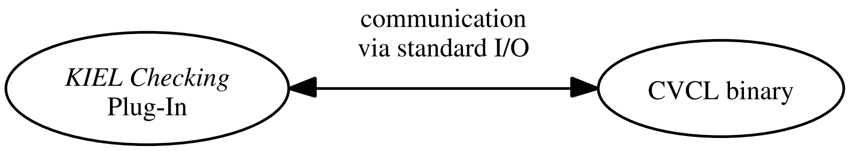


Figure 6.3: Outline of *KIEL Checking* plug-in communicating directly via standard I/O with CVC Lite executable.

purpose (cf. Section 3.3). After compilation, CVC Lite is available as a stand-alone executable and also as a shared-object C++ library. Using the executable is straightforward, as shown in Figure 6.3, but requires error-prone and time-consuming argument passing via standard I/O. In addition, CVC Lite's output must be parsed for interpretation. Alternatively, the CVC Lite library can be integrated into *KIEL*'s Java framework using the standardized Java Native Interface (JNI) [101]. As it could not be known ahead of time whether calling the CVC Lite library via JNI or communicating directly with the executable via standard I/O is more efficient, both possibilities have been implemented. Concluding performance issues are discussed below in Section 6.3.

The Java Native Interface enables Java programs to call native methods written in C or C++. Reasons for incorporating native methods generally include timing constraints, reuse of existing native code, and platform-specific tasks. Three conceptually different approaches to utilizing native code via JNI exist:

Native code modification: Modifying the native code to be in accord with the JNI naming scheme and augmenting two JNI pointer arguments to each method declaration; declaring native methods on the Java side.

Wrapper implementation: Manually implementing native wrapper methods to avoid modifying the existing native code; implementing Java wrapper methods to avoid declaring native methods on the Java side. By doing so, native methods may be used from within Java just as if they were Java methods.

Wrapper generation: Auto-generating native wrapper methods as well as Java wrapper methods.

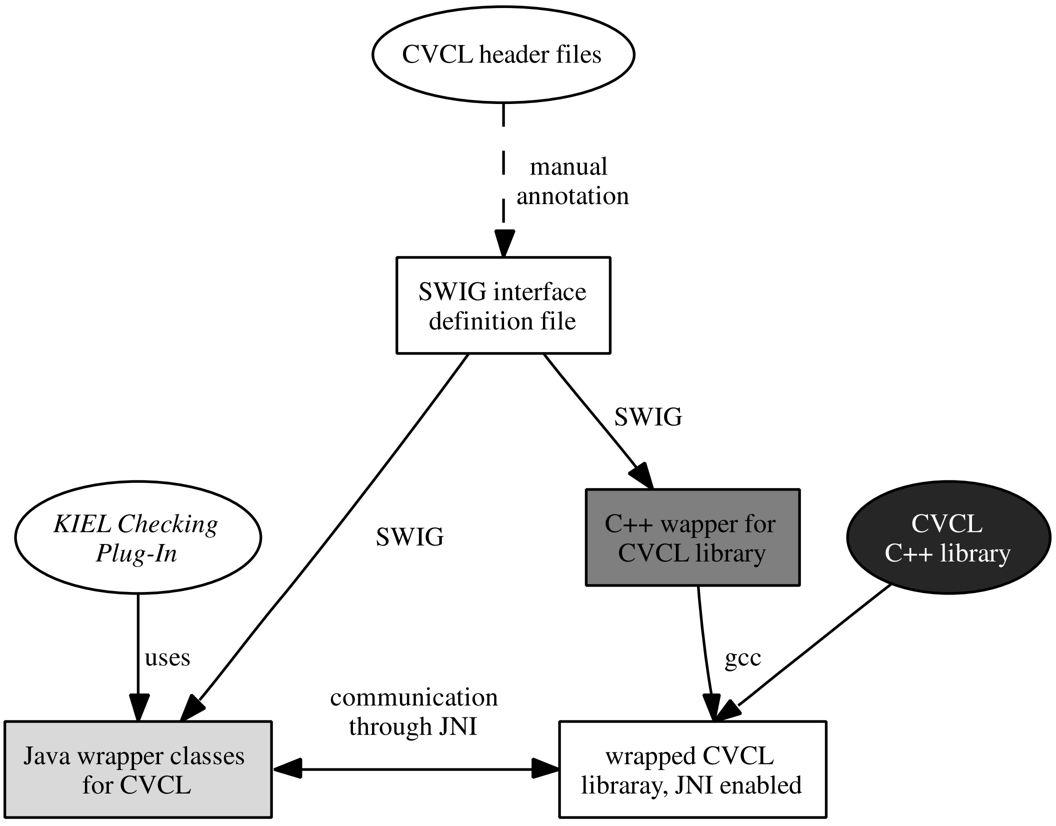
In case of CVC Lite, modifying the native source code is not feasible, simply due to the great size and complexity of the tool. Moreover, implementing native and Java wrappers manually is quite a tedious and error-prone task, especially on a large scale, as required for CVC Lite. Therefore, the open-source *Simplified Wrapper and Interface Generator*

(*SWIG*) [103] was employed. SWIG generates wrappers and interface files enabling the use of native methods from within many programming languages, including Java. The Java and C++ JNI wrappers, in this case, are produced from CVC Lite's annotated C++ header files, as shown in Appendix A.1. The complete SWIG workflow as well as the conceptual hierarchy of wrapper layers around the CVC Lite library are shown in Figure 6.4.

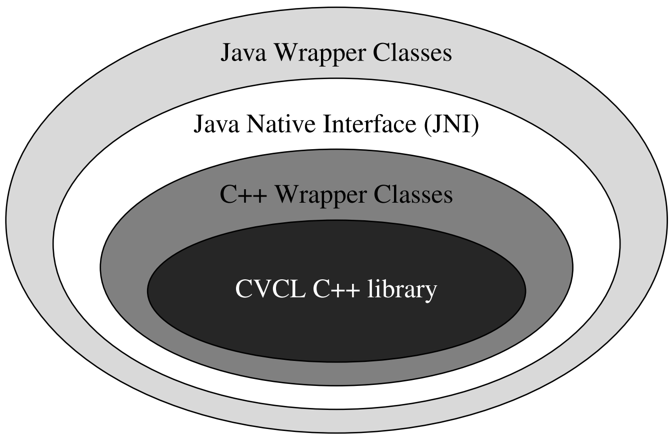
To perform the required satisfiability checking addressed in Section 5.3, the statechart transition predicates must be decomposed into their components and those components transformed from *KIEL* syntax to equivalent CVC Lite syntax. This is done by recursively parsing the transition predicates in order to form appropriate satisfiability queries. Depending on whether the CVC Lite library or binary is in use, a nested tool-specific expression is built for the library, while a lengthy string is concatenated for the binary (Appendix A.4). However, as mentioned in Section 3.3, CVC Lite does not determine the satisfiability of a formula, but rather its validity. Thus, the negation of Formula (1), derived in Section 5.3, is passed to CVC Lite. *Iff* this formula is found to be not valid, then the original Formula (1) is satisfiable.

The results of the semantic robustness checks are either displayed as a table on the *Semantic Checker* tab of *KIEL*'s GUI or as a textual description on the command line. When working with the GUI, individual or multiple messages may be selected with the mouse or keyboard (multiple interval selection [100]). For each selected message, the pertaining states and transitions are highlighted in the statechart above, as shown in Figure 6.5.

During the entire implementation process, only minor problems were encountered. One unsolvable technical predicament, however, regrettably entails the impossibility of utilizing the CVC Lite library when working on Windows XP. Here, CVC Lite cannot be compiled directly as it depends on several unavailable POSIX functions. With the aid of Cygwin's POSIX subsystem [26], however, compilation is straightforward. Unfortunately, native libraries depending on Cygwin cause the Java Virtual Machine (JVM) to crash when it tries to load such a library. This behavior is documented [49] and arises because the bottom-four kilobytes of the JVM's stack are overwritten by Cygwin. The incompatibility exists since version 1.5.5 of Cygwin's POSIX subsystem and occurs with all JVM versions. As a result, the CVC Lite library cannot be employed on the Windows platform, and thus, only the CVC Lite binary is available.

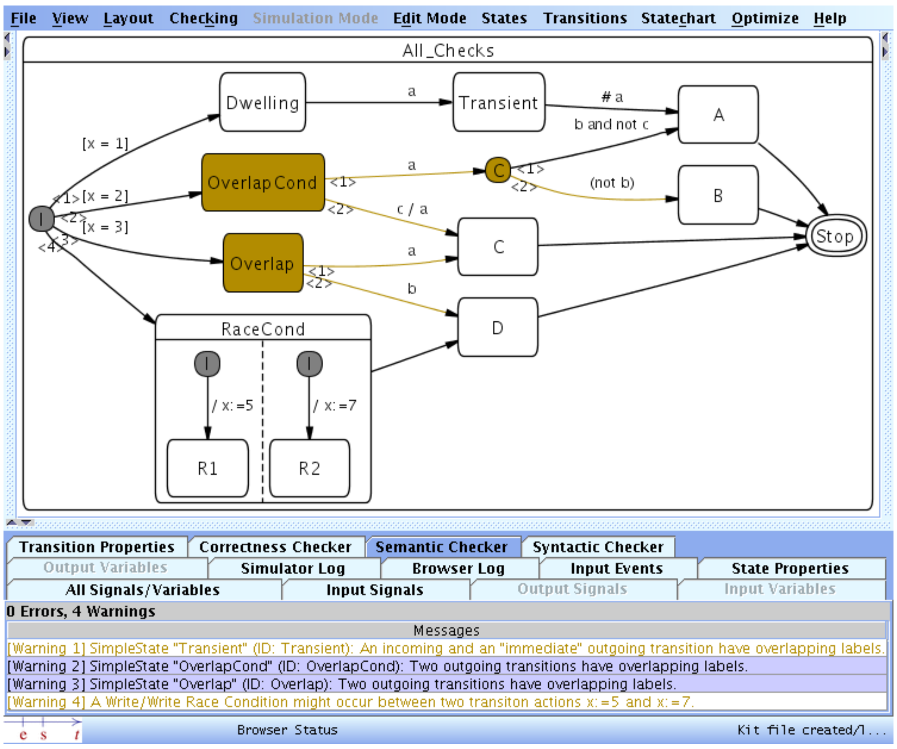

CVCL header files
manual annotation
SWIG interface definition file
SWIG
SWIG
KIEL Checking Plug-In
C++ wapper for CVCL library
CVCL C++ library
uses
gcc
communication through JNI
Java wrapper classes for CVCL
wrapped CVCL libraray, JNI enabled

(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 6.4: Interfacing of *KIEL* and the CVC Lite library via JNI and SWIG.



```
0 Errors, 4 Warnings
Messages
[Warning 1] SimpleState "Transient" (ID: Transient): An incoming and an "immediate" outgoing transition have overlapping labels.
[Warning 2] SimpleState "OverlapCond" (ID: OverlapCond): Two outgoing transitions have overlapping labels.
[Warning 3] SimpleState "Overlap" (ID: Overlap): Two outgoing transitions have overlapping labels.
[Warning 4] A Write/Write Race Condition might occur between two transiton actions x:=5 and x:=7.
```

Figure 6.5: Results of the semantic robustness analysis displayed in *KIEL*'s graphical user interface.

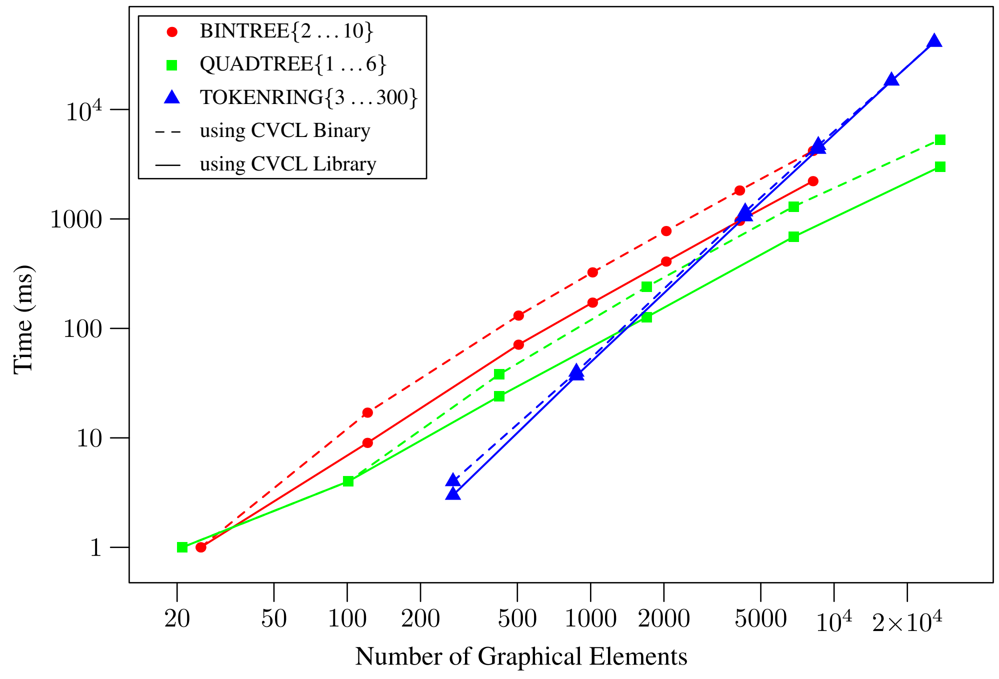
6.3 Performance Analysis

To quantitatively assess the efficiency of the automatic conformance checking of the semantic robustness rules, the checks were applied to a number of pertinent examples. The examples were largely taken from the Estbench Esterel benchmark suite [21] and the Columbia Esterel Compiler distribution [32]. In order to apply the checks, the Esterel programs were transformed to equivalent Safe State Machines using *KIEL*'s Esterel transformation [90]. Without any further optimization, this transformation generally yields statecharts with a large number of graphical elements (states, pseudo-states, transitions, etc.), well-fit for a performance analysis of the checking framework. In addition to these "artificial" examples, which were mainly chosen to assess the scaling of the checking run-time, manually created models (ELEVATOR DOOR[1], ABRO, and STOPWATCH) were evaluated, as well. Table 6.1 and Figure 6.6 present the experimental results; specifically, the number of graphical elements in the model, the run-time of the semantic robustness analysis, and the number of generated warnings are shown.

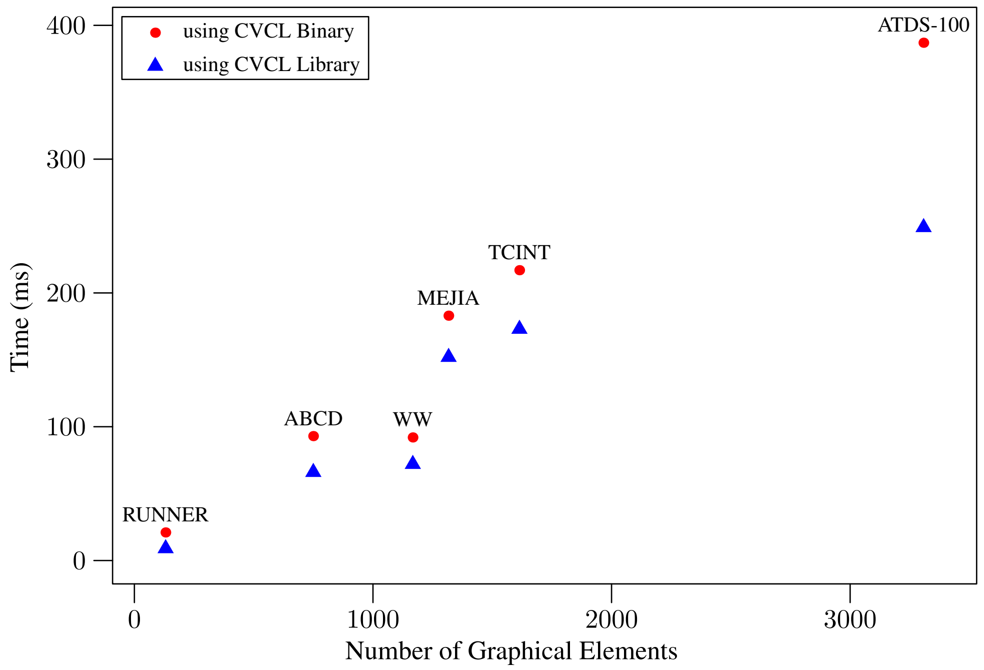
[1]The model of an elevator door is taken from a larger example which is part of the Esterel Studio distribution.

Table 6.1: Experimental results of the semantic robustness analysis of statecharts, listing the number of graphical elements in the models, run-times of the analysis, and numbers of generated warnings.

Model	Graphical Elements	Time (ms)		Generated Warnings
		CVCL Bin	CVCL Lib	
ELEVATOR DOOR	27	1	1	2
ABRO	37	1	<1	2
STOPWATCH	57	4	3	8
SCHIZOPHRENIA	42	1	1	0
REINCARNATION	86	4	3	8
JACKY1	98	6	3	11
RUNNER	131	21	9	55
ABCD	750	93	66	136
GREYCOUNTER	768	48	48	28
WW	1,167	92	72	137
MEJIA	1,317	183	152	142
TCINT	1,614	217	173	235
ATDS-100	3,308	387	249	1,262
MCA200	19,156	60,371	57,662	51,196
BINTREE-2	25	1	1	4
BINTREE-4	121	17	9	68
BINTREE-6	505	131	71	516
BINTREE-8	2,041	775	408	3,076
BINTREE-10	8,185	4,181	2,214	16,388
QUADTREE-1	21	<1	<1	0
QUADTREE-2	101	4	4	16
QUADTREE-3	421	38	24	144
QUADTREE-4	1,701	240	127	912
QUADTREE-5	6,821	1,299	687	5,008
QUADTREE-6	27,285	5,303	2,998	20,032
TOKENRING-3	272	4	3	0
TOKENRING-10	874	40	37	0
TOKENRING-50	4,314	1,165	1,051	0
TOKENRING-100	8,614	4,708	4,345	0
TOKENRING-300	25,814	41,376	41,265	0



(a) Checking run-times of the BINTREE, QUADTREE, and TOKENRING series.



(b) Checking run-times of the Estbench examples [21].

Figure 6.6: Plots of checking run-time vs. model size, illustrating the experimental results of the automated semantic robustness analysis of statecharts.

Irrespective of whether using the CVC Lite binary or library, the time required to perform the checking varies from a few milliseconds needed for small statechart models up to 60 seconds needed for very large models containing elaborate parallelism and generating a great number of warnings. For the majority of the examples, checking is much faster, though. Run-times scale approximately linearly w. r. t. the number of graphical elements of the checked models, as illustrated in Figure 6.6a. However, the TOKENRING series of statecharts exhibits a quadratically scaling run-time characteristic. The massive parallelism (all within a single AND-state) causes a quadratically growing number of *Race Condition* checks. Also, for the MCA200 model, the enormous number of more than 51,000 generated warnings (cf. Table 6.1) contributes heavily to the 60-second run-time. Overall, the collected data suggests the complexity class $O(n^2)$, where n is the number of graphical elements, as an upper bound to the checking run-time.

When utilizing the CVC Lite library, run-times are consistently shorter as compared to the CVC Lite binary. This performance enhancement, also portrayed in Figure 6.6, certainly justifies the additional labor required to incorporate the CVC Lite library into the *KIEL* framework, as described in Section 6.2. Moreover, the run-time interval between the library and binary expands as the number of queries sent to CVC Lite grows for larger models. The increasing amount of communication via standard I/O affects the run-time of the binary more severely than the equally growing number of JNI calls affects the library run-time.

In Figure 6.6b, it can be further observed that the checking run-times for the models ABCD and WW are very similar, although the model WW is approximately 50% larger. This is caused by WW's low number of states with more than one departing transition, resulting in a low number of *Transition Overlap* checks. Moreover, WW contains an unusually high number of transition labels which are not interpreted by *KIEL*, *e. g.*, containing calls to C-functions.

Despite the synthetic nature of the majority of the examples, the mere quantity of generated warnings suggests, but certainly does not prove, the necessity and success of a robustness analysis for statecharts. Beyond that, even small and concise handmade models of a stopwatch or an elevator door yield several robustness warnings (cf. Table 6.1). The conclusion of this thesis (Chapter 7) contains a more thorough qualitative appraisal of the efficacy of the suggested robustness rules.

Chapter 7

Discussion and Conclusion

Statecharts have made reactive system development easier, certainly more intuitive, and hence, less error-prone. Although, they are no magic bullet. To take error prevention even further, a statechart style guide has been developed as the foundation of this thesis. 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 [55]. Great care was taken in devising the rule set and in designing the checking framework as not to constrict the modelers' creativity, but to cater for more explicit, easy to comprehend, and less error-prone models. This was achieved by adhering to the following fundamental design principles:

Modularity and Configurability: In the spirit of Buck and Rau's notion of *flexibility* and *adaptability* [19] (cf. Chapter 5), all robustness checks are independently implemented, individually selectable, and parametrizable via a preferences management. A configurable default checking profile for models of each statechart dialect further enhances useability.

Extendability of the rule set: The set of checks is easily extendable by either implementing a new Java class or, if feasible, by adding an appropriate OCL constraint.

Automatic conformance checking: As shown in Section 6.3, compliance with the semantic robustness rules can be checked very rapidly—a key quality, 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.

In working with the presented semantic robustness rules, a duality between robustness and minimality of statecharts has become evident. For example, 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 in Section 5.3, constitutes an infringement of the *write things once* principle of modeling [14]. Thus, fully explicit behavior specification postulated by robustness rules stands in contrast to the clever exploitation of implicity. Correspondingly, statechart models optimized w. r. t. compactness and minimality frequently yield more robustness warnings than their non-optimized counterparts. In the development of new robustness rules and the enforcement of existing rules this duality must be kept in mind to ensure that modeling does not loose its inherent intuitiveness. Ultimately, it must always be left up to those who utilize the proposed rules to decide what actually represents an undesirable modeling construct and what not.

Backed by the experimental results presented in Section 6.3, the robustness rules, in combination with an automatic conformance checking, appear to fulfill their expected purpose. Large numbers of all three kinds of robustness violations, denoting potential modeling errors, are detected in models ranging from small to large and from trivial to highly complex. Even long before exposing true modeling errors, the robustness checks can expedite and ease the understanding of an unfamiliar model. For example, the *Dwelling* violations found in the model of a wristwatch (cf. model WW in Table 6.1) reveal a multitude of transient states resulting from the transformation of Esterel code to Safe State Machines. Eliminating these transient states leads to a more stringent and, hence, more intuitive model. The *Transition Overlap* violations found in the same model expose far-reaching relations spread-out over many levels of hierarchy, which might otherwise remain undiscovered. Similarly, the potential race conditions found in the model MCA200 help to identify concurrent access of variables even in far apart regions of the model. Altogether—although no immediate errors have been found in any model—the robustness rules and their automatic conformance checking satisfy the expectations. Intricate interrelations between various components of a model, which are nearly impossible to discover for the human beholder, are elegantly brought to attention.

The true practicality and efficacy in terms of error prevention of the proposed set of robustness rules can, however, only be proven in an extensive study. For this, two groups of test subjects would need to be asked to model a realistic system. Of these two groups, only one would have knowledge of the robustness rules and access to the automatic conformance checking. Significantly fewer modeling errors would then be expected from the “robustness” group. Assuming that everything else was equivalent between the two groups, this would conclusively indicate the success of the suggested rules. A likewise elaborate alternative would be to apply the robustness checks to a (large) number of industry-sized models of real systems and consequently detecting errors in these models. This would obviously require in-depth knowledge of the modeled systems. Both propositions are unfortunately far beyond the scope of this thesis. Nevertheless, *KIEL*'s transformation of Esterel programs to Safe State Machines could be modified so that it generates statechart models containing only a minimal number of robustness violations—or better even, none at all. Moreover, in light of the small number of robustness rules established so far, the development of a much more extensive catalog of statechart dialect-independent and dialect-dependent robustness rules remains as future work.

In closing, statechart robustness rules step up, where the human modeler lets off—either due to inadvertence and carelessness or because size and complexity become nearly unmanageable for models of many real-life systems. Robustness rules and their automatic checking, in conjunction with the right modeling attitude, have the potential to make software and system development safer and less error-prone. Hence, we come a small step closer to the age-old dream of error-free software.

Bibliography

[1] Charles André. SyncCharts: A Visual Representation of Reactive Behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France, Rev. April 1996. URL `http://www.i3s.unice.fr/~andre/CAPublis/SYNCCHARTS/SyncCharts.pdf`.

[2] Mark A. Ardis, John A. Chaves, Lalita Jategaonkar Jagadeesan, Peter Mataga, Carlos Puchol, Mark G. Staskauskas, and James Von Olnhausen. A Framework for Evaluating Specification Methods for Reactive Systems: Experience Report. *IEEE Transactions on Software Engineering*, 22(6):378–389, 1996.

[3] Alessandro Armando and Enrico Giunchiglia. Embedding Complex Decision Procedures Inside an Interactive Theorem Prover. *Annals of Mathematics and Artificial Intelligence*, 8(3-4):475–502, 1993.

[4] Alessandro Armando, Claudio Castellini, Enrico Giunchiglia, Marco Maratea, and Massimo Idini. TSAT++: An Open Platform for Satisfiability Modulo Theories. In *Proceedings of the 2nd Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR'04)*, 2004.

[5] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded Model Checking of C Programs Using SMT Solvers Instead of SAT Solvers. Technical report, Artificial Intelligence Laboratory, Department of Communication, Computer and System Sciences, University of Genova, 2005.

[6] Cyrille Artho. Jlint – Find Bugs in Java Programs, 2006. URL `http://jlint.sourceforge.net/`.

[7] BarcelogicTools for SMT, Technical University of Catalonia in Barcelona, 2005. URL `http://www.lsi.upc.edu/~oliveras/bclt-main.html`.

[8] 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.

[9] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. Validity Checking for Combinations of Theories with Equality. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design, Palo Alto, CA*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 1996.

[10] Clark W. Barrett, Leonardo de Moura, and Aaron Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In K. Etessami and S. Rajamani, editors, *17th International Conference on Computer Aided Verification*, pages 20–23. Springer, 2005.

[11] Clark W. Barrett, Leonardo de Moura, and Aaron Stump. Design and Results of the 1st Satisfiability Modulo Theories Competition (SMT-COMP 2005). *Journal of Automated Reasoning*, to be published.

[12] 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.

[13] Ken Bell. Überprüfung der Syntaktischen Robustheit von Statecharts auf der Basis von OCL. Diplomarbeit, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, 2006.

[14] 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.

[15] 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`.

[16] Gérard Berry and the Esterel Team. *The Esterel v5_91 System Manual*. INRIA, June 2000. URL `http://www-sop.inria.fr/esterel.org/`.

[17] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi A. Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. The MathSAT 3 System. In *Proceedings of the Conference on Automated Deduction (CADE-20), Tallin, Estonia*, 2005.

[18] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted

Functions. In *Proceedings Computer-Aided Verification, Copenhagen, Denmark*, volume 2404 of *Lecture Notes in Computer Science*. Springer, 2002.

[19] Daniel Buck and Andreas Rau. On Modelling Guidelines: Flowchart Patterns for STATEFLOW. *Softwaretechnik-Trends*, 21(2):7–12, August 2001.

[20] Oliver Burn. Checkstyle, 2005. URL `http://checkstyle.sourceforge.net/`.

[21] CEC. Estbench Esterel Benchmark Suite. `http://www1.cs.columbia.edu/~sedwards/software/estbench-1.0.tar.gz`.

[22] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the Really Hard Problems Are. In John Myopoulos and Ray Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 331–340, 1991.

[23] Alonzo Church. A Note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1: 40–41 and 101–102, 1936.

[24] Stephen A. Cook. The Complexity of Theorem Proving-Procedures. In *Proceedings Third Annual ACM Symposium on Thoery of Computing*, pages 151–158, 1971.

[25] CVC Lite Homepage, 2004. URL `http://chicory.stanford.edu/CVCL/`.

[26] Cygwin Information and Installation, 2006. URL `http://www.cygwin.com/`.

[27] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

[28] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[29] Jerry Doland and Jon Valett. C STYLE GUIDE. Technical Report, Software Engineering Laboratory Series SEL-94-003, National Aeronautics and Space Administration (NASA), 1994. URL `http://sel.gsfc.nasa.gov/website/documents/online-doc/94-003.pdf`.

[30] Bruce Powel Douglass. Modeling Behavior with UML Interactions and Statecharts. Tutorial at OMG Real-Time and Embedded Distributed Object Computing Workshop, July 2002.

[31] Dresden OCL Toolkit, 2006. URL `http://dresden-ocl.sourceforge.net/`.

[32] Stephen A. Edwards. CEC: The Columbia Esterel Compiler. `http://www1.cs.columbia.edu/~sedwards/cec/`.

[33] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Proceedings of the 2003 International Conference on Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

[34] *Merriam-Webster's. Collegiate Dictionary.* Merriam-Webster, Springfield, MA, 11th edition, 2003.

[35] *Esterel Studio User Guide and Reference Manual.* Esterel Technologies, 5.0 edition, May 2003.

[36] David Evans and David Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 19(1):42–51, 2002.

[37] Free Software Foundation. GCC – The GNU Compiler Collection. `http://gcc.gnu.org/`.

[38] 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`.

[39] Kurt Gödel. *Über die Vollständigkeit des Logikkalküls.* PhD thesis, University Of Vienna, 1929.

[40] Reinhard von Hanxleden. Stateflow and STATEMATE – A Comparison of Statechart Dialects. Projektbericht FT3/AS-1999-003, DaimlerChrysler, August 1999.

[41] Reinhard von Hanxleden, Ritwik Bhattacharya, and Kay Kossowan. The State Analyzer, Version 0.1. Projektbericht FT3/AS-1999-002, DaimlerChrysler, February 1999.

[42] 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`.

[43] Reinhard von Hanxleden, Kay Kossowan, and Jörg Donandt. Robustheitskriterien für Statecharts. Projektbericht FT3/AS-2000-003, DaimlerChrysler, March 2000.

[44] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[45] David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[46] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach.* McGraw-Hill, New York, 1998.

[47] David Hilbert. Mathematische Probleme. *Nachrichten der Königlichen Gesellschaft der Wissenschaften zu Göttingen*, Heft 3:253–297, 1900.

[48] Tony Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. *Journal of the ACM*, 50(1):63–69, 2003.

[49] Elliott Hughes. Porting JNI Code to Win32 with Cygwin, 2005. URL `http://elliotth.blogspot.com/2005/08/porting-jni-code-to-win32-with-cygwin.html`.

[50] Intel White Paper. Statistical Analysis of Floating Point Flaw, November 1994. URL `http://support.intel.com/support/processors/pentium/fdiv/wp/`.

[51] Andrew Ireland. Tool Integration for Reasoned Programming. In *Proceedings of the Verified Software: Theories, Tools, Experiments (VSTTE) Conference*, 2005. URL `http://vstte.ethz.ch/`.

[52] 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.

[53] Richard M. Karp. Reducibility Among Combinatorial Problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations (Proceedings of a Symposium on the Complexity of Computer Computations, March, 1972, Yorktown Heights, NY)*, pages 85–103. Plenum Press, New York, 1972.

[54] Matt Kaufmann and J. Strother Moore. Some Key Research Problems in Automated Theorem Proving for Hardware and Software Verification. *Spanish Royal Academy of Science (RACSAM)*, 98(1):181–195, 2004.

[55] The KIEL Project. Project Homepage, 2004. URL `http://www.informatik.uni-kiel.de/~rt-kiel/`. Kiel Integrated Environment for Layout.

[56] The KIEL Project. Project API Documentation, 2004. URL `http://www.informatik.uni-kiel.de/~rt-kiel/kiel/kiel/doc/`. Kiel Integrated Environment for Layout.

[57] Tobias Kloss. Automatisches Layout von Statecharts unter Verwendung von GraphViz. Diplomarbeit, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, May 2005.

[58] Kay Kossowan. Automatisierte Überprüfung semantischer Modellierungsrichtlinien für Statecharts. Diplomarbeit, Technische Universität Berlin, 2000.

[59] 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.

[60] Peter B. Ladkin. Causal Reasoning about Aircraft Accidents. In Floor Koornneef and Meine van der Meulen, editors, *Computer Safety, Reliability and Security, 19th International Conference, SAFECOMP 2000, Rotterdam, The Netherlands, October 24-27, 2000*, volume 1943 of *Lecture Notes in Computer Science*, pages 344–360. Springer, 2000.

[61] Nancy G. Leveson. *Safeware: System Safety And Computers*. Addison-Wesley, Reading, MA, 1995.

[62] Nancy G. Leveson and Clark S. Turner. Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, 1993.

[63] Jacques-Louis Lions. Ariane 5: Flight 501 failure – report by the inquiry board. Press Release 33, July 1996.

[64] Florian Lüpke. Implementierung eines Statechart-Editors mit layoutbasierten Bearbeitungshilfen. Diplomarbeit, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, June 2005.

[65] Evan Martin and Karen Smiley. Experiment and Comparison of Automated Static Code Analyzers and Automated Dynamic Tests. Technical report, Department of Computer Science, North Carolina State University, 2005.

[66] The MathSAT page, 2005. URL `http://mathsat.itc.it/`.

[67] 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`.

[68] The MathWorks, Inc. *Stateflow and Stateflow Coder User's Guide, Version 6*. Natick, MA, September 2005. URL `http://www.mathworks.com/access/helpdesk/help/pdf_doc/stateflow/sf_ug.pdf`.

[69] Yuri Matiyasevich. Enumerable Sets are Diophantine. *Doklady Akademii Nauk SSSR*, 191:279–282, 1970. English translation in Soviet Mathematics, Doklady, vol. 11, no. 2, 1970.

[70] W. S. McCulloch and W. Pitts. A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[71] G. H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34:1045–1079, September 1955.

[72] David G. Mitchell, Bart Selman, and Hector J. Levesque. Hard and Easy Distributions for SAT Problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–465, 1992.

[73] Edward F. Moore. Gedanken-Experiments on Sequential Machines. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, number 34 in Annals of Mathematical Studies, pages 129–153. Princeton University Press, Princeton, NJ, 1956.

[74] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 39th Design Automation Conference (DAC 2001), Las Vegas*, June 2001.

[75] 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.

[76] Motor Industry Software Reliability Association (MISRA). *MISRA-C:1998. Guidelines for the Use of the C Language in Vehicle Based Software.* Motor Industry Research Association (MIRA), Nuneaton CV10 0TU, UK, 1998.

[77] Miltiadis Moutos, Albrecht Korn, and Carsten Fisel. Guideline-Checker. Studienarbeit, University of Applied Sciences in Esslingen, June 2000.

[78] Martin Mutz. Ein Regel Checker zur Verbesserung des modellbasierten Softwareentwurfs. Toolpräsentation, Technische Universität Braunschweig, 2003. URL `http://www.cs.tu-bs.de/ips/mutz/papers/EKA2003.pdf`.

[79] Martin Mutz. *Eine durchgängige modellbasierte Entwurfsmethodik fÃijr eingebettete Systeme im Automobilbereich.* Dissertation, Technische Universität Braunschweig, 2005.

[80] Martin Mutz and Michaela Huhn. Automated Statechart Analysis for User-defined Design Rules. Technical report, Technische Universität Braunschweig, 2003. URL `http://www.cs.tu-bs.de/ips/mutz/papers/TR2003-10.pdf`.

[81] Robert Nieuwenhuis and Albert Oliveras. Decision procedures for SAT, SAT Modulo Theories and Beyond. The BarcelogicTools. In *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), Jamaica*, December 2005.

[82] Object Management Group. Unified Modeling Language: Superstructure, Version 2.0, Aug 2005. URL `http://www.omg.org/docs/formal/05-07-04.pdf`.

[83] Object Management Group. MOF 2.0/XMI Mapping Specification, v2.1, September 2005. URL `http://www.omg.org/docs/formal/05-09-01.pdf`.

[84] André Ohlhoff. Simulating the Behavior of SyncCharts. Studienarbeit, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, November 2004.

[85] Steve Oualline. *C Elements of Style*. M & T Publishing, Inc., San Mateo, CA, 1992.

[86] David L. Parnas. On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System. In *Proceedings of the 24th National ACM Conference*, pages 379–385, 1969.

[87] 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.

[88] Mojzesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du Premier Congrès de Mathématiciens des Pays Slaves*, pages 92–101, Warszawa, 1929.

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

[90] 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.

[91] Programming Research Ltd., 2005. URL `http://www.programmingresearch.com/`.

[92] M. O. Rabin and D. Scott. Finite Automata and their Decision Problems. *IBM Journal of Research and Development*, 3:114–125, 1959.

[93] The Ricardo Company. Mint – A Style checker for Simulink and Stateflow, 2006. URL `http://www.ricardo.com/engineeringservices/controlelectronics.aspx?page=mint`.

[94] Satisfiability Modulo Theories Competition (SMT-COMP), 2005. URL `http://www.csl.sri.com/users/demoura/smt-comp/2005/`.

[95] Bart Selman, Hector Levesque, and David Mitchell. A New Method for Solving Hard Satisfiability Problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 440–446, San Jose, CA, July 1992.

[96] Bart Selman, Henry Kautz, and Bram Cohen. Local Search Strategies for Satisfiability Testing. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–532. American Mathematical Society, 1996.

[97] Sanjit A. Seshia, Shuvendu K. Lahiri, and Randal E. Bryant. *A User's Guide to UCLID version 1.0*. School of Computer Science, Department of Electrical & Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, June 2003. URL `http://www.cs.cmu.edu/~uclid/userguide.pdf`.

[98] Henry Spencer. The Ten Commandments for C Programmers, 1992. URL `http://www.lysator.liu.se/c/ten-commandments.html`.

[99] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A Cooperating Validity Checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the 14th International Conference on Computer-Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer, 2002.

[100] Sun Microsystems, Inc. Java 2 Platform, Standard Edition, v 1.4.2, API Specification, 2006. URL `http://java.sun.com/j2se/1.4.2/docs/api/`.

[101] Sun Microsystems, Inc. Java Native Interface, 2005. URL `http://java.sun.com/j2se/1.4.2/docs/guide/jni/`.

[102] Sun Microsystems, Inc. Code Conventions for the Java Programming Language, 1997. URL `http://java.sun.com/docs/codeconv/`.

[103] SWIG – Simplified Wrapper and Interface Generator, 2006. URL `http://www.swig.org/`.

[104] TIOBE Software BV – The Coding Standards Company, 2006. URL `http://www.tiobe.com/standards/`.

[105] Edward R. Tufte. *Visual Explanations*. Graphics Press, Cheshire, Connecticut, 1997.

[106] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 2, pages 230–265, 1936.

[107] Mirko Wischer. Ein FileInterface für das KIEL Projekt. Praktikumsbericht, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, 2005.

[108] Mirko Wischer. Textuelle Darstellung und strukturbasiertes Editieren von Statecharts. Diplomarbeit, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, 2006.

[109] Yices, 2005. URL `http://fm.csl.sri.com/yices/`.

[110] Richard Zippel. On Satisfiability. In *Proceedings of the 4th Israel Symposium on Theory of Computing and Systems, ISTCS'96 (Jerusalem, Israel, June 10-12, 1996)*, pages 162–169, Los Alamitos-Washington-Brussels-Tokyo, 1996. IEEE Computer Society Press.

Appendix A

Source Code

As discussed in Section 6.2, this appendix presents the source code of the *KIEL Checking* plug-in, including the implementation of the semantic robustness checks for statecharts. In particular, Appendix A.1 displays the SWIG interface definition file used to generate C++ and Java wrappers for CVC Lite. Next, Appendix A.2 presents the Java source code of the underlying checking infrastructure, which provides for an arbitrary number of checking packages. As illustrated in Figure A.1, all checks extend a common base class (see Appendix A.2.1). Appendix A.3 displays the Java-implementation of the sematic robustness checks: the *Dwelling* check, the *Transition Overlap* check, and the *Race Conditions* check. Finally, Appendix A.4 exhibits the link-up of the CVC Lite binary and library, which includes the recursive parsing of the transition predicates and the forming of appropriate CVC Lite satisfiability queries.

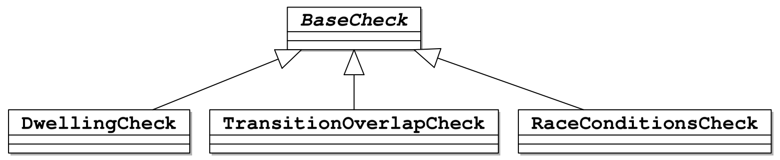


Figure A.1: Partial class diagram showing the semantic robustness checks and their common base class.

A.1 SWIG Interface Definition

The SWIG interface definition file is composed of CVC Lite's annotated C++ header files. It is used to generate C++ and Java wrappers for CVC Lite. Thereby, the CVC Lite library may be used from within Java just as if it was a Java library.

```
%module JavaCVC
%{
#include "expr_op.h"
#include "vc.h"
#include "exception.h"
#include "command_line_flags.h"
#include "debug.h"
#include "expr.h"
#include "type.h"
#include "statistics.h"
#include "theorem.h"
#include "proof.h"
#include "rational.h"
#include "theory_arith.h"

%}
%include typemaps.i

%include "carrays.i"
%array_class(int, CIntArray);
%include std_map.i
%include std_pair.i
%include std_string.i
%include std_vector.i

%rename(assign) CVCL::Expr::operator=;
%rename(get) CVCL::Expr::operator[];
%rename(equals) CVCL::Expr::operator==;
namespace std{
        %template(CVectorExpr)  vector<CVCL::Expr>;
        %template(CVectorType)  vector<CVCL::Type>;
}

namespace CVCL
{

  class Context;

  class Statistics;

  // The commonly used kinds and the kinds needed by the parser.  All
  // these kinds are registered by the ExprManager and are readily
  // available for everyone else.
typedef enum {
  NULL_KIND = 0,
  // Generic LISP kinds for representing raw parsed expressions
  RAW_LIST, //!< May have any number of children >= 0
  //! Identifier is (ID (STRING_EXPR "name"))
  ID,
  // Leaf exprs
  STRING_EXPR,
  RATIONAL_EXPR,
  TRUE,
  FALSE,
  // Types
  BOOLEAN,
//   TUPLE_TYPE,
  ARROW,
  // The "type" of any expression type (as in BOOLEAN : TYPE).
  TYPE,
  // Declaration of new (uninterpreted) types: T1, T2, ... : TYPE
  // (TYPEDECL T1 T2 ...)
  TYPEDECL,
  // Declaration of a defined type T : TYPE = type === (TYPEDEF T type)
  TYPEDEF,

  // Equality
  EQ,
  NEQ,

  // Propositional connectives
  NOT,
  AND,
  OR,
  XOR,
  IFF,
  IMPLIES,
  //  BOOL_VAR, //!< Boolean variables are treated as 0-ary predicates

  // Propositional relations (for circuit propagation)
  AND_R,
  IFF_R,
  ITE_R,

  // (ITE c e1 e2) == IF c THEN e1 ELSE e2 ENDIF, the internal
  // representation of the conditional.  Parser produces (IF ...).
  ITE,

  // Quantifiers
  FORALL,
  EXISTS,

  // Uninterpreted function
  UFUNC,
  // Application of a function
  APPLY,

  // Top-level Commands
```

```
    ASSERT,
    QUERY,
    CHECKSAT,
    CONTINUE,
    RESTART,
    DBG,
    TRACE,
    UNTRACE,
    OPTION,
    HELP,
    TRANSFORM,
    PRINT,
    CALL,
    ECHO,
    INCLUDE,
    DUMP_PROOF,
    DUMP_ASSUMPTIONS,
    DUMP_SIG,
    DUMP_TCC,
    DUMP_TCC_ASSUMPTIONS,
    DUMP_TCC_PROOF,
    DUMP_CLOSURE,
    DUMP_CLOSURE_PROOF,
    WHERE,
    ASSERTIONS,
    ASSUMPTIONS,
    COUNTEREXAMPLE,
    COUNTERMODEL,
    PUSH,
    POP,
    POPTO,
    PUSH_SCOPE,
    POP_SCOPE,
    POPTO_SCOPE,
    CONTEXT,
    FORGET,
    GET_TYPE,
    CHECK_TYPE,
    GET_CHILD,
    SUBSTITUTE,
    SEQ,

    // Kinds used mostly in the parser

    TCC,
    // Variable declaration (VARDECL v1 v2 ... v_n type).  A variable
    // can be an ID or a BOUNDVAR.
    VARDECL,
    // A list of variable declarations (VARDECLS (VARDECL ...) (VARDECL ...) ...)
    VARDECLS,

    BOUND_VAR,
    BOUND_ID,

    SUBTYPE,

    IF,
    IFTHEN,
    ELSE,

    COND,

    LET,
    LETDECLS,
    LETDECL,
    // Lambda-abstraction LAMBDA (<vars>) : e  === (LAMBDA <vars> e)
    LAMBDA,
    // Symbolic simulation operator
    SIMULATE,

    // Constant declaration x : type = e  === (CONSTDEF x type e)
    CONSTDEF,

    CONST,
    VARLIST,
    UCONST,

    // User function definition f(args) : type = e === (DEFUN args type e)
    // Here 'args' are bound var declarations
    DEFUN,

    // Kinds for proof terms
    PF_APPLY,
    PF_HOLE,

    //Skolem variable
    SKOLEM_VAR,
    //! Must always be the last kind
    LAST_KIND
  } Kind;

    typedef enum {
      REAL = 3000,
      INT,
      SUBRANGE,

      UMINUS,
      PLUS,
    MINUS,
    MULT,
    DIVIDE,
    POW,
    INTDIV,
    MOD,
    LT,
    LE,
    GT,
    GE,
    IS_INTEGER,
    NEGINF,
    POSINF,
    DARK_SHADOW,
    GRAY_SHADOW
  } ArithKinds;

  %nodefault;
  class Proof{
  public:
    std::string toString() const {
      std::ostringstream ss;
      ss<<(*this);
      return ss.str();
    }
  };
  class Rational{
  public:
    Rational getNumerator() const;
    Rational getDenominator() const;

    // Equivalent to (getDenominator() == 1), but possibly more efficient
    bool isInteger() const;
    // Convert to int; defined only on integer values
    int getInt() const;
    Rational();
    // Copy constructor
    Rational(const Rational &n);
    Rational(int n, int d = 1);
  };
  class CLFlags {
  public:
    size_t countFlags(const std::string& name);
    size_t countFlags(const std::string& name,
                      std::vector<std::string>& names);
    void setFlag(const std::string& name, bool b);
    void setFlag(const std::string& name, int i);
    //void setFlag(const std::string& name, const std::string& s);
    void setFlag(const std::string& name, const char* s);
    void setFlag(const std::string& name,
                 const std::pair<std::string,bool>& p);
    void setFlag(const std::string& name,
                 const std::vector<std::pair<std::string,bool> >& sv);

    %extend{
      ~CLFlags()
        {
          delete self;
        }
    }


  };
  typedef long long unsigned ExprIndex;
  //inline bool operator==(const Expr& e1, const Expr& e2) {
  // Comparing pointers (equal expressions are always shared)
  //  return e1.d_expr == e2.d_expr;
  //}
  class Expr;
  class Op;
  class ExprManager{
  public:
    //Expr rebuildExpr(const Expr& e) { return Expr(rebuild(e)); }
  };

  %typemap(javacode) Expr %{
    public boolean equals (Object o)
      {
        if (o == null || !(o instanceof Expr)) return false;
        return equal (this, (Expr)o);
      }

    public int hashCode ()
      {
        return (int)computeHashCode (this);
      }
    %}

  class Expr {
  public:
    ExprIndex getIndex() const;
    Expr(): d_expr(NULL) {}
    Expr(const Expr& e);
    const Rational & getRational();
    int arity() const;
    int getKind() const;

    // These constructors grab the ExprManager from the Op or the first
  // child.  The operator and all children must belong to the same
  // ExprManager.
  Expr(const Op& op, const Expr& child);
  Expr(const Op& op, const Expr& child0, const Expr& child1);
  Expr(const Op& op, const Expr& child0, const Expr& child1,
       const Expr& child2);
  Expr(const Op& op, const Expr& child0, const Expr& child1,
       const Expr& child2, const Expr& child3);
  Expr(const Op& op, const std::vector<Expr>& children,
       ExprManager* em = NULL);



  //bool hasOp() const;
  //const Expr& getOp() const;
  bool isFalse() const { return getKind() == FALSE; }
  bool isTrue() const { return getKind() == TRUE; }
  bool isBoolConst() const { return isFalse() || isTrue(); }
  bool isVar() const;
  bool isString() const;
  bool isClosure() const;
  bool isQuantifier() const;
  bool isLambda() const;
  //! Expr is an application of a LAMBDA-term to arguments
  bool isApply() const;
  /* bool isRecord() const;
  bool isRecordAccess() const;
  bool isTupleAccess() const;*/
  /*size_t isGeneric() const;
    bool isGeneric(size_t idx) const;*/

  bool isEq() const { return getKind() == EQ; }
  bool isNot() const { return getKind() == NOT; }
  bool isAnd() const { return getKind() == AND; }
  bool isOr() const { return getKind() == OR; }
  bool isITE() const { return getKind() == ITE; }
  bool isIff() const { return getKind() == IFF; }
  bool isImpl() const { return getKind() == IMPLIES; }

  bool isForall() const { return getKind() == FORALL; }
  bool isExists() const { return getKind() == EXISTS; }
  // Arith testers
  bool isRational() const { return getKind() == RATIONAL_EXPR; }

  bool isNull() const;

  Expr eqExpr(const Expr& right) const;
  Expr notExpr() const;
  Expr negate() const; // avoid double-negatives
  Expr andExpr(const Expr& right) const;
  //static Expr andExpr(std::vector <Expr>& children);
  Expr orExpr(const Expr& right) const;
  //static Expr orExpr(std::vector <Expr>& children);
  Expr iteExpr(const Expr& thenpart, const Expr& elsepart) const;
  Expr iffExpr(const Expr& right) const;
  Expr impExpr(const Expr& right) const;

  Expr& operator=(const Expr& e);
  //   inline bool operator==(const Expr& e1, const Expr& e2) {
  //     return e1.d_expr == e2.d_expr;
  //   };

  const Expr& operator[](int i) const;
  std::string toString() const;
  %extend{
    bool equal(const Expr& e1, const Expr& e){
      return CVCL::operator==(e1, e);
    }
    size_t computeHashCode (const Expr& e)
      {
        return e.getEM ()->hash (e);
      }
    //         ~Expr ()
    //            {
    //                  delete self;
    //            };
  }
};



class Op{
public:
  std::string toString() const;
  Op(const Op& op);
  Op(ExprManager *em, int kind);
  int getKind() const { return d_kind; }

  Op(int kind);

  //bool hasExpr() const;

  // Return the expr associated with this operator if applicable.  It
  // is an error to call this when Op has no associated Expr.
  const Expr& getExpr() const {
    return d_expr;
  }
  /*%extend{
    Op getOp(const Expr& e){
      return CVCL::getOp(e);
    }
    //         ~Op()
    //            {
    //                delete self;
    //            };
    }*/
};


class Type {
public:
  Type ();
  std::string toString() const { return getExpr().toString(); }
  const Expr& getExpr() const { return d_expr; }
  int arity() const { return d_expr.arity(); }
  //     %extend{
  //       ~Type ()
  //    {
  //        delete self;
  //    };
  //    }
};
class Statistics{
public:
  %extend{
    ~Statistics(){
      delete self;
    };
  }
};
class Theorem3{
public:
  bool isNull() const { return d_thm.isNull(); }

  // True if theorem is of the form t=t' or phi iff phi'
  bool isRewrite() const { return d_thm.isRewrite(); }
  //bool isAxiom() const { return d_thm.isAxiom(); }

  // Return the theorem value as an Expr
  const Expr& getExpr() const { return d_thm.getExpr(); }
  const Expr& getLHS() const { return d_thm.getLHS(); }
  const Expr& getRHS() const { return d_thm.getRHS(); }

  // Return the proof of the theorem.  If running without proofs,
  // return the Null proof.
  const Proof& getProof() const { return d_thm.getProof(); }

  // Return the lowest scope level at which this theorem is valid.
  // Value -1 means no information is available.
  // int getScope() const;

  // Test if we are running in a proof production mode and with assumptions
  bool withProof() const { return d_thm.withProof(); }
  bool withAssumptions() const { return d_thm.withAssumptions(); }

  // Printing
  std::string toString() const;

  // For debugging
  void printx() const { d_thm.printx(); }
  void print() const { d_thm.print(); }

  //! Check if the theorem is a literal
  bool isAbsLiteral() const { return d_thm.isLiteral(); }

  %extend{
    ~Theorem3(){
      delete self;
    };
  }
};
class Context {
public:
  %extend{
    ~Context(){
      delete self;
    }
  }
};
class ValidityChecker {

public:

  static CLFlags createFlags ();
  static ValidityChecker* create ();
  static ValidityChecker* create (CLFlags flags);

  Type boolType();
  Type realType();
  Type intType();
  Type subrangeType(const Expr& l, const Expr& r);
  //! Creates a subtype defined by the given predicate (a LAMBDA-term)
  /*!
   * \param pred is a term (LAMBDA (x: T): p(x)), and the resulting
   * type is a subtype of T whose elements x are those satisfying the
   * predicate p(x).
```

```
 */
Type subtypeType(const Expr& pred);
// Tuple types
//! 2-element tuple
Type tupleType(const Type& type0, const Type& type1);
//! 3-element tuple
Type tupleType(const Type& type0, const Type& type1,
               const Type& type2);
//! n-element tuple (from a vector of types)
Type tupleType(const std::vector<Type>& types);

// Record types
//! 1-element record
Type recordType(const std::string& field, const Type& type);
//! 2-element record
/*! Fields will be sorted automatically */
Type recordType(const std::string& field0, const Type& type0,
                const std::string& field1, const Type& type1);
//! 3-element record
/*! Fields will be sorted automatically */
Type recordType(const std::string& field0, const Type& type0,
                const std::string& field1, const Type& type1,
                const std::string& field2, const Type& type2);
//! n-element record (fields and types must be of the same length)
/*! Fields will be sorted automatically */
Type recordType(const std::vector<std::string>& fields,
                const std::vector<Type>& types);

//! Create an array type (ARRAY typeIndex OF typeData)
Type arrayType(const Type& typeIndex, const Type& typeData);

//! Create a function type ([typeDom -> typeRan])
Type funType(const Type& typeDom, const Type& typeRan);
Type funType(const std::vector<Type>& typeDom, const Type& typeRan);

//! Create named user-defined uninterpreted type
Type createType(const std::string& typeName);
//! Create named user-defined interpreted type (type abbreviation)
Type createType(const std::string& typeName, const Type& def);
//! Lookup a user-defined (uninterpreted) type by name
Type lookupType(const std::string& typeName);

ExprManager* getEM();

Expr varExpr(const std::string& name, const Type& type);

Expr boundVarExpr(const std::string& name,
                  const std::string& uid,
                  const Type& type);

    //! Get the variable associated with a name, and its type
    /*!
      \param name is the variable name
      \param type is where the type value is returned

      \return a variable by the name. If there is no such Expr, a NULL \
      Expr is returned.
    */
    Expr lookupVar(const std::string& name, Type* type);

    //! Get the type of the Expr.
    Type getType(const Expr& e);
    //! Get the largest supertype of the Expr.
    Type getBaseType(const Expr& e);

  Expr listExpr(const std::vector<Expr>& kids);
  //! Overloaded version of listExpr with one argument
  Expr listExpr(const Expr& e1);
  //! Overloaded version of listExpr with two arguments
  Expr listExpr(const Expr& e1, const Expr& e2);
  //! Overloaded version of listExpr with three arguments
  Expr listExpr(const Expr& e1, const Expr& e2, const Expr& e3);
  //! Overloaded version of listExpr with string operator and many arguments
  Expr listExpr(const std::string& op,
                      const std::vector<Expr>& kids);
  //! Overloaded version of listExpr with string operator and one argument
  Expr listExpr(const std::string& op, const Expr& e1);
  //! Overloaded version of listExpr with string operator and two arguments
  Expr listExpr(const std::string& op, const Expr& e1, const Expr& e2);
  //! Overloaded version of listExpr with string operator and three arguments
Expr listExpr(const std::string& op, const Expr& e1, const Expr& e2, const Expr& e3);


    //! Prints e to the standard output
    void printExpr(const Expr& e);
    //! Parse an expression using a Theory-specific parser
    Expr parseExpr(const Expr& e);

    Expr importExpr(const Expr& e);
    //! Import the Type from another instance of ValidityChecker
    /*! \sa getType() */
    Type importType(const Type& t);

    //! Return TRUE Expr
    Expr trueExpr();
    //! Return FALSE Expr
    Expr falseExpr();
  //! Create negation
  Expr notExpr(const Expr& child);
  //! Create 2-element conjunction
  Expr andExpr(const Expr& left, const Expr& right);
  //! Create n-element conjunction
  Expr andExpr(std::vector<Expr>& children);
  //! Create 2-element disjunction
  Expr orExpr(const Expr& left, const Expr& right);
  //! Create n-element disjunction
  Expr orExpr(std::vector<Expr>& children);
  //! Create Boolean implication
  Expr impliesExpr(const Expr& hyp, const Expr& conc);
  //! Create left IFF right (boolean equivalence)
  Expr iffExpr(const Expr& left, const Expr& right);
  //! Create an equality expression.
  /*!
    The two children must have the same type, and cannot be of type
    Boolean.
  */
  Expr eqExpr(const Expr& child0, const Expr& child1);
  //! Create IF ifpart THEN thenpart ELSE elsepart ENDIF
  /*!
    \param ifpart must be of type Boolean.
    \param thenpart and \param elsepart must have the same type, which will
    also be the type of the ite expression.
  */
  Expr iteExpr(const Expr& ifpart, const Expr& thenpart,
               const Expr& elsepart);

  /*@}*/ // End of Core expression methods

 Op createOp(const std::string& name, const Type& type);

  //! Unary function application (op must be of function type)
  Expr funExpr(const Op& op, const Expr& child);
  //! Binary function application (op must be of function type)
  Expr funExpr(const Op& op, const Expr& left, const Expr& right);
  //! Ternary function application (op must be of function type)
  Expr funExpr(const Op& op, const Expr& child0,
               const Expr& child1, const Expr& child2);
  //! n-ary function application (op must be of function type)
  Expr funExpr(const Op& op, const std::vector<Expr>& children);

  Expr ratExpr(int n, int d = 1);
  //! Create a rational number with numerator n and denominator d.
  /*!
    Here n and d are given as strings.  They are converted to
    arbitrary-precision integers according to the given base.
  */
  Expr ratExpr(const std::string& n, const std::string& d, int base);

  Expr ratExpr(const std::string& n, int base = 10);

  //! Unary minus.
  Expr uminusExpr(const Expr& child);

  //! Create 2-element sum (left + right)
  Expr plusExpr(const Expr& left, const Expr& right);
  //! Make a difference (left - right)
  Expr minusExpr(const Expr& left, const Expr& right);
  //! Create a product (left * right)
  Expr multExpr(const Expr& left, const Expr& right);
  //! Create a power expression (x ^ n); n must be integer
  Expr powExpr(const Expr& x, const Expr& n);
  Expr divideExpr(const Expr& numerator, const Expr& denominator);

  //! Create (left < right)
  Expr ltExpr(const Expr& left, const Expr& right);
  //! Create (left <= right)
  Expr leExpr(const Expr& left, const Expr& right);
  //! Create (left > right)
  Expr gtExpr(const Expr& left, const Expr& right);
  //! Create (left >= right)
  Expr geExpr(const Expr& left, const Expr& right);
  //! Create FLOOR(expr)
  //Expr floorExpr(const Expr& expr);

  //! Create a 1-element record value (# field := expr #)
  /*! Fields will be sorted automatically */
  Expr recordExpr(const std::string& field, const Expr& expr);
  //! Create a 2-element record value (# field0 := expr0, field1 := expr1 #)
  /*! Fields will be sorted automatically */
  Expr recordExpr(const std::string& field0, const Expr& expr0,
                  const std::string& field1, const Expr& expr1);
  //! Create a 3-element record value (# field_i := expr_i #)
  /*! Fields will be sorted automatically */
  Expr recordExpr(const std::string& field0, const Expr& expr0,
                  const std::string& field1, const Expr& expr1,
                  const std::string& field2, const Expr& expr2);
  //! Create an n-element record value (# field_i := expr_i #)
  /*!
   * \param fields
   * \param exprs must be the same length as fields
   *
   * Fields will be sorted automatically
   */
  Expr recordExpr(const std::vector<std::string>& fields,
                  const std::vector<Expr>& exprs);
```

```
//! Create record.field (field selection)
/*! Create an expression representing the selection of a field from
  a record. */
Expr recSelectExpr(const Expr& record, const std::string& field);

//! Record update; equivalent to "record WITH .field := newValue"
/*! Notice the `.' before field in the presentation language (and
  the comment above); this is to distinguish it from datatype
  update.
*/
Expr recUpdateExpr(const Expr& record, const std::string& field,
                   const Expr& newValue);

/*@}*/ // End of Record expression methods

//! Create an expression array[index] (array access)
/*! Create an expression for the value of array at the given index */
Expr readExpr(const Expr& array, const Expr& index);

//! Array update; equivalent to "array WITH index := newValue"
Expr writeExpr(const Expr& array, const Expr& index,
               const Expr& newValue);

/*@}*/ // End of Array expression methods

//! Tuple expression
Expr tupleExpr(const std::vector<Expr>& exprs);
//! Tuple select; equivalent to "tuple.n", where n is an numeral (e.g. tup.5)
Expr tupleSelectExpr(const Expr& tuple, int index);
//! Tuple update; equivalent to "tuple WITH index := newValue"
Expr tupleUpdateExpr(const Expr& tuple, int index,
                     const Expr& newValue);
//! Datatype update; equivalent to "dt WITH accessor := newValue"
//Expr datatypeUpdateExpr(const Expr& dt, const Expr& accessor,
//const Expr& newValue);
//! Universal quantifier
Expr forallExpr(const std::vector<Expr>& vars, const Expr& body);
//! Existential quantifier
Expr existsExpr(const std::vector<Expr>& vars, const Expr& body);
//! Lambda-expression
Op lambdaExpr(const std::vector<Expr>& vars, const Expr& body);
//! Symbolic simulation expression
/*!
 * \param f is the next state function (LAMBDA-expression)
 * \param s0 is the initial state
 * \param inputs is the vector of LAMBDA-expressions representing
 * the sequences of inputs to f
 * \param n is a constant, the number of cycles to run the simulation.
 */
Expr simulateExpr(const Expr& f, const Expr& s0,
                  const std::vector<Expr>& inputs,
                  const Expr& n);

/*@}*/ // End of Other expression methods

//! Assert a new formula in the current context.
/*! The formula must have Boolean type.
 */
void assertFormula(const Expr& e);

//! Simplify e with respect to the current context
Expr simplify(const Expr& e);

Theorem3 simplifyThm(const Expr& e);

void printV(const Expr& e);

bool query(const Expr& e, bool tcc = false) throw(CVCL::Exception);

void getCounterExample(std::vector<Expr>& assumptions,
                       bool inOrder=true);

bool inconsistent(std::vector<Expr>& assumptions);

//! Returns the proof term for the last proven query
/*! If there has not been a successful query, it should return a NULL proof
 */
Proof getProof();

//! Get all assumptions made in this and all previous contexts.
/*! \param assumptions should be empty on entry.
 */
void getAssumptions(std::vector<Expr>& assumptions);

void getAssumptionsUsed(std::vector<Expr>& assumptions);

//! Returns the TCC of the last assumption or query
/*! Returns Null if no assumptions or queries were performed. */
const Expr& getTCC();
//! Return the set of assertions used in the proof of the last TCC
void getAssumptionsTCC(std::vector<Expr>& assumptions);
//! Returns the proof of TCC of the last assumption or query
/*! Returns Null if no assumptions or queries were performed. */
const Proof& getProofTCC();
//! After successful query, return its closure |- Gamma => phi
/*! Turn a valid query Gamma |- phi into an implication
 * |- Gamma => phi.
```

```
     *
     * Returns Null if last query was invalid.
     */
    const Expr& getClosure();
    //! Construct a proof of the query closure |- Gamma => phi
    /*! Returns Null if last query was Invalid. */
    const Proof& getProofClosure();

    /*@}*/ // End of Validity checking methods

  //! Returns the current stack level.  Initial level is 0.
  int stackLevel() = 0;

  //! Checkpoint the current context and increase the scope level
  void push() = 0;

  //! Restore the current context to its state at the last checkpoint
  void pop() = 0;

  //! Restore the current context to the given stackLevel.
  /*!
    \param stackLevel should be greater than or equal to 0 and less
    than or equal to the current scope level.
  */
  void popto(int stackLevel) = 0;

  //! Returns the current scope level.  Initially, the scope level is 1.
  int scopeLevel() = 0;

  /*! @brief Checkpoint the current context and increase the
   * <strong>internal</strong> scope level.  Do not use unless you
   * know what you're doing!
   */
  void pushScope() = 0;

  /*! @brief Restore the current context to its state at the last
   * <strong>internal</strong> checkpoint.  Do not use unless you know
   * what you're doing!
   */
  void popScope() = 0;

  //! Restore the current context to the given scopeLevel.
  /*!
    \param scopeLevel should be less than or equal to the current scope level.

    If scopeLevel is less than 1, then the current context is reset
    and the scope level is set to 1.
  */
  void poptoScope(int scopeLevel) = 0;

  //! Get the current context
  Context* getCurrentContext() = 0;

    /*@}*/ // End of Context methods

    Statistics& getStatistics();
    //! Print collected statistics to stdout
    void printStatistics();
    %extend{
      ValidityChecker ()
        {
          return CVCL::ValidityChecker::create();
        };
      ValidityChecker (CLFlags flags)
        {
          return CVCL::ValidityChecker::create (flags);
        };
    }

  };
}
```

A.2 Checking Infrastructure

A.2.1 BaseCheck

```
//$Id: BaseCheck.java,v 1.2 2006/04/12 15:00:27 gsc Exp $
package kiel.checking;

import kiel.dataStructure.GraphicalObject;

/**
 * <p>Description: This interface defines a method to be implemented by all
 * custom robustness checks.</p>
 * <p>Copyright: (c) 2006</p>
 * <p>Company: Uni Kiel</p>.
 * @author <a href="mailto:kbe@informatik.uni-kiel.de">Ken Bell</a>
 * @author <a href="mailto:gsc@informatik.uni-kiel.de">Gunnar Schaefer</a>
 * @version $Revision: 1.2 $ last modified $Date: 2006/04/12 15:00:27 $
 *
 */
public abstract class BaseCheck {

    /**
     * True, if the check is enabled.
     */
    private boolean enabled = true;

    /**
     * This method is called from a @see RobustnessChecker.
     * @param checker The checker where the @see RobustnessProblems will be
     * added.
     * @param o The current object in the depth first search of the statechart.
     */
    public abstract void apply(StateChartCheckerBase checker,
            GraphicalObject o);

    /**
     * Determines if this rule should be invoked on the
     * current @see GraphicalObject.
     * @param o The current object in the depth first traversal.
     * @return Returns true if the rule should be applied to object o.
     */
    public abstract boolean isValidTarget(final GraphicalObject o);

    /**
     * .
     * @param isEnabled .
     */
    public final void setEnabled(final boolean isEnabled) {
        this.enabled = isEnabled;
    }

    /**
     * .
     * @return True, if the check is enabled.
     */
    public final boolean isEnabled() {
        return this.enabled;
    }

    /**
     * Returns the name of the rule.
     * @return The name of the rule.
     */
    public final String getRuleName() {
        String name;
        name = this.getClass().getName();
        return name.substring(name.lastIndexOf('.') + 1);
    }

    /**
     * Sets the robustness properties.
     * @param props The robustness properties to be set.
     */
    public void setProperties(final CheckingProperties props) {

    }

    /**
     * Sets the CVCL process.
     * @param cvclProc The process object.
     */
    public void setCvclProcess(final Process cvclProc) {

    }
}
```

A.2.2 StateChartCheckerBase

```java
//$Id: StateChartCheckerBase.java,v 1.2 2006/04/12 15:00:27 gsc Exp $
package kiel.checking;

import java.io.File;
import java.net.URL;
import java.net.URLClassLoader;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.jar.JarFile;
import java.util.jar.Manifest;

import javax.swing.JComponent;
import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;

import kiel.dataStructure.GraphicalObject;
import kiel.dataStructure.StateChart;
import kiel.util.IStateChartVisitor;
import kiel.util.KielFrameRefresh;
import kiel.util.LogFile;
import kiel.util.StateChartDepthFirstAdapter;

/**
 *<p> Description: This class manages the checking of a statechart. It uses a
 * @see StateChartDepthFirstAdapter to visit all nodes of the data structure.
 * The checks to be performed are loaded dynamically.</p>
 *
 * <p> Copyright: (c) 2006 </p>
 * <p> Company: Uni Kiel </p>.
 * @author <a href="mailto:kbe@informatik.uni-kiel.de">Ken Bell</a>
 * @author <a href="mailto:gsc@informatik.uni-kiel.de">Gunnar Schaefer</a>
 * @version $Revision: 1.2 $ last modified $Date: 2006/04/12 15:00:27 $
 */
public class StateChartCheckerBase extends AbstractTableModel implements
        IStateChartVisitor {

    /**
     * The depth first adapter to visit all nodes of the datastructure.
     */
    private StateChartDepthFirstAdapter depthfirst;

    /**
     *
     */
    private boolean dispatchRegions = true;

    /**
     * Holds instances of dynamically loaded checks.
     */
    private ArrayList rules;

    /**
     * Stores the found errors.
     */
    private ArrayList errors;

    /**
     * Stores generated warnings.
     */
    private ArrayList warnings;

    /**
     * Directory containing the jar file with the checks to be loaded.
     */
    private String rulePackage;

    /**
     *
     */
    private int robustnessLevel = 0;

    /**
     * True in GUI version, false in commandline version.
     */
    private boolean showResultsInGUI = false;

    /**
     *
     */
    private CheckerOutputGUI gui;

    /**
     *
     */
    private int tabbedIndex;

    /**
     *
     */
    private Process cvclProcess;

    /**
     *
     */
    private StateChart stateChart;

    /**
     * The constructor.
     *
     * @param dir The sourcedir of the jar file.
     * @param doShow True in GUI version, false in commandline version.
     */
    public StateChartCheckerBase(final String dir, final boolean doShow) {
        KielFrameRefresh.register(this);
        cvclProcess = null;
        rules = new ArrayList();
        rulePackage = dir;
        errors = new ArrayList();
        warnings = new ArrayList();
        showResultsInGUI = doShow;
        if (showResultsInGUI) {
            gui = new CheckerOutputGUI(this);
            setTextInLabel();
        }
    }

    /**
     * The constructor with an extra process argument.
     *
     * @param dir The sourcedir of the jar file.
     * @param doShow True in GUI version, false in commandline version.
     * @param cvclProc The CVCl process.
     */
    public StateChartCheckerBase(final String dir, final boolean doShow,
            final Process cvclProc) {
        KielFrameRefresh.register(this);
        cvclProcess = cvclProc;
        rules = new ArrayList();
        rulePackage = dir;
        errors = new ArrayList();
        warnings = new ArrayList();
        showResultsInGUI = doShow;
        if (showResultsInGUI) {
            gui = new CheckerOutputGUI(this);
            setTextInLabel();
        }
    }

    /**
     * Performs a depth first search to visit all subnodes of the passed node.
     *
     * @param chart The current StateChart.
     * @return Returns True if no errors and no warnings were found.
     */
    public final boolean checkDataStructure(final StateChart chart) {
        errors.clear();
        warnings.clear();
        stateChart = chart;
        depthfirst = new StateChartDepthFirstAdapter();
        depthfirst.traverse(this, stateChart.getRootNode(), dispatchRegions);
        if (showResultsInGUI) {
            this.fireTableDataChanged();
            setTextInLabel();
        }
        return ((getErrors().size() == 0) && (getWarnings().size() == 0));
    }

    /**
     *  @param o The object to perform all checks on.
     */
    public final void dispatch(final GraphicalObject o) {
        BaseCheck rule;
        for (int i = 0; i < rules.size(); i++) {
            rule = (BaseCheck) rules.get(i);
            if ((rule.isEnabled()) && (rule.isValidTarget(o))) {
                rule.apply(this, o);
            }
        }
    }

    /**
     * Returns the type of the StateChart.
     * @return The type of the StateChart.
     */
    public final String getStateChartType() {
        return stateChart.getModelSource();
    }

    /**
     * @return The collection of all warnings that occured during the last run
     * of checkDataStructure.
     */
    public final Collection getWarnings() {
        return warnings;
    }

    /**
     *
     * @return The collection of all errors that occured during the last run of
     * checkDataStructure.
     */
    public final Collection getErrors() {
        return errors;
    }

    /**
     *
     * @return Does the actual instance dispatch objects of type Region?
     */
    public final boolean isDispatchRegions() {
        return dispatchRegions;
    }

    /**
     *
     * @param doDispatch Denote if the instance should dispatch Region.
     */
    public final void setDispatchRegions(final boolean doDispatch) {
        this.dispatchRegions = doDispatch;
    }

    /**
     * This method loads the rules from the directory passed to the constructor.
     * The rules are dynamically load from a jar file and instances of the found
     * checks are stored in the
     */
    public final void loadRules() {

        rules.clear();


        CheckingProperties.getRobustnessLog().log(LogFile.DETAIL,
                "Starting to load rules from: " + this.rulePackage);

        String classname = "";
        Class c;
        Object o;
        JarFile jar = null;
        Manifest m = null;
        try {
            jar = new JarFile(rulePackage);
            if (jar != null) {
                m = jar.getManifest();
                if (m == null) {
                    CheckingProperties.getRobustnessLog().log(LogFile.ERROR,
                            "Jarfile contains no manifest.");
                }
                Iterator iter = m.getEntries().keySet().iterator();
                while (iter.hasNext()) {
                    classname = (String) iter.next();
                    URL[] url = {(new File(jar.getName())).toURL()};
                    URLClassLoader ucl = new URLClassLoader(url, this
                            .getClass().getClassLoader());
                    c = ucl.loadClass(classname);
                    o = c.newInstance();
                    if (o instanceof BaseCheck && cvclProcess != null) {
                        ((BaseCheck) o).setCvclProcess(cvclProcess);
                    }
                    if (o != null) {
                        rules.add(o);
                        CheckingProperties.getRobustnessLog().log(
                                LogFile.DETAIL, "Loading of rule:" + classname
                                + " successful.");
                    }
                }
            } else {
                CheckingProperties.getRobustnessLog().log(LogFile.ERROR,
                        "No valid jarfile specified.");
            }
            loadRuleStatuses();
        } catch (Exception e) {
            CheckingProperties.getRobustnessLog().log(LogFile.ERROR,
                    "Rule loading failed.");
        }
    }

    /**
     * Loads the statuses of all rules in the current rulePackage.
     */
    private void loadRuleStatuses() {
        BaseCheck b;
        for (int i = 0; i < rules.size(); i++) {
            b = (BaseCheck) rules.get(i);
            b.setEnabled(CheckingProperties.getRuleStatus(
                this.rulePackage.substring(
                    this.rulePackage.lastIndexOf(File.separatorChar) + 1,
                    this.rulePackage.lastIndexOf('.'))
                    + "." + b.getRuleName()));
        }
    }

    /**
     *
     * @return The level of semantic robustness checks
     */
    public final int getRobustnessLevel() {
        return robustnessLevel;
    }

    /**
```

```
 * Set the level of the checks manually.
 * @param level The level of the checks.
 */
public final void setRobustnessLevel(final int level) {
    this.robustnessLevel = level;
}

/**
 * This method prints the errors and warnings, that were generated during
 * checkDataStructure.
 * @param w The destination writer, where the messages will be written to.
 **/
public final void printMessages(final Object w) {

    CheckingProblem r;

    for (int i = 0; i < errors.size(); i++) {
        r = (CheckingProblem) errors.get(i);
        CheckingProperties.getSemLog().log(LogFile.ERROR,
                "Error " + (i + 1) + ": " + r.getProblem());
    }
    for (int i = 0; i < warnings.size(); i++) {
        r = (CheckingProblem) warnings.get(i);
        CheckingProperties.getSemLog().log(LogFile.ERROR,
                "Warning " + (i + 1) + ": " + r.getProblem());
    }
}

/**
 * @return One column is needed for displaying the problems in a table.
 */
public final int getColumnCount() {
    return 1;
}

/**
 * The count of rows is calculated by adding the size of the errors and
 * warnings vector.
 * @return The count of rows the table has to display.
 */
public final int getRowCount() {
    return errors.size() + warnings.size();
}

/**
 * This method transforms the given cell coordinates to the appropriate
 * robustness problem and returns the message of the problem.
 * @param rowIndex Th
 * @param colIndex .
 * @return The text to be displayed in the cell with the given coordinates.
 */
public final Object getValueAt(final int rowIndex, final int colIndex) {
    String result;
    CheckingProblem p;
    if (rowIndex >= errors.size()) {
        p = (CheckingProblem) (warnings.get(rowIndex - errors.size()));
        if (CheckingProperties.showClassification()) {
            if (CheckingProperties.showCounters()) {
                result = "[Warning " + (rowIndex - errors.size() + 1)
                        + "] ";
            } else {
                result = "[Warning] ";
            }
        } else {
            if (CheckingProperties.showCounters()) {
                result = "[" + (rowIndex - errors.size() + 1) + "] ";
            } else {
                result = "";
            }
        }
    } else {
        p = (CheckingProblem) (errors.get(rowIndex));
        if (CheckingProperties.showClassification()) {
            if (CheckingProperties.showCounters()) {
                result = "[Error " + (rowIndex + 1) + "] ";
            } else {
                result = "[Error] ";
            }
        } else {
            if (CheckingProperties.showCounters()) {
                result = "[" + (rowIndex + 1) + "] ";
            } else {
                result = "";
            }
        }
    }

    return result + p.getProblem();
}

/**
 * @param col .
 * @return The string, which is shown in the header of the outputtable.
 */
public final String getColumnName(final int col) {
    return "Messages";
}
```

```java
/**
 *
 * @param index The index of the Problem.
 * @return The Graphicalobject of the Problem.
 *
 */
public final ArrayList getProblemsByIndex(final int index) {
    CheckingProblem p;

    if (index > (errors.size() + warnings.size())) {
        return null;
    } else {

        if (index < errors.size()) {
            p = (CheckingProblem) errors.get(index);
        } else {
            p = (CheckingProblem) warnings.get(index - errors.size());
        }

        return p.getObjects();
    }
}

/**
 *
 * @return The table if the gui is shown.
 */
public final JTable getOutputTable() {
    if (showResultsInGUI) {
        return gui.getTable();
    } else {
        return null;
    }
}

/**
 *
 * @return The component from the gui.
 */
public final JComponent getComponent() {
    return gui.getComponent();
}

/**
 * This method sets the warning and error messages.
 */
private void setTextInLabel() {
    gui.setLabelText(getSummary());
}
```

```java
/**
 * This method builds the warning and error messages.
 * @return The warning and error messages.
 */
public final String getSummary() {
    String result = "";

    if (errors.size() == 1) {
        result = errors.size() + " Error, ";
    } else {
        result = errors.size() + " Errors, ";
    }
    if (warnings.size() == 1) {
        result += warnings.size() + " Warning";
    } else {
        result += warnings.size() + " Warnings";
    }

    return result;
}

/**
 *
 * @return Index of the appropriate tab in KIELFrame.
 */
public final int getTabbedIndex() {
    return tabbedIndex;
}

/**
 * Setter for the rules.
 * @param ti The index of the tab.
 */
public final void setTabbedIndex(final int ti) {
    this.tabbedIndex = ti;
}

/**
 * Getter for the rules.
 * @return The rules.
 */
public final ArrayList getRules() {
    return rules;
}

/**
 * Enables or disables a rule.
 * @param ruleName The name of the rule.
```

```
     * @param isEnabled .
     */
    public final void setEnabled(final String ruleName,
            final boolean isEnabled) {
        BaseCheck item;
        for (int i = 0; i < rules.size(); i++) {
            item = (BaseCheck) rules.get(i);
            if (item.getRuleName().equals(ruleName)) {
                CheckingProperties.setRuleStatus(this.rulePackage.substring(
                        this.rulePackage.lastIndexOf(File.separatorChar) + 1,
                        this.rulePackage.lastIndexOf('.'))
                        + "." + ruleName, isEnabled);
                item.setEnabled(isEnabled);
            }
        }
    }

    /**
     * Clears the errors and warnings in the table.
     */
    public final void refresh() {
        errors.clear();
        warnings.clear();
        this.setTextInLabel();
        this.fireTableDataChanged();
    }

    /**
     * Clears all user input.
     */
    public final void clearUserInput() {
        JTable table = getOutputTable();
        table.clearSelection();
    }
}
```

A.2.3 StateChartDepthFirstAdapter

```
//$Id: StateChartDepthFirstAdapter.java,v 1.5 2006/03/27 15:52:08 gsc Exp $
package kiel.util;

import java.util.ArrayList;

import kiel.dataStructure.CompositeState;
import kiel.dataStructure.Node;
import kiel.dataStructure.Region;
/**
 * <p>Description: State chart visitor using the depth first search to
 * traverse the datastructure</p>
 * <p>Copyright: (c) 2006</p>
 * <p>Company: Uni Kiel</p>.
 * @author <a href="mailto:kbe@informatik.uni-kiel.de">Ken Bell</a>
 * @author <a href="mailto:gsc@informatik.uni-kiel.de">Gunnar Schaefer</a>
 * @version $Revision: 1.5 $ last modified $Date: 2006/03/27 15:52:08 $
 */
public class StateChartDepthFirstAdapter {

    /**
     *
     * @param intf The visitor whose dispatch method is called when a node is
     * found
     * @param root The start node of the search
     * @param doVisitRegions Denoting wheter to dispatch regions to the visitor
     * or not
     * @return Can be modified for later usage.
     */
    public final boolean traverse(final IStateChartVisitor intf,
            final Node root, final boolean doVisitRegions) {

        ArrayList stack = new ArrayList();
        Node n;

        stack.add(root);

        while (stack.size() > 0) {

            n = (Node) stack.remove(0);

            /**
             * The node is dispatched if either the parameter doVisitResions is
             * true or the node is not a region.
             */
            if (doVisitRegions || (!(n instanceof Region))) {
                intf.dispatch(n);
            }

            if (n instanceof CompositeState) {
                stack.addAll(((CompositeState) n).getSubnodes());
            }
        }
        return true;
    }
}
```

A.2.4 CheckerOutputGUI

```java
//$Id: CheckerOutputGUI.java,v 1.1 2006/04/10 10:22:22 gsc Exp $
package kiel.checking;

import java.awt.BorderLayout;

import javax.swing.JComponent;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.ListSelectionModel;

/**
 * <p>Description: This class displays the problems found by a robustness
 * checker.</p>
 *
 * <p>Copyright: (c) 2006</p>
 * <p>Company: Uni Kiel</p>
 * @author <a href="mailto:kbe@informatik.uni-kiel.de">Ken Bell</a>
 * @author <a href="mailto:gsc@informatik.uni-kiel.de">Gunnar Schaefer</a>
 *
 * @version $Revision: 1.1 $ last modified $Date: 2006/04/10 10:22:22 $
 */
public class CheckerOutputGUI {


    /**
     * The table that displays the data.
     */
    private JTable table;


    /**
     * The panel that holds the table and the label.
     */
    private JPanel panel;

    /**
     * The label displys the number of errors and warnings.
     */
    private JLabel label;

    /**
     * This method is used to create and setup the table
     * which is displayed in the frame.
     * @param c The StateChartCheckerBase.
     */
    private void createTable(final StateChartCheckerBase c) {

        table = new JTable(c);
        /* Set the selection mode to full row selection*/
        table.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
        /* Set the cell renderer. */
        table.getColumnModel().getColumn(0).setCellRenderer(
                new MyCellRenderer(c));
    }

    /**
     * The constructor of the frame.
     * It sets the title of the frame, creates and adds the table to itself.
     * @param c The StateChartCheckerBase.
     */
    public CheckerOutputGUI(final StateChartCheckerBase c) {

        createTable(c);
        label = new JLabel("");
        panel = new JPanel();
        panel.setLayout(new BorderLayout());
        panel.add(label, BorderLayout.NORTH);
        panel.add(new JScrollPane(table), BorderLayout.CENTER);
    }

    /**
     * Get the table.
     * @return The instance of the table.
     */
    public final JTable getTable() {
        return table;
    }

    /**
     * @return Return the Scrollpane
     */
    public final JComponent getComponent() {
        return panel;
    }

    /**
     *
     * @param t Text to be displayed in the label above the table.
     */
    public final void setLabelText(final String t) {
        label.setText(t);
    }
}
```

A.2.5 CheckingProblem

```
//$Id: CheckingProblem.java,v 1.1 2006/04/10 10:22:22 gsc Exp $
package kiel.checking;

import java.util.ArrayList;

import kiel.dataStructure.GraphicalObject;


/**
 *
 * <p>Description: This class holds the informations about problems that
 * occured when parsing the datastructure. Instances will be stored in an
 * instance of @see StateChartCheckerBase.</p>
 * <p>Copyright: (c) 2006</p>
 * <p>Company: Uni Kiel</p>.
 * @author <a href="mailto:kbe@informatik.uni-kiel.de">Ken Bell</a>
 * @author <a href="mailto:gsc@informatik.uni-kiel.de">Gunnar Schaefer</a>
 * @version $Revision: 1.1 $ last modified $Date: 2006/04/10 10:22:22 $
 *
 */
public class CheckingProblem {

    /**
     * Holds a short description of the problem.
     */
    private String problem;


    /*
     * Holds the object which is the source of the problem.
     */
    //private GraphicalObject object;

    /**
     * Holds the object that are the source of the problem.
     */
    private ArrayList objects;

    /**
     * The constructor of the robustness problem.
     * @param o The source of the problem.
     * @param p A problem description.
     */
    public CheckingProblem(final GraphicalObject o, final String p) {
        this.objects = new ArrayList();
        this.problem = p;
        this.objects.add(o);
    }


    /**
     *
     * @param o An ArrayList containing the source of the problem.
     * @param p A description of the problem.
     */
    public CheckingProblem(final ArrayList o, final String p) {
        this.problem = p;
        this.objects = new ArrayList(o);
    }


    /**
     *
     * @return Returns the source of the problem.
     */
    public final GraphicalObject getFirstObject() {
        return (GraphicalObject) objects.get(0);
    }

    /**
     * Set the source of a problem manually.
     * @param o The source of the problem.
     */
    public final void setFirstObject(final GraphicalObject o) {
        this.objects.set(0, o);
    }


    /**
     *
     * @return A short description of the problem.
     */
    public final String getProblem() {
        return problem;
    }


    /**
     *
     * @param p Set the description of the problem manually.
     */
    public final void setProblem(final String p) {
        this.problem = p;
    }
    /**
     *
     * @return The sources of the problem in an ArrayList.
     */
    public final ArrayList getObjects() {
        return objects;
    }


    /**
     *
     * @param o An ArrayList containing graphical objects.
     */
    public final void setObjects(final ArrayList o) {
        this.objects = o;
    }

}
```

A.2.6 MyCellRenderer

```
//$Id: MyCellRenderer.java,v 1.1 2006/04/10 10:22:22 gsc Exp $
package kiel.checking;

import java.awt.Color;
import java.awt.Component;

import javax.swing.JTable;
import javax.swing.JTextArea;
import javax.swing.table.TableCellRenderer;

/**
 * Description: The cell renderer for the table of errors and warnings.
 * @author <a href="mailto:kbe@informatik.uni-kiel.de">Ken Bell</a>
 * @author <a href="mailto:gsc@informatik.uni-kiel.de">Gunnar Schaefer</a>
 * @version
 */
public class MyCellRenderer extends JTextArea implements TableCellRenderer {

    /**
     * The StateChartCheckerBase.
     */
    private StateChartCheckerBase checker;

    /**
     * Constructor of a CellRenderer.
     * @param c The StateChartCheckerBase.
     */
    public MyCellRenderer(final StateChartCheckerBase c) {
        this.checker = c;
        setLineWrap(true);
        setWrapStyleWord(true);
    }

    /**
     * @param aTable The table of which the given cell has to be rendered.
     * @param value The value that has to be displayed.
     * @param isSelected Denoting if the cell is selected.
     * @param hasFocus Denoting if the cell has the focus.
     * @param row The rowindex of the cell.
     * @param column The columnindex of the cell.
     * @return A instance of the renderer.
     */
    public final Component getTableCellRendererComponent(final JTable aTable,
            final Object value, final boolean isSelected,
            final boolean hasFocus, final int row, final int column) {
        this.setText(value.toString());
        setSize(aTable.getColumnModel().getColumn(column).getWidth(),
                getPreferredSize().height);
        if (row < checker.getErrors().size()) {
            this.setForeground(CheckingProperties.getErrorColor());
        } else {
            this.setForeground(CheckingProperties.getWarningColor());
        }
        if (aTable.getRowHeight(row) != getPreferredSize().height) {
                aTable.setRowHeight(row, getPreferredSize().height);
        }
        if (isSelected) {
            this.setBackground(aTable.getSelectionBackground());
            this.setForeground(Color.BLACK);
        } else {
            this.setBackground(Color.WHITE);
        }
        return this;
    }
}
```

A.2.7 CheckingProperties

```
//$Id: CheckingProperties.java,v 1.1 2006/04/10 10:22:22 gsc Exp $
package kiel.checking;

import java.awt.Color;

import kiel.util.preferences.Preferences;
import kiel.util.LogFile;

/**
 * <p>Description: Used to load and store user defineable properties.</p>
 *
 * <p>Copyright: (c) 2006</p>
 * <p>Company: Uni Kiel</p>.
 * @author <a href="mailto:gsc@informatik.uni-kiel.de">Gunnar Schaefer</a>
 * @author <a href="mailto:kbe@informatik.uni-kiel.de">Ken Bell</a>
 * @version $Revision: 1.1 $ last modified $Date: 2006/04/10 10:22:22 $
 */
public final class CheckingProperties {

    /**
     * These are the internal properties.
     */
    private static Preferences prefs;

    /**
     * This the key for the resouce file containing the default values.
     */
    private static final String DEFAULTRESOURCE = "checking.properties";

    /**
     * The robustness log.
     */
    private static LogFile checkingLog = new LogFile("Checking", 0);

    static {
        prefs = Preferences.getInstance("checking",
                CheckingProperties.class.getResource(DEFAULTRESOURCE),
                checkingLog);
        checkingLog.setLogLevel(getRobustnessLogLevel());
    }

    /**
     * @return The robustness log.
     */
    public static LogFile getRobustnessLog() {
        return checkingLog;
    }

    /**
     * The semantic checker log.
     */
    private static LogFile corrLog = new LogFile("Correctness",
            getCorrLogLevel());

    /**
     * @return The semantic checker log.
     */
    public static LogFile getCorrLog() {
        return corrLog;
    }

    /**
     * The semantic checker log.
     */
    private static LogFile semLog = new LogFile("Semantic Robustness",
            getSemLogLevel());

    /**
     * @return The semantic checker log.
     */
    public static LogFile getSemLog() {
        return semLog;
    }

    /**
     * The syntactic checker log.
     */
    private static LogFile synLog = new LogFile("Syntactic Robustness",
            getSynLogLevel());

    /**
     * @return The syntactic checker log.
     */
    public static LogFile getSynLog() {
        return synLog;
    }

    /**
     * Static class, do not instantiate.
     */
    private CheckingProperties() {

    }

    /**
```

```
     * This is the key for property key delimiter.
     */
    private static final String DELIMITER = ".";

    /**
     *
     */
    private static final String CHECKING = "checking";

    /**
     *
     */
    private static final String ERRORCOLOR = "errorColor";

    /**
     *
     */
    private static final String WARNINGSCOLOR = "warningColor";

    /**
     *
     */
    private static final String RULESENABLED = "ruleEnabled";

    /**
     *
     */
    private static final String CVCLMODE = "cvclMode";

    /**
     *
     */
    private static final String CHECKINGLOGLEVEL = "CheckingLogLevel";

    /**
     *
     */
    private static final String CORRECTNESSLOGLEVEL = "CorrLogLevel";

   /**
   *
   */
   private static final String SEMANTICLOGLEVEL = "SemLogLevel";

   /**
    *
   */
   private static final String SYNTACTICLOGLEVEL = "SynLogLevel";

   /**
    *
    */
   private static final String SHOWNODEID = "showNodeID";

   /**
    *
    */
   private static final String SHOWNODENAME = "showNodeName";

   /**
    *
    */
   private static final String SHOWCLASSIFICATION = "showClassification";

   /**
    *
    */
   private static final String SHOWCOUNTERS = "showCounters";

   /**
    *
    */
   private static final String SHOWTRANSITIONS = "showTransitions";

   /**
    *
    */
   private static final String SHOWPRIORITY = "showPriority";

   /**
    * Reloads the properties file.
    */
   protected static void reload() {
       prefs.reload();
   }

   /**
    *
    * @return The color of errors.
    */
   public static Color getErrorColor() {
       return prefs.getRGBColor(CHECKING + DELIMITER + ERRORCOLOR);
   }

   /**
    *
    * @return The color of warnings.
    */
    public static Color getWarningColor() {
        return prefs.getRGBColor(CHECKING + DELIMITER + WARNINGSCOLOR);
    }

    /**
     * Reads the rule status from the properties.
     * @param ruleName The name of the rule.
     * @return The status of the rule.
     */
    public static boolean getRuleStatus(final String ruleName) {
        return prefs.getBooleanAndAddIfMissing(CHECKING + DELIMITER
                + RULESENABLED + DELIMITER + ruleName, true);
    }

    /**
     * Sets the rule status in the properties.
     * @param ruleName The name of the rule.
     * @param value The value.
     */
    public static void setRuleStatus(final String ruleName,
            final boolean value) {
        prefs.setBoolean(ruleName, value);
    }

    /**
     * @return .
     */
    public static int getRobustnessLogLevel() {
        return prefs.getInt(CHECKING + DELIMITER + CHECKINGLOGLEVEL);
    }

    /**
     * @return .
     */
    public static int getCorrLogLevel() {
        return prefs.getInt(CHECKING + DELIMITER + CORRECTNESSLOGLEVEL);
    }

    /**
     * @return .
     */
    public static int getSemLogLevel() {
        return prefs.getInt(CHECKING + DELIMITER + SEMANTICLOGLEVEL);
    }

    /**
     * @return .
     */
    public static int getSynLogLevel() {
        return prefs.getInt(CHECKING + DELIMITER + SYNTACTICLOGLEVEL);
    }

    /**
     * @return Whether to use CVCL lib or bin.
     */
    public static boolean isCvclModeBin() {
        return prefs.getString(CHECKING + DELIMITER + CVCLMODE)
                .equalsIgnoreCase("bin");
    }

    /**
     * @return .
     */
    public static boolean showNodeID() {
        return prefs.getString(CHECKING + DELIMITER + SHOWNODEID)
                .equals("true");
    }

    /**
     * @return .
     */
    public static boolean showNodeName() {
        return prefs.getString(CHECKING + DELIMITER + SHOWNODENAME)
                .equals("true");
    }

    /**
     * @return .
     */
    public static boolean showClassification() {
        return prefs.getString(CHECKING + DELIMITER + SHOWCLASSIFICATION)
                .equals("true");
    }

    /**
     * @return .
     */
    public static boolean showCounters() {
        return prefs.getString(CHECKING + DELIMITER + SHOWCOUNTERS)
                .equals("true");
    }

    /**
     * @return .
     */
    public static boolean showTransitions() {
        return prefs.getString(CHECKING + DELIMITER + SHOWTRANSITIONS)
                .equals("true");
    }

    /**
     * @return .
     */
    public static boolean showPriority() {
        return prefs.getString(CHECKING + DELIMITER + SHOWPRIORITY)
                .equals("true");
    }
}
```

A.2.8 Checking Properties

```
################################################
#
# do not edit this file !
# unless you know exactly what you are doing.
#
################################################

###################
# color settings
# Specify the color in the format "R,G,B"
# [int, int, int]
checking.errorColor=255,0,0
checking.warningColor=178,140,0

###################
# whether to use JCVCL library or CVCL binary
# JCVCL library not possible on Windows,
# if set to [lib] on Windows, KIEL will still use [bin]
# [lib] or [bin]
checking.cvclMode=bin

###################
# the log levels
# [ERROR], [STRUCTURE], [INFO], [DETAIL], [DEBUG], or [ALL]
checking.CheckingLogLevel=10
checking.CorrLogLevel=10
checking.SemLogLevel=10
checking.SynLogLevel=10

###################
# custom settings for error and warning messages
# [true] or [false]
checking.showNodeID=true
checking.showNodeName=true
checking.showClassification=true
checking.showCounters=true
checking.showTransitions=false
checking.showPriority=true

###################
# statuses of robustness rules
# rule is checked: [true] or [false]
```

A.3 Semantic Robustness Checks

A.3.1 DwellingCheck

```java
//$Id: DwellingCheck.java,v 1.2 2006/04/12 15:31:15 gsc Exp $
package kiel.checking.semanticRobustnessChecks;

import java.text.MessageFormat;
import java.util.ArrayList;
import java.util.Iterator;

import kiel.checking.BaseCheck;
import kiel.checking.CheckingProblem;
import kiel.checking.CheckingProperties;
import kiel.checking.StateChartCheckerBase;
import kiel.dataStructure.ANDState;
import kiel.dataStructure.CompoundLabel;
import kiel.dataStructure.GraphicalObject;
import kiel.dataStructure.Node;
import kiel.dataStructure.ORState;
import kiel.dataStructure.SimpleState;
import kiel.dataStructure.State;
import kiel.dataStructure.Suspension;
import kiel.dataStructure.Transition;
import kiel.util.LogFile;

/**
 * <p>Description: Ensures dwelling in all states, i.e., no outgoing immediate
 * transition and an incomming trantition may have overlapping lables,
 * dynamically loaded by @see StateChartCheckerBase.</p>
 *
 * <p>Copyright: (c) 2006</p>
 * <p>Company: Uni Kiel</p>
 * @author <a href="mailto:gsc@informatik.uni-kiel.de">Gunnar Schaefer</a>
 * @version $Revision: 1.2 $ last modified $Date: 2006/04/12 15:31:15 $
 */
public final class DwellingCheck extends BaseCheck {

    /**
     * The error string.
     */
    private final String message = "{0}{1}{2}{3}{4}"
            + "An incomming and an \"immediate\" outgoing transition "
            + "have overlapping labels.";

    /**
     * Number of arguments used of string formatting of error and warning
     * messages.
     */
    private final int numOfArgs = 5;

    /**
     * The validity checker used to check for overlapping transition predicates.
     */
    private MyValidityCheckerBase myVC;

    /**
     * Contains a state and two transitions.
     */
    private ArrayList problemLocation = new ArrayList();

    /**
     * All transitions comming into a state.
     */
    private ArrayList incommingTransitions = new ArrayList();

    /**
     * All transitions directly and indirectly outgoing from a state.
     */
    private ArrayList outgoingTransitions = new ArrayList();

    /**
     * @see BaseCheck
     * @param cvclProc @see BaseCheck
     */
    public void setCvclProcess(final Process cvclProc) {
        myVC = new MyValidityCheckerBin(cvclProc);
    }

    /**
     * @see BaseCheck
     * @param o @see BaseCheck
     * @return @see BaseCheck
     */
    public boolean isValidTarget(final GraphicalObject o) {

        Class c = o.getClass();
        return c.equals(ANDState.class)
            || c.equals(ORState.class)
            || c.equals(SimpleState.class);
    }

    /**
     * @see BaseCheck
     * @param checker @see BaseCheck
     * @param o @see BaseCheck
     */
    public void apply(final StateChartCheckerBase checker,
            final GraphicalObject o) {

        // is only null on first pass in lib mode
        if (myVC == null) {
            myVC = new MyValidityCheckerLib();
        }

        State s = (State) o;
        Transition incomming;
        Transition outgoing;
        incommingTransitions.clear();
        outgoingTransitions.clear();

        CheckingProperties.getSemLog().log(LogFile.DETAIL,
                "DwellingCheck, State: " + s);

        // remove suspensions from incommingTransitions
        incommingTransitions.addAll(s.getIncomingTransitions());
        Iterator incommingIter = incommingTransitions.iterator();
        while (incommingIter.hasNext()) {
            incomming = (Transition) incommingIter.next();
            if (incomming instanceof Suspension) {
                incommingIter.remove();
            } else if (!(incomming.getLabel() instanceof CompoundLabel)) {
                incommingIter.remove();
                CheckingProperties.getSemLog().log(LogFile.DEBUG,
                        "WARNING: Transition " + incomming
                            + " ignored, StringLabel");
            }
        }

//        // get ALL outgoing transitions
//        while (s != null) {
            outgoingTransitions.addAll(s.getOutgoingTransitions());
//            s = s.getParent();
//        }

        // remove "non immediate" transitions from outgoingTransitions
        Iterator outgoingIter = outgoingTransitions.iterator();
        while (outgoingIter.hasNext()) {
            outgoing = (Transition) outgoingIter.next();
            if (outgoing.getLabel() instanceof CompoundLabel) {
                if (!((CompoundLabel) outgoing.getLabel()).
                        getTrigger().isImmediate()) {
                    outgoingIter.remove();
                }
            } else {
                CheckingProperties.getSemLog().log(LogFile.DEBUG,
                        "WARNING: Transition " + outgoing
                            + " ignored, StringLabel");
                outgoingIter.remove();
            }
        }

        incommingIter = incommingTransitions.iterator();
        while (incommingIter.hasNext()) {
            incomming = (Transition) incommingIter.next();
            outgoingIter = outgoingTransitions.iterator();
            while (outgoingIter.hasNext()) {
                outgoing = (Transition) outgoingIter.next();
                if (incomming != outgoing) { // avoid identical transitions
                   if (myVC.areOverlapping((CompoundLabel) incomming.getLabel(),
                            (CompoundLabel) outgoing.getLabel())) {
                        problemLocation.clear();
                        problemLocation.add(o);
                        problemLocation.add(incomming);
                        problemLocation.add(outgoing);
                        generateProblem(checker, problemLocation);
                    }
                }
            }
        }
    }

    /**
     * This method is used internally by a check to generate the according
     * instance of a RobustnessProblem.
     * @param checker The checker where the problem will be added to.
     * @param objects The sources of the problem.
     */
    private void generateProblem(final StateChartCheckerBase checker,
            final ArrayList objects) {

        Node n = (Node) objects.get(0);
        String s = n.getClass().getName();
        Object[] args = new Object[numOfArgs];

        if (CheckingProperties.showNodeName()) {
            args[0] = s.substring(s.lastIndexOf('.') + 1);
            if (n instanceof State) {
                if (n.getName().equals("")) {
                    args[1] = "";
                } else {
                    args[1] = " \"" + n.getName() + "\"";
                }
            } else {
                args[1] = " Pseudo State";
            }
        } else {
            args[0] = "";
            args[1] = "";
        }

        if (CheckingProperties.showNodeID()) {
            args[2 + 1] = "(ID: " + n.getID() + ")";
        } else {
            args[2 + 1] = "";
        }

        if (CheckingProperties.showNodeName()
                && CheckingProperties.showNodeID()) {
            args[2] = " ";
            args[2 + 2] = ": ";
        } else if (CheckingProperties.showNodeName()
                || CheckingProperties.showNodeID()) {
            args[2] = "";
            args[2 + 2] = ": ";

        } else {
            args[2] = "";
            args[2 + 2] = "";
        }

        checker.getWarnings().add(new CheckingProblem(
                objects, MessageFormat.format(message, args)));
    }
}
```

A.3.2 TransitionOverlapCheck

```
//$Id: TransitionOverlapCheck.java,v 1.2 2006/04/24 12:20:41 gsc Exp $
package kiel.checking.semanticRobustnessChecks;

import java.security.InvalidParameterException;
import java.text.MessageFormat;
import java.util.ArrayList;
import java.util.Iterator;

import javax.swing.JOptionPane;
import kiel.checking.BaseCheck;
import kiel.checking.CheckingProblem;
import kiel.checking.CheckingProperties;
import kiel.checking.StateChartCheckerBase;
import kiel.dataStructure.Choice;
import kiel.dataStructure.CompoundLabel;
import kiel.dataStructure.ForkConnector;
import kiel.dataStructure.GraphicalObject;
import kiel.dataStructure.Junction;
import kiel.dataStructure.Node;
import kiel.dataStructure.NormalTermination;
import kiel.dataStructure.State;
import kiel.dataStructure.Transition;
import kiel.util.LogFile;

/**
 * <p>Description: Checks for overlapping transitions, i.e., one transition
 * being shadowed by another, dynamically loaded
 * by @see StateChartCheckerBase.</p>
 *
 * <p>Copyright: (c) 2006</p>
 * <p>Company: Uni Kiel</p>
 * @author <a href="mailto:gsc@informatik.uni-kiel.de">Gunnar Schaefer</a>
 * @version $Revision: 1.2 $ last modified $Date: 2006/04/24 12:20:41 $
 */
public final class TransitionOverlapCheck extends BaseCheck {

    /**
     * The error string.
     */
    private final String message = "{0}{1}{2}{3}{4}"
        + "Two outgoing transitions have overlapping labels.";

    /**
     * Number of arguments used of string formatting of error and warning
     * messages.
     */
    private final int numOfArgs = 5;

    /**
     * The validity checker used to check for overlapping transition predicates.
     */
    private MyValidityCheckerBase myVC;

    /**
     * Contains a state and two transitions.
     */
    private ArrayList problemLocation = new ArrayList();
```

```
/**
 * All transitions directly and indirectly outgoing from a state.
 */
private ArrayList outgoingTransitions = new ArrayList();

/**
 * @see BaseCheck
 * @param cvclProc @see BaseCheck
 */
public void setCvclProcess(final Process cvclProc) {
    myVC = new MyValidityCheckerBin(cvclProc);
}

/**
 * @see BaseCheck
 * @param o @see BaseCheck
 * @return @see BaseCheck
 */
public boolean isValidTarget(final GraphicalObject o) {

    // Choice and Junction get special handling below (junctionHandler)
    return (o instanceof Node) && !(o instanceof ForkConnector);
}

/**
 * @see BaseCheck
 * @param checker @see BaseCheck
 * @param o @see BaseCheck
 */
public void apply(final StateChartCheckerBase checker,
        final GraphicalObject o) {

    // is only null on first pass in lib mode
    if (myVC == null) {
        myVC = new MyValidityCheckerLib();
    }

    Transition t1;
    Transition t2;
    Node n = (Node) o;
    outgoingTransitions.clear();

    CheckingProperties.getSemLog().log(LogFile.DETAIL,
            "TransitionOverlapCheck, State: " + n);

    // for States, get ALL outgoing transitions
    if (n instanceof State) {
        while (n != null) {
            outgoingTransitions.addAll(n.getOutgoingTransitions());
            n = n.getParent();
        }
    } else {
        outgoingTransitions.addAll(n.getOutgoingTransitions());
    }

    // remove transitions that do not have a CompoundLabel
    Iterator outgoingIter = outgoingTransitions.iterator();
    while (outgoingIter.hasNext()) {
        t1 = (Transition) outgoingIter.next();
        if (!(t1.getLabel() instanceof CompoundLabel)) {
            outgoingIter.remove();
            CheckingProperties.getSemLog().log(LogFile.DEBUG,
                    "WARNING: Transition " + t1 + " ignored, StringLabel");
        }
    }

    // iterate pairwise over all outgoing transitions
    int size = outgoingTransitions.size();
    int j;
    for (int i = 0; i < size - 1; i++) {
        t1 = (Transition) outgoingTransitions.get(i);
        for (j = i + 1; j < size; j++) {
            t2 = (Transition) outgoingTransitions.get(j);
            // make sure overlap check actually makes sense for t1 and t2
            if (makesSense(t1, t2, (Node) o)) {
                try {
                    if ((t1.getTarget() instanceof Choice)
                            || (t1.getTarget() instanceof Junction)
                            || (t2.getTarget() instanceof Choice)
                            || (t2.getTarget() instanceof Junction)) {
                        junctionHandler(t1, t2, (Node) o, checker);
                    } else if (myVC.areOverlapping((CompoundLabel)
                            t1.getLabel(), (CompoundLabel) t2.getLabel())) {
                        problemLocation.clear();
                        problemLocation.add(o);
                        problemLocation.add(t1);
                        problemLocation.add(t2);
                        generateProblem(checker, problemLocation);
                    }
                } catch (InvalidParameterException e) {
                    JOptionPane.showMessageDialog(null, e.getMessage(),
                            "Semantic Robustness Checker",
                            JOptionPane.ERROR_MESSAGE);
                }
            }
        }
    }
}
```

```java
    /**
     * Determines if it makes sense to call overlap check.
     * @param t1 The first Transition.
     * @param t2 The second Transiton.
     * @param n The current Node.
     * @return True is overlap check should be called.
     */
    private boolean makesSense(final Transition t1, final Transition t2,
            final Node n) {

        // at least one transiton must be directly outgoing from o
        boolean direct = (t1.getSource() == n || t2.getSource() == n);

        // NormalTerminations are not "in competition" with other transitions
        boolean normalTerm = (t1 instanceof NormalTermination)
                || (t2 instanceof NormalTermination);

        /*
         * label of lower priority transition may be empty or "tick"
         * (default transition), but not the labels of both transitions
         */
        boolean isDefaultTrans = false;
        if (t1.getSource() == t2.getSource()) {
            if (t1.getPriority().getValue() > t2.getPriority().getValue()) {
                isDefaultTrans = ((t1.getLabel().toString().equals("")
                        || t1.getLabel().toString().equals("tick"))
                        && !(t2.getLabel().toString().equals("")
                        || t2.getLabel().toString().equals("tick")));
            } else if (t1.getPriority().getValue()
                    < t2.getPriority().getValue()) {
                isDefaultTrans = ((t2.getLabel().toString().equals("")
                        || t2.getLabel().toString().equals("tick"))
                        && !(t1.getLabel().toString().equals("")
                        || t1.getLabel().toString().equals("tick")));
            }
        }
        return direct && !isDefaultTrans && !normalTerm;
    }

    /**
     * Takes care of the special case when at least one of the transitions'
     * target is a Choice of Junction.
     * @param t1 The first transition.
     * @param t2 The second transition.
     * @param n The current Node.
     * @param checker The current StateChartCheckerBase.
     */
    private void junctionHandler(final Transition t1, final Transition t2,
            final Node n, final StateChartCheckerBase checker) {

        ArrayList transitions1 = new ArrayList();
        ArrayList transitions2 = new ArrayList();
        ArrayList labels1 = new ArrayList();
        ArrayList labels2 = new ArrayList();
        ArrayList stack = new ArrayList();
        ArrayList compTrans1 = new ArrayList();
        ArrayList compTrans2 = new ArrayList();
        Iterator iter;
        Transition t;
        CompoundLabel l;
        Transition stackTrans;

        // for t1, build all possible transition chains
        stack.add(t1);
        while (stack.size() > 0) {
            stackTrans = (Transition) stack.remove(0);
            if (transitions1.size() > 0) {
                if (stackTrans.getSource()
                        == ((Transition) transitions1.get(0)).getSource()) {
                    transitions1.remove(0);
                }
            }
            transitions1.add(0, stackTrans);
            if (stackTrans.getTarget() instanceof Junction
                    || stackTrans.getTarget() instanceof Choice) {
                stack.addAll(0,
                        stackTrans.getTarget().getOutgoingTransitions());
            } else {
                compTrans1.add(new ArrayList(transitions1));
                transitions1.remove(0);
            }
        }
        // for t2, build all possible transition chains
        stack.add(t2);
        while (stack.size() > 0) {
            stackTrans = (Transition) stack.remove(0);
            if (transitions2.size() > 0) {
                if (stackTrans.getSource()
                        == ((Transition) transitions2.get(0)).getSource()) {
                    transitions2.remove(0);
                }
            }
            transitions2.add(0, stackTrans);
            if (stackTrans.getTarget() instanceof Junction
                    || stackTrans.getTarget() instanceof Choice) {
                stack.addAll(0,
                        stackTrans.getTarget().getOutgoingTransitions());
        } else {
            compTrans2.add(new ArrayList(transitions2));
            transitions2.remove(0);
        }
    }

    // iterate pairwise over both transition lists
    for (int i = 0; i < compTrans1.size(); i++) {
        transitions1 = (ArrayList) compTrans1.get(i);
        for (int j = 0; j < compTrans2.size(); j++) {
            transitions2 = (ArrayList) compTrans2.get(j);

            problemLocation.clear();
            problemLocation.add(n);

            labels1.clear();
            iter = transitions1.iterator();
            while (iter.hasNext()) {
                t = (Transition) iter.next();
                l = (CompoundLabel) t.getLabel();
                labels1.add(l);
                problemLocation.add(t.getSource());
                problemLocation.add(t);
            }

            labels2.clear();
            iter = transitions2.iterator();
            while (iter.hasNext()) {
                t = (Transition) iter.next();
                l = (CompoundLabel) t.getLabel();
                labels2.add(l);
                problemLocation.add(t.getSource());
                problemLocation.add(t);
            }
            if (myVC.areOverlappingMulti(labels1, labels2)) {
                generateProblem(checker, problemLocation);
            }
        }
    }
}

/**
 * This method is used internally by a check to generate the according
 * instance of a RobustnessProblem.
 * @param checker The checker where the problem will be added to.
 * @param objects The sources of the problem.
 */
private void generateProblem(final StateChartCheckerBase checker,
        final ArrayList objects) {

    Node n = (Node) objects.get(0);
    String s = n.getClass().getName();
    Object[] args = new Object[numOfArgs];

    if (CheckingProperties.showNodeName()) {
        args[0] = s.substring(s.lastIndexOf('.') + 1);
        if (n instanceof State) {
            if (n.getName().equals("")) {
                args[1] = "";
            } else {
                args[1] = " \"" + n.getName() + "\"";
            }
        } else {
            args[1] = " Pseudo State";
        }
    } else {
        args[0] = "";
        args[1] = "";
    }

    if (CheckingProperties.showNodeID()) {
        args[2 + 1] = "(ID: " + n.getID() + ")";
    } else {
        args[2 + 1] = "";
    }

    if (CheckingProperties.showNodeName()
            && CheckingProperties.showNodeID()) {
        args[2] = " ";
        args[2 + 2] = ": ";
    } else if (CheckingProperties.showNodeName()
            || CheckingProperties.showNodeID()) {
        args[2] = "";
        args[2 + 2] = ": ";

    } else {
        args[2] = "";
        args[2 + 2] = "";
    }

    checker.getWarnings().add(new CheckingProblem(
            objects, MessageFormat.format(message, args)));
}
}
```

A.3.3 RaceConditionsCheck

```java
//$Id: RaceConditionsCheck.java,v 1.6 2006/04/23 14:37:15 gsc Exp $
package kiel.checking.semanticRobustnessChecks;

import java.text.MessageFormat;
import java.util.ArrayList;
import java.util.Iterator;

import kiel.checking.BaseCheck;
import kiel.checking.CheckingProblem;
import kiel.checking.CheckingProperties;
import kiel.checking.StateChartCheckerBase;
import kiel.dataStructure.CompositeState;
import kiel.dataStructure.CompoundLabel;
import kiel.dataStructure.GraphicalObject;
import kiel.dataStructure.Node;
import kiel.dataStructure.Region;
import kiel.dataStructure.Transition;
import kiel.dataStructure.action.Action;
import kiel.dataStructure.action.GenerateValuedEvent;
import kiel.dataStructure.action.IntegerAssignment;
import kiel.dataStructure.boolexp.BooleanAnd;
import kiel.dataStructure.boolexp.BooleanBrackets;
import kiel.dataStructure.boolexp.BooleanComparator;
import kiel.dataStructure.boolexp.BooleanExpression;
import kiel.dataStructure.boolexp.BooleanNot;
import kiel.dataStructure.boolexp.BooleanOr;
import kiel.dataStructure.boolexp.BooleanVariable;
import kiel.dataStructure.boolexp.Pre;
import kiel.dataStructure.eventexp.CombineSignal;
import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.dataStructure.eventexp.Signal;
import kiel.dataStructure.intexp.IntegerExpression;
import kiel.dataStructure.intexp.IntegerVariable;
import kiel.util.LogFile;

/**
 * <p>Description: Checks for uniqueness of transition pririties, dynamically
 * loaded by @see StateChartCheckerBase.</p>
 *
 * <p>Copyright: (c) 2006</p>
 * <p>Company: Uni Kiel</p>
 * @author <a href="mailto:gsc@informatik.uni-kiel.de">Gunnar Schaefer</a>
 * @version $Revision: 1.6 $ last modified $Date: 2006/04/23 14:37:15 $
 */
public final class RaceConditionCheck extends BaseCheck {

    /**
     * The write/write error string.
     */
    private final String message1 = "A Write/Write Race Condition might occur "
        + "between two transiton actions {0} and {1}{2}.";

    /**
     * The write/read error string.
     */
    private final String message2 = "A Write/Read Race Condition might occur "
        + "between transiton action {0} and transition condition {1}{2}.";
    /**
     * Number of arguments used of string formatting of error and warning
     * messages.
     */
    private final int numOfArgs = 3;

    /**
     * Contains a state and two transitions.
     */
    private ArrayList problemLocation = new ArrayList();

    /**
     * All transitions directly and indirectly outgoing from a state.
     */
    private ArrayList outgoingTransitions = new ArrayList();

    /**
     * All transitions directly and indirectly outgoing from a state.
     */
    private ArrayList parallelTransitions;

    /**
     * All valued signals and variables read by a transition condition.
     */
    private ArrayList readVars = new ArrayList();

    /**
     * The parent of the source of the race condition.
     */
    private BooleanComparator parent = null;

    /**
     * @see BaseCheck
     * @param o @see BaseCheck
     * @return @see BaseCheck
     */
    public boolean isValidTarget(final GraphicalObject o) {
        return (o instanceof Node);
    }

    /**
     * @see BaseCheck
     * @param checker @see BaseCheck
     * @param o @see BaseCheck
     */
    public void apply(final StateChartCheckerBase checker,
            final GraphicalObject o) {

        Node n = (Node) o;
        Transition outgoing;
        Transition parallel;
        Iterator parallelTransIter;
        Iterator outgoingIter;

        CheckingProperties.getSemLog().log(LogFile.DETAIL,
                "RaceConditionCheck, State: " + n);

        // get outgoing transitions
        outgoingTransitions.clear();
        outgoingTransitions.addAll(n.getOutgoingTransitions());
        outgoingIter = outgoingTransitions.iterator();

        while (outgoingIter.hasNext()) {
            outgoing = (Transition) outgoingIter.next();
            if (outgoing.getLabel() instanceof CompoundLabel) {
                // care only about transitions that have an action
                if (((CompoundLabel) outgoing.getLabel())
                        .getEffect().getActions().length != 0) {
                    CheckingProperties.getSemLog().log(LogFile.DEBUG,
                            "Outgoing: " + outgoing);
                    parallelTransitions = getParallelTransitions(outgoing);
                    parallelTransIter = parallelTransitions.iterator();

                    while (parallelTransIter.hasNext()) {
                        parallel = (Transition) parallelTransIter.next();
                        if (parallel.getLabel() instanceof CompoundLabel) {
                            CheckingProperties.getSemLog().log(
                                    LogFile.DEBUG, "Parallel: " + parallel);
                            // care only about transitions that have an action
                            if (((CompoundLabel) parallel.getLabel())
                                    .getEffect().getActions().length != 0) {
                                haveWWRace(outgoing, parallel, checker);
                            }
                            haveWRRace(outgoing, parallel, checker);
                        } else {
                            CheckingProperties.getSemLog().log(LogFile.DEBUG,
                                    "WARNING: Transition " + parallel
                                    + " ignored, StringLabel");
                        }
                    }
                }
            } else {
                CheckingProperties.getSemLog().log(LogFile.DEBUG,
                        "WARNING: Transition " + outgoing
                        + " ignored, StringLabel");
            }
        }
    }

    /**
     * Find all transitions that can potentially occur in parallel with
     * Transition base.
     * @param base .
     * @return All transitions parallel to base.
     */
    private ArrayList getParallelTransitions(final Transition base) {

        ArrayList pTrans = new ArrayList();
        ArrayList parallelRegions;
        ArrayList stack = new ArrayList();
        Node stackNode;

        Node source = base.getSource();

        while (source != null) {
            source = source.getParent();
            if (source instanceof Region) {
                parallelRegions = source.getParent().getSubnodes();
                for (int i = 0; i < parallelRegions.size(); i++) {
                    if (!parallelRegions.get(i).equals(source)) {
                        stack.add(parallelRegions.get(i));
                        while (stack.size() > 0) {
                            stackNode = (Node) stack.remove(0);
                            pTrans.addAll(stackNode.getOutgoingTransitions());
                            if (stackNode instanceof CompositeState) {
                                stack.addAll(((CompositeState) stackNode)
                                        .getSubnodes());
                            }
                        }
                    }
                }
            }
        }
        return pTrans;
    }
```

```java
/**
 * Determines whether two transitions have a possible write/write race
 * condition.
 * @param t1 The first transition.
 * @param t2 The second transtion.
 * @param checker The StateChartCheckerBase.
 */
private void haveWWRace(final Transition t1, final Transition t2,
        final StateChartCheckerBase checker) {

    Action[] t1Actions = ((CompoundLabel) t1.getLabel()).getEffect()
                .getActions();
    Action[] t2Actions = ((CompoundLabel) t2.getLabel()).getEffect()
                .getActions();

    // iterate pairwise over all the actions of t1 and t2
    for (int i = 0; i < t1Actions.length; i++) {
        for (int j = 0; j < t2Actions.length; j++) {
            if (t1Actions[i] instanceof IntegerAssignment
                    && t2Actions[j] instanceof IntegerAssignment) {
                if (((IntegerAssignment) t1Actions[i]).getVariable()
                        == ((IntegerAssignment) t2Actions[j]).getVariable()
                        // makes sure there are no duplicate errors
                        && t1.toString().compareTo(t2.toString()) < 0) {
                    problemLocation.add(t1);
                    problemLocation.add(t2);
                    problemLocation.add(t1Actions[i]);
                    problemLocation.add(t2Actions[j]);
                    generateProblem(checker, problemLocation, 1);
                    problemLocation.clear();
                }
            } else if (t1Actions[i] instanceof GenerateValuedEvent
                    && t2Actions[j] instanceof GenerateValuedEvent) {
                if (((GenerateValuedEvent) t1Actions[i]).getEvent()
                        == ((GenerateValuedEvent) t2Actions[j]).getEvent()
                        // makes sure there are no duplicate errors
                        && t1.toString().compareTo(t2.toString()) < 0) {
                    // ignore CombineSingnals
                    if (!(((GenerateValuedEvent) t1Actions[i]).getEvent()
                            instanceof CombineSignal)) {
                        problemLocation.add(t1);
                        problemLocation.add(t2);
                        problemLocation.add(t1Actions[i]);
                        problemLocation.add(t2Actions[j]);
                        generateProblem(checker, problemLocation, 1);
                        problemLocation.clear();
                    }
                }
            }
        }
    }
}

/**
 * Determines whether two transitions have a possible write/read race
 * condition.
 * @param t1 The first transition.
 * @param t2 The second transtion.
 * @param checker The StateChartCheckerBase.
 */
private void haveWRRace(final Transition t1, final Transition t2,
        final StateChartCheckerBase checker) {

    Action[] t1Actions = ((CompoundLabel) t1.getLabel()).getEffect()
                .getActions();

    readVars.clear();
    setVars(((CompoundLabel) t2.getLabel()).getCondition());

    for (int i = 0; i < t1Actions.length; i++) {
        if (t1Actions[i] instanceof IntegerAssignment) {
            for (int j = 0; j < readVars.size(); j = j + 2) {
                if (readVars.get(j).toString().equals((
                        (IntegerAssignment) t1Actions[i]).
                                getVariable().toString())) {
                    problemLocation.add(t1);
                    problemLocation.add(t2);
                    problemLocation.add(t1Actions[i]);
                    if (readVars.get(j + 1) == null) {
                        problemLocation.add(readVars.get(j));
                    } else {
                        problemLocation.add(readVars.get(j + 1));
                    }
                    generateProblem(checker, problemLocation, 2);
                    problemLocation.clear();
                }
            }
        } else if (t1Actions[i] instanceof GenerateValuedEvent) {
            for (int j = 0; j < readVars.size(); j = j + 2) {
                if (readVars.get(j).toString().equals((
                        (GenerateValuedEvent) t1Actions[i]).
                                getEvent().toString())) {
                    problemLocation.add(t1);
                    problemLocation.add(t2);
                    problemLocation.add(t1Actions[i]);
                    if (readVars.get(j + 1) == null) {
                        problemLocation.add(readVars.get(j));
                        } else {
                            problemLocation.add(readVars.get(j + 1));
                        }
                        generateProblem(checker, problemLocation, 2);
                        problemLocation.clear();
                    }
                }
            }
        }
    }

    /**
     * Finds all boolean variables read in a transition condition.
     * @param bx The KIEL BooleanExpression.
     */
    public void setVars(final BooleanExpression bx) {

        if (bx instanceof Signal) {
            readVars.add(bx);
            readVars.add(null);

        } else if (bx instanceof BooleanVariable) {
            readVars.add(bx);
            readVars.add(null);

        } else if (bx instanceof Pre) {
            Pre p = (Pre) bx;
            setVars(p.getExpression());

        } else if (bx instanceof BooleanBrackets) {
            BooleanBrackets bb = (BooleanBrackets) bx;
            setVars(bb.getBody());

        } else if (bx instanceof BooleanAnd) {
            BooleanAnd ba = (BooleanAnd) bx;
            setVars(ba.getLeft());
            setVars(ba.getRight());

        } else if (bx instanceof BooleanOr) {
            BooleanOr bo = (BooleanOr) bx;
            setVars(bo.getLeft());
            setVars(bo.getRight());

        } else if (bx instanceof BooleanNot) {
            BooleanNot bn = (BooleanNot) bx;
            setVars(bn.getBody());

        } else if (bx instanceof BooleanComparator) {
            BooleanComparator bc = (BooleanComparator) bx;
            parent = bc;
            setIntVars(bc.getLeftExp());
            setIntVars(bc.getRightExp());
        }
    }

    /**
     * Finds all integer variables read in a transition condition.
     * @param ix The KIEL IntegerExpression.
     */
    public void setIntVars(final IntegerExpression ix) {

        if (ix instanceof IntegerSignal) {
            readVars.add(ix);
            readVars.add(parent);

        } else if (ix instanceof IntegerVariable) {
            readVars.add(ix);
            readVars.add(parent);

        } else if (ix instanceof Pre) {
            Pre p = (Pre) ix;
            setIntVars((IntegerExpression) p.getExpression());
        }
    }

    /**
     * This method is used internally by a check to generate the according
     * instance of a RobustnessProblem.
     * @param checker The checker where the problem will be added to.
     * @param objects The sources of the problem.
     * @param type Write/write or write/read.
     */
    private void generateProblem(final StateChartCheckerBase checker,
            final ArrayList objects, final int type) {

        Object[] args = new Object[numOfArgs];

        args[0] = objects.get(2);
        args[1] = objects.get(2 + 1);

        if (CheckingProperties.showTransitions()) {
            args[2] = ", at transitions " + objects.get(0)
                    + " and " + objects.get(1);
        } else {
            args[2] = "";
        }

        if (type == 1) {
                checker.getWarnings().add(new CheckingProblem(
                        objects, MessageFormat.format(message1, args)));
            } else {
                checker.getWarnings().add(new CheckingProblem(
                        objects, MessageFormat.format(message2, args)));
            }
        }
    }
```

A.4 Validity Checker Link-Up

A.4.1 MyValidityCheckerBase

```
//$Id: MyValidityCheckerBase.java,v 1.1 2006/04/10 10:22:22 gsc Exp $
package kiel.checking.semanticRobustnessChecks;

import java.util.ArrayList;

import kiel.dataStructure.CompoundLabel;

/**
 * <p>Description: Validity Checker base class.</p>
 *
 * <p>Copyright: (c) 2006</p>
 * <p>Company: Uni Kiel</p>
 * @author <a href="mailto:gsc@informatik.uni-kiel.de">Gunnar Schaefer</a>
 * @version $Revision: 1.1 $ last modified $Date: 2006/04/10 10:22:22 $
 */
public abstract class MyValidityCheckerBase {

    /**
     * Determines whether the predicates of two transitions overlap.
     *
     * @param cl1 The first transition.
     * @param cl2 The second transition.
     * @return True if the transition predicates overlap.
     */
    public abstract boolean areOverlapping(final CompoundLabel cl1,
            final CompoundLabel cl2);

    /**
     * Determines whether the predicates of two transition-groups overlap.
     *
     * @param labels1 The first group of transitions.
     * @param labels2 The second group of transitions.
     * @return True if the transition predicates overlap.
     */
    public abstract boolean areOverlappingMulti(final ArrayList labels1,
            final ArrayList labels2);

}
```

A.4.2 MyValidityCheckerBin

```
//$Id: MyValidityCheckerBin.java,v 1.2 2006/04/12 09:29:05 gsc Exp $
package kiel.checking.semanticRobustnessChecks;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.security.InvalidParameterException;
import java.util.ArrayList;

import javax.swing.JOptionPane;

import kiel.checking.CheckingProperties;
import kiel.dataStructure.CompoundLabel;
import kiel.dataStructure.boolexp.BooleanAnd;
import kiel.dataStructure.boolexp.BooleanBrackets;
import kiel.dataStructure.boolexp.BooleanComparator;
import kiel.dataStructure.boolexp.BooleanExpression;
import kiel.dataStructure.boolexp.BooleanFalse;
import kiel.dataStructure.boolexp.BooleanNot;
import kiel.dataStructure.boolexp.BooleanOr;
import kiel.dataStructure.boolexp.BooleanTrue;
import kiel.dataStructure.boolexp.BooleanTrueDummy;
import kiel.dataStructure.boolexp.BooleanVariable;
import kiel.dataStructure.boolexp.Equal;
import kiel.dataStructure.boolexp.GreaterOrEqual;
import kiel.dataStructure.boolexp.GreaterThan;
import kiel.dataStructure.boolexp.LessOrEqual;
import kiel.dataStructure.boolexp.LessThan;
import kiel.dataStructure.boolexp.NotEqual;
import kiel.dataStructure.boolexp.Pre;
import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.dataStructure.eventexp.Signal;
import kiel.dataStructure.eventexp.Tick;
import kiel.dataStructure.eventexp.TrueSignal;
import kiel.dataStructure.intexp.IntegerConstant;
import kiel.dataStructure.intexp.IntegerExpression;
import kiel.dataStructure.intexp.IntegerVariable;
import kiel.util.LogFile;

/**
 * <p>Description: Validity Checker using CVCL binary.</p>
 *
 * <p>Copyright: (c) 2006</p>
 * <p>Company: Uni Kiel</p>
 * @author <a href="mailto:gsc@informatik.uni-kiel.de">Gunnar Schaefer</a>
 * @version $Revision: 1.2 $ last modified $Date: 2006/04/12 09:29:05 $
 */
public final class MyValidityCheckerBin extends MyValidityCheckerBase {

    /**
     *
     */
    private BooleanExpression e1;

    /**
     *
     */
    private BooleanExpression e2;

    /**
     *
     */
    private BooleanExpression c1;

    /**
     *
     */
    private BooleanExpression c2;

    /**
     *
     */
    private Process cvcl = null;

    /**
     *
     */
    private BufferedReader cvclOutput = null;

    /**
     *
     */
    private PrintWriter cvclInput = null;

    /**
     *
     */
    private String cvclDecls;

    /**
     *
     */
    private String cvclQuery;
    /**
     *
     */
    private String cvclResult;


    /**
     * True if currently parsing body of a pre() expression.
     * Does not support nested pre’s.
     */
    private boolean isPre = false;

    /**
     * The default constructor.
     * @param cvclProc The CVCL process
     */
    public MyValidityCheckerBin(final Process cvclProc) {
        cvcl = cvclProc;
        cvclOutput = new BufferedReader(
                new InputStreamReader(cvcl.getInputStream()));
        cvclInput = new PrintWriter(cvcl.getOutputStream());
    }

    /**
     * @see MyValidityCheckerBase
     *
     * @param cl1 @see MyValidityCheckerBase
     * @param cl2 The @see MyValidityCheckerBase
     * @return @see MyValidityCheckerBase
     */
    public boolean areOverlapping(final CompoundLabel cl1,
            final CompoundLabel cl2) {

        e1 = cl1.getTrigger().getEventExpression();
        e2 = cl2.getTrigger().getEventExpression();
        c1 = cl1.getCondition();
        c2 = cl2.getCondition();

        cvclDecls = ""; //resetting
        cvclQuery = "QUERY NOT ("
                + "(" + triggerConvert(e1) + ")" + " AND "
                + "(" + conditionConvert(c1) + ")" + " AND "
                + "(" + triggerConvert(e2) + ")" + " AND "
                + "(" + conditionConvert(c2) + ")"
                + ");";

        CheckingProperties.getSemLog().log(LogFile.DEBUG,
                "cvclDecls: " + cvclDecls);
        CheckingProperties.getSemLog().log(LogFile.DETAIL,
                "cvclQuery: " + cvclQuery);

        cvclInput.println(cvclDecls);
        cvclInput.println("PUSH;");
        cvclInput.println(cvclQuery);
        cvclInput.println("POP;");
        cvclInput.flush();

        try {
            cvclResult = cvclOutput.readLine();
        } catch (IOException e) {
            JOptionPane.showMessageDialog(null, "CVCL binary not responding",
                    "Semantic Robustness Checker", JOptionPane.ERROR_MESSAGE);
            return false;
        }
        CheckingProperties.getSemLog().log(LogFile.DETAIL,
                "isSAT: " + cvclResult.endsWith("Invalid."));

        return cvclResult.endsWith("Invalid.");
    }

    /**
     * @see MyValidityCheckerBase
     *
     * @param labels1 @see MyValidityCheckerBase
     * @param labels2 The @see MyValidityCheckerBase
     * @return @see MyValidityCheckerBase
     */
    public boolean areOverlappingMulti(final ArrayList labels1,
            final ArrayList labels2) {

        CompoundLabel cl1;
        CompoundLabel cl2;
        String ex1 = "TRUE";
        String ex2 = "TRUE";

        cvclDecls = ""; //resetting

        for (int i = 0; i < labels1.size(); i++) {
            cl1 = (CompoundLabel) labels1.get(i);
            e1 = cl1.getTrigger().getEventExpression();
            c1 = cl1.getCondition();
            ex1 += " AND " + triggerConvert(e1)
                    + " AND " + conditionConvert(c1);
            CheckingProperties.getSemLog().log(LogFile.DEBUG,
                    "ex1: " + ex1);
        }
        for (int i = 0; i < labels2.size(); i++) {
            cl2 = (CompoundLabel) labels2.get(i);
```

```java
            e2 = cl2.getTrigger().getEventExpression();
            c2 = cl2.getCondition();
            ex2 += " AND " + triggerConvert(e2)
                    + " AND " + conditionConvert(c2);
            CheckingProperties.getSemLog().log(LogFile.DEBUG,
                    "ex2: " + ex2);
        }

        cvclQuery = "QUERY NOT (" + ex1 + " AND " + ex2 + ");";

        CheckingProperties.getSemLog().log(LogFile.DEBUG,
                "cvclDecls: " + cvclDecls);
        CheckingProperties.getSemLog().log(LogFile.DETAIL,
                "cvclQuery: " + cvclQuery);

        cvclInput.println(cvclDecls);
        cvclInput.println("PUSH;");
        cvclInput.println(cvclQuery);
        cvclInput.println("POP;");
        cvclInput.flush();

        try {
            cvclResult = cvclOutput.readLine();
        } catch (IOException e) {
            JOptionPane.showMessageDialog(null, "CVCL binary not responding",
                    "Semantic Robustness Checker", JOptionPane.ERROR_MESSAGE);
            return false;
        }
        CheckingProperties.getSemLog().log(LogFile.DETAIL,
                "cvclResult: " + cvclResult);

        return cvclResult.endsWith("Invalid.");
    }

    /**
     * Converts a KIEL Trigger BooleanExpression into an according JCVCL Expr.
     * @param bx The KIEL BooleanExpression.
     * @return An according JCVCL Expr.
     */
    public String triggerConvert(final BooleanExpression bx) {

        if (bx instanceof TrueSignal) {
            return "TRUE";

        } else if (bx instanceof Tick) {
            return "TRUE";

        } else if (bx instanceof Pre) {
            //maybe true, maybe false => acts like a boolean signal
            Pre p = (Pre) bx;
            isPre = true;            //does not support nested pre's
            String s = triggerConvert(p.getExpression());
            isPre = false;
            return s;

        } else if (bx instanceof Signal) {
            if (isPre) {
                cvclDecls += "pre_e_" + bx.toString() + " : BOOLEAN; ";
                return "pre_e_" + bx.toString();
            } else {
                cvclDecls += "e_" + bx.toString() + " : BOOLEAN; ";
                return "e_" + bx.toString();
            }

        } else if (bx instanceof BooleanVariable) {
            if (isPre) {
                cvclDecls += "pre_e_" + bx.toString() + " : BOOLEAN; ";
                return "pre_e_" + bx.toString();
            } else {
                cvclDecls += "e_" + bx.toString() + " : BOOLEAN; ";
                return "e_" + bx.toString();
            }

        } else if (bx instanceof BooleanBrackets) {
            BooleanBrackets bb = (BooleanBrackets) bx;
            return "(" + triggerConvert(bb.getBody()) + ")";

        } else if (bx instanceof BooleanAnd) {
            BooleanAnd ba = (BooleanAnd) bx;
            return triggerConvert(ba.getLeft()) + " AND "
                + triggerConvert(ba.getRight());

        } else if (bx instanceof BooleanOr) {
            BooleanOr bo = (BooleanOr) bx;
            return triggerConvert(bo.getLeft()) + " OR "
            + triggerConvert(bo.getRight());

        } else if (bx instanceof BooleanNot) {
            BooleanNot bn = (BooleanNot) bx;
            return "NOT " + triggerConvert(bn.getBody());

        } else {
            throw new InvalidParameterException("\"" + bx.getClass().getName()
                    + "\" is not a supported trigger expression.");
        }
    }

    /**
```

```
 * Converts a KIEL Condition BooleanExpression into an according JCVCL Expr.
 * @param bx The KIEL BooleanExpression.
 * @return An according JCVCL Expr.
 */
public String conditionConvert(final BooleanExpression bx) {

    if (bx instanceof BooleanVariable) {
        cvclDecls += "c_" + bx.toString() + " : BOOLEAN; ";
        return "c_" + bx.toString();

    } else if (bx instanceof BooleanTrueDummy) {
        return "TRUE";

    } else if (bx instanceof BooleanTrue) {
        return "TRUE";

    } else if (bx instanceof BooleanFalse) {
        return "FALSE";

    } else if (bx instanceof BooleanBrackets) {
        BooleanBrackets bb = (BooleanBrackets) bx;
        return "(" + conditionConvert(bb.getBody()) + ")";

    } else if (bx instanceof BooleanAnd) {
        BooleanAnd ba = (BooleanAnd) bx;
        return conditionConvert(ba.getLeft()) + " AND "
            + conditionConvert(ba.getRight());

    } else if (bx instanceof BooleanOr) {
        BooleanOr bo = (BooleanOr) bx;
        return conditionConvert(bo.getLeft()) + " OR "
            + conditionConvert(bo.getRight());

    } else if (bx instanceof BooleanNot) {
        BooleanNot bn = (BooleanNot) bx;
        return "NOT " + conditionConvert(bn.getBody());

    } else if (bx instanceof BooleanComparator) {
        return compConvert((BooleanComparator) bx);

    } else {
        throw new InvalidParameterException("\"" + bx.getClass().getName()
                + "\" is not a supported condition expression.");
    }
}

/**
 * Converts a KIEL BooleanComparator into an according JCVCL Expr.
 * @param bc The KIEL BooleanComparator.
 * @return An according JCVCL Expr.
 */
public String compConvert(final BooleanComparator bc) {

    if (bc instanceof Equal) {
        return intExConvert(bc.getLeftExp()) + " = "
            + intExConvert(bc.getRightExp());

    } else if (bc instanceof GreaterOrEqual) {
        return intExConvert(bc.getLeftExp()) + " >= "
            + intExConvert(bc.getRightExp());

    } else if (bc instanceof GreaterThan) {
        return intExConvert(bc.getLeftExp()) + " > "
            + intExConvert(bc.getRightExp());

    } else if (bc instanceof LessOrEqual) {
        return intExConvert(bc.getLeftExp()) + " <= "
            + intExConvert(bc.getRightExp());

    } else if (bc instanceof LessThan) {
        return intExConvert(bc.getLeftExp()) + " < "
            + intExConvert(bc.getRightExp());

    } else if (bc instanceof NotEqual) {
        return "NOT (" + intExConvert(bc.getLeftExp()) + " = "
            + intExConvert(bc.getRightExp()) + ")";

    } else {
        throw new InvalidParameterException("\"" + bc.getClass().getName()
                + "\" is not a supported comparator expression.");
    }
}

/**
 * Converts a KIEL IntegerExpression into an according JCVCL Expr.
 * @param ix The KIEL IntegerExpression.
 * @return An according JCVCL Expr.
 */
public String intExConvert(final IntegerExpression ix) {

    if (ix instanceof IntegerConstant) {
        return ix.toIntExpString();

    } else if (ix instanceof IntegerSignal) {
        if (isPre) {
            cvclDecls += "pre_c_" + ix.toString() + " : INT; ";
            return "pre_c_" + ix.toString();
        } else {
                cvclDecls += "c_" + ix.toString() + " : INT; ";
                return "c_" + ix.toString();
            }

        } else if (ix instanceof IntegerVariable) {
            if (isPre) {
                cvclDecls += "pre_c_" + ix.toString() + " : INT; ";
                return "pre_c_" + ix.toString();
            } else {
                cvclDecls += "c_" + ix.toString() + " : INT; ";
                return "c_" + ix.toString();
            }

        } else if (ix instanceof Pre) {
            //maybe true, maybe false => acts like a boolean signal
            Pre p = (Pre) ix;
            isPre = true;           //does not support nested pre's
            String s = intExConvert((IntegerExpression) p.getExpression());
            isPre = false;
            return s;

        } else {
            throw new InvalidParameterException("\"" + ix.getClass().getName()
                    + "\" is not a supported integer expression.");
        }
    }
}
```

A.4.3 MyValidityCheckerLib

```
//$Id: MyValidityCheckerLib.java,v 1.1 2006/04/10 10:22:22 gsc Exp $
package kiel.checking.semanticRobustnessChecks;

import java.security.InvalidParameterException;
import java.util.ArrayList;

import kiel.checking.CheckingProperties;
import kiel.dataStructure.CompoundLabel;
import kiel.dataStructure.boolexp.BooleanAnd;
import kiel.dataStructure.boolexp.BooleanBrackets;
import kiel.dataStructure.boolexp.BooleanComparator;
import kiel.dataStructure.boolexp.BooleanExpression;
import kiel.dataStructure.boolexp.BooleanFalse;
import kiel.dataStructure.boolexp.BooleanNot;
import kiel.dataStructure.boolexp.BooleanOr;
import kiel.dataStructure.boolexp.BooleanTrue;
import kiel.dataStructure.boolexp.BooleanTrueDummy;
import kiel.dataStructure.boolexp.BooleanVariable;
import kiel.dataStructure.boolexp.Equal;
import kiel.dataStructure.boolexp.GreaterOrEqual;
import kiel.dataStructure.boolexp.GreaterThan;
import kiel.dataStructure.boolexp.LessOrEqual;
import kiel.dataStructure.boolexp.LessThan;
import kiel.dataStructure.boolexp.NotEqual;
import kiel.dataStructure.boolexp.Pre;
import kiel.dataStructure.eventexp.IntegerSignal;
import kiel.dataStructure.eventexp.Signal;
import kiel.dataStructure.eventexp.Tick;
import kiel.dataStructure.eventexp.TrueSignal;
import kiel.dataStructure.intexp.IntegerConstant;
import kiel.dataStructure.intexp.IntegerExpression;
import kiel.dataStructure.intexp.IntegerVariable;
import kiel.util.LogFile;
import kiel.util.jcvcl.Expr;
import kiel.util.jcvcl.ValidityChecker;

/**
 * <p>Description: Validity Checker using JCVCL library and JNI.</p>
 *
 * <p>Copyright: (c) 2006</p>
 * <p>Company: Uni Kiel</p>
 * @author <a href="mailto:gsc@informatik.uni-kiel.de">Gunnar Schaefer</a>
 * @version $Revision: 1.1 $ last modified $Date: 2006/04/10 10:22:22 $
 */
public final class MyValidityCheckerLib extends MyValidityCheckerBase {

    /**
     *
     */
    private ValidityChecker vc = ValidityChecker.create();

    /**
     *
     */
    private BooleanExpression e1;

    /**
     *
     */
    private BooleanExpression e2;

    /**
     *
     */
    private BooleanExpression c1;
    /**
     *
     */
    private BooleanExpression c2;

    /**
     *
     */
    private Expr cvclE1;

    /**
     *
     */
    private Expr cvclE2;

    /**
     *
     */
    private Expr cvclC1;

    /**
     *
     */
    private Expr cvclC2;

    /**
     * True if currently parsing body of a pre() expression.
     * Does not support nested pre's.
     */
    private boolean isPre = false;

    /**
     * Checks the satisfiability of the expression.
     * @param ex The expression to check.
     * @return True if the expression is satisfiable.
     */
    public boolean isSat(final Expr ex) {

        vc.push();
        boolean isSat = !vc.query(ex.negate());
        vc.pop();

        CheckingProperties.getSemLog().log(LogFile.DETAIL,
                "isSAT: " + isSat);

        return isSat;
    }

    /**
     * @see MyValidityCheckerBase
     *
     * @param cl1 @see MyValidityCheckerBase
     * @param cl2 The @see MyValidityCheckerBase
     * @return @see MyValidityCheckerBase
     */
    public boolean areOverlapping(final CompoundLabel cl1,
            final CompoundLabel cl2) {

        e1 = cl1.getTrigger().getEventExpression();
        e2 = cl2.getTrigger().getEventExpression();
        c1 = cl1.getCondition();
        c2 = cl2.getCondition();

        cvclE1 = triggerConvert(e1);
        cvclE2 = triggerConvert(e2);
        cvclC1 = conditionConvert(c1);
        cvclC2 = conditionConvert(c2);

        CheckingProperties.getSemLog().log(LogFile.DEBUG,
                "cvclE1: " + cvclE1);
        CheckingProperties.getSemLog().log(LogFile.DEBUG,
                "cvclE2: " + cvclE2);
        CheckingProperties.getSemLog().log(LogFile.DEBUG,
                "cvclC1: " + cvclC1);
        CheckingProperties.getSemLog().log(LogFile.DEBUG,
                "cvclC2: " + cvclC2);

        CheckingProperties.getSemLog().log(LogFile.DETAIL,
                "CVCL query: [" + cvclE1 + "] /\\ [" + cvclC1
                + "] /\\ [" + cvclE2 + "] /\\ [" + cvclC2 + "]");

        return isSat(vc.andExpr(vc.andExpr(cvclE1, cvclC1),
                (vc.andExpr(cvclE2, cvclC2))));
    }

    /**
     * @see MyValidityCheckerBase
     *
     * @param labels1 @see MyValidityCheckerBase
     * @param labels2 The @see MyValidityCheckerBase
     * @return @see MyValidityCheckerBase
     */
    public boolean areOverlappingMulti(final ArrayList labels1,
            final ArrayList labels2) {

        CompoundLabel cl1;
        CompoundLabel cl2;
        Expr ex1 = vc.trueExpr();
        Expr ex2 = vc.trueExpr();

        for (int i = 0; i < labels1.size(); i++) {
            cl1 = (CompoundLabel) labels1.get(i);
            e1 = cl1.getTrigger().getEventExpression();
            c1 = cl1.getCondition();
            cvclE1 = triggerConvert(e1);
            cvclC1 = conditionConvert(c1);
            ex1 = vc.andExpr(ex1, vc.andExpr(cvclE1, cvclC1));
            CheckingProperties.getSemLog().log(LogFile.DEBUG,
                    "ex1: " + ex1);
        }
        for (int i = 0; i < labels2.size(); i++) {
            cl2 = (CompoundLabel) labels2.get(i);
            e2 = cl2.getTrigger().getEventExpression();
            c2 = cl2.getCondition();
            cvclE2 = triggerConvert(e2);
            cvclC2 = conditionConvert(c2);
            ex2 = vc.andExpr(ex2, vc.andExpr(cvclE2, cvclC2));
            CheckingProperties.getSemLog().log(LogFile.DEBUG,
                    "ex2: " + ex2);
        }

        return isSat(vc.andExpr(ex1, ex2));
    }

    /**
     * Converts a KIEL Trigger BooleanExpression into an according JCVCL Expr.
     * @param bx The KIEL BooleanExpression.
     * @return An according JCVCL Expr.
     */
    public Expr triggerConvert(final BooleanExpression bx) {

        if (bx instanceof TrueSignal) {
            return vc.trueExpr();

        } else if (bx instanceof Tick) {
            return vc.trueExpr();

        } else if (bx instanceof Pre) {
            //maybe true, maybe false => acts like a boolean signal
            Pre p = (Pre) bx;
            isPre = true;           //does not support nested pre’s
            Expr e = triggerConvert(p.getExpression());
            isPre = false;
            return e;

        } else if (bx instanceof Signal) {
            if (isPre) {
                return vc.varExpr("e_pre_" + bx.toString(), vc.boolType());
            } else {
                return vc.varExpr("e_" + bx.toString(), vc.boolType());
            }

        } else if (bx instanceof BooleanVariable) {
            if (isPre) {
                return vc.varExpr("e_pre_" + bx.toString(), vc.boolType());
            } else {
                return vc.varExpr("e_" + bx.toString(), vc.boolType());
            }

        } else if (bx instanceof BooleanBrackets) {
            BooleanBrackets bb = (BooleanBrackets) bx;
            return triggerConvert(bb.getBody());

        } else if (bx instanceof BooleanAnd) {
            BooleanAnd ba = (BooleanAnd) bx;
            return vc.andExpr(triggerConvert(ba.getLeft()),
                    triggerConvert(ba.getRight()));

        } else if (bx instanceof BooleanOr) {
            BooleanOr bo = (BooleanOr) bx;
            return vc.orExpr(triggerConvert(bo.getLeft()),
                    triggerConvert(bo.getRight()));

        } else if (bx instanceof BooleanNot) {
            BooleanNot bn = (BooleanNot) bx;
            return vc.notExpr(triggerConvert(bn.getBody()));

//          } else if (bx instanceof IntegerSignal) {

        } else {
            throw new InvalidParameterException("\"" + bx.getClass().getName()
                    + "\" is not a supported trigger expression.");
        }
    }


    /**
     * Converts a KIEL Condition BooleanExpression into an according JCVCL Expr.
     * @param bx The KIEL BooleanExpression.
     * @return An according JCVCL Expr.
     */
    public Expr conditionConvert(final BooleanExpression bx) {

        if (bx instanceof BooleanVariable) {
            return vc.varExpr("c_" + bx.toString(), vc.boolType());
        } else if (bx instanceof BooleanTrueDummy) {
            return vc.trueExpr();

        } else if (bx instanceof BooleanTrue) {
            return vc.trueExpr();

        } else if (bx instanceof BooleanFalse) {
            return vc.falseExpr();

        } else if (bx instanceof BooleanBrackets) {
            BooleanBrackets bb = (BooleanBrackets) bx;
            return conditionConvert(bb.getBody());

        } else if (bx instanceof BooleanAnd) {
            BooleanAnd ba = (BooleanAnd) bx;
            return vc.andExpr(conditionConvert(ba.getLeft()),
                    conditionConvert(ba.getRight()));

        } else if (bx instanceof BooleanOr) {
            BooleanOr bo = (BooleanOr) bx;
            return vc.orExpr(conditionConvert(bo.getLeft()),
                    conditionConvert(bo.getRight()));

        } else if (bx instanceof BooleanNot) {
            BooleanNot bn = (BooleanNot) bx;
            return vc.notExpr(conditionConvert(bn.getBody()));

        } else if (bx instanceof BooleanComparator) {
            return compConvert((BooleanComparator) bx);

        } else {
            throw new InvalidParameterException("\"" + bx.getClass().getName()
                    + "\" is not a supported condition expression.");
        }
    }


    /**
     * Converts a KIEL BooleanComparator into an according JCVCL Expr.
     * @param bc The KIEL BooleanComparator.
     * @return An according JCVCL Expr.
     */
    public Expr compConvert(final BooleanComparator bc) {

        if (bc instanceof Equal) {
            return vc.eqExpr(intExConvert(bc.getLeftExp()),
                     intExConvert(bc.getRightExp()));

        } else if (bc instanceof GreaterOrEqual) {
            return vc.geExpr(intExConvert(bc.getLeftExp()),
                    intExConvert(bc.getRightExp()));

        } else if (bc instanceof GreaterThan) {
            return vc.gtExpr(intExConvert(bc.getLeftExp()),
                    intExConvert(bc.getRightExp()));

        } else if (bc instanceof LessOrEqual) {
            return vc.leExpr(intExConvert(bc.getLeftExp()),
                    intExConvert(bc.getRightExp()));

        } else if (bc instanceof LessThan) {
            return vc.ltExpr(intExConvert(bc.getLeftExp()),
                    intExConvert(bc.getRightExp()));

        } else if (bc instanceof NotEqual) {
            return vc.notExpr(vc.eqExpr(intExConvert(bc.getLeftExp()),
                    intExConvert(bc.getRightExp())));

        } else {
            throw new InvalidParameterException("\"" + bc.getClass().getName()
                    + "\" is not a supported comparator expression.");
        }
    }


    /**
     * Converts a KIEL IntegerExpression into an according JCVCL Expr.
     * @param ix The KIEL IntegerExpression.
     * @return An according JCVCL Expr.
     */
    public Expr intExConvert(final IntegerExpression ix) {

        if (ix instanceof IntegerConstant) {
            return vc.ratExpr(((IntegerConstant) ix).getValue());

        } else if (ix instanceof IntegerSignal) {
            if (isPre) {
                return vc.varExpr("c_pre_" + ix.toString(), vc.intType());
            } else {
                return vc.varExpr("c_" + ix.toString(), vc.intType());
            }

        } else if (ix instanceof IntegerVariable) {
            if (isPre) {
                return vc.varExpr("c_pre_" + ix.toString(), vc.intType());
            } else {
                return vc.varExpr("c_" + ix.toString(), vc.intType());
                }

            } else if (ix instanceof Pre) {
                //maybe true, maybe false => acts like a boolean signal
                Pre p = (Pre) ix;
                isPre = true;           //does not support nested pre’s
                Expr e = intExConvert((IntegerExpression) p.getExpression());
                isPre = false;
                return e;

            } else {
                throw new InvalidParameterException("\"" + ix.getClass().getName()
                        + "\" is not a supported integer expression.");
            }
        }
    }
```

LaTeX