# Analyzing Robustness of UML State Machines

Steffen Prochnow, Gunnar Schaefer, Ken Bell, and Reinhard von Hanxleden

{spr,gsc,kbe,rvh}@informatik.uni-kiel.de
Real-Time and Embedded Systems Group, Department of Computer Science
Christian-Albrechts-Universität zu Kiel, Olshausenstr. 40, D-24118 Kiel, Germany

**Abstract.** State Machines constitute an integral part of software behavior specification within the UML. The development of realistic software applications often results in complex and distributed models. Potential errors can be very subtle and hard to locate for the developer. Thus, we present a set of *robustness rules* that seek to avoid common types of errors by ruling out certain modeling constructs. Furthermore, adherence to these rules can improve model readability and maintainability. The robustness rules constitute a general Statechart style guide for different dialects, such as UML State Machines, *Statemate*, and *Esterel Studio*. Based on this style guide, an automated checking framework has been implemented as a plug-in for the prototypical Statechart modeling tool *KIEL*. Simple structural checks can be formulated in a compact, abstract manner in the OCL. The framework can also incorporate checks that go beyond the expressiveness of OCL by implementing them directly in Java, which can also serve as a gateway to formal verification tools; we have exploited this to incorporate a theorem prover for more advanced checks. As a case study, we have adopted the UML *well-formedness rules*; this confirms that individual rules are easily incorporated into the framework.

## 1 Introduction

Statecharts [1] constitute a widely accepted formalism for the specification of reactive real-time systems. They extend the classical formalism of finite-state machines and state transition diagrams by incorporating notions of hierarchy, orthogonality, compound events, and a broadcast mechanism for communication between concurrent components. Statecharts provide an effective graphical notation, not only for the specification and design of reactive systems, but also for the simulation of the modeled system behavior. They have been incorporated into the Unified Modeling Language (UML) (as object-based State Machines) [2] and are supported by several commercial tools, *e. g.*, *Esterel Studio*[1], *Matlab/Simulink/Stateflow*[2], and UML *CASE* tools, such as *Rhapsody*[3] and *ArgoUML*[4].

[1] http://www.esterel-technologies.com/products/esterel-studio/
[2] http://www.mathworks.com/products/stateflow/
[3] http://www.ilogix.com/
[4] http://argouml.tigris.org/

The assurance of quality, *i. e.*, ensuring readability and avoiding error-prone constructs, is one of the most essential aspects in the development of safety-critical reactive systems, since the failure of such systems—often attributable to programming flaws—can cause loss of property or even human life. As Parnas has observed, human code reviews are time-consuming and highly undependable in revealing errors [3]. Taking part of the burden off the reviewers, as well as off the designers, is the rationale for *automated error prevention*, where a computer performs preliminary checks. Hence, this paper is a contribution to ensure certain aspects of safety in developing Statecharts. We achieve this by applying methods of automated error-source detection.

We propose a rule set that forms a fundamental Statechart style guide. Based on this well-structured set of robustness rules, both syntactic and semantic, an automated checking framework has been implemented as a plug-in for the *Kiel Integrated Environment for Layout* Statechart modeling tool[5]. A key objective in devising the rule set and in designing the checking framework was to not restrict the modelers' creativity, but to achieve more explicit, easy to comprehend, and less error-prone models. Our approach, therefore, was developed adhering to the following requirements:

**Modularity and Configurability:** All robustness checks are independently implemented, individually selectable, and parametrizable via a preferences management.

**Extendability of the rule set:** The set of checks is easily extendable by either adding a constraint, specified in the Object Constraint Language (OCL) [4], or by implementing a new Java class.

**Automatic conformance checking:** Compliance with the 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.

Even though a wide range of applications for Statechart verification already exists, none fulfills all needs. They are either highly specialized and therefore not extendable or they are extendable but do not provide the possibility to check complex problems. In contrast, we present a general approach to style checking in Statecharts. It is easily extendable and incorporates a theorem prover to provide for complex semantic checks. The main contributions of this paper are:

– An inspection and classification of error prevention methods for software in general as well as with a focus on style checking in Statecharts (Section 3),
– a comparison of existing style guides and applications for textual programming languages and Statecharts (Section 3.3),
– a collection of *robustness rules* for less error-prone Statecharts (Section 4),
– a checking framework, which automatically and quickly evaluates rules defined in OCL; additionally, checks based on theorem proving are evaluated (Section 5), and

[5] `http://www.informatik.uni-kiel.de/~rt-kiel/`

– an experimental evaluation, based on the aforementioned checking rules, showing the applicability and efficiency of our robustness rules (Section 7).

## 2 Related Work

Error prevention in software development is as old as the field of software development itself. Therefore, many style guides for classical textual programming languages have been developed, dealing not only with code layout, but also with robustness aspects; *e. g.,* MISRA proposed a C programming style [5], Sun proposed a Java programming style[6]. Style guides have been developed for the Statecharts modeling paradigm as well, *e. g.,* by the MathWorks Automotive Advisory Board (MAAB)[7] and by Ford Motor Company[8], both exclusively for *Simulink/Stateflow*. Scaife *et al.* [6] propose the development of a *safe subset* of the *Stateflow* language, which is considered to be less error-prone. Furthermore, Kreppold [7] has presented a style guide for *Statemate*.

In the context of UML State Machines, the *well-formedness rule*s defined within the UML specification clarify the semantics of Statechart elements. Besides the *well-formedness rule*s, other rules for UML State Machines were formulated, *e. g.,* by Mutz [8]. For our style guide we pick up some of these rules; furthermore, we specify dialect independent as well as dependent rules inspired by different sources and add rules based on our own experience.

Automatically checking robustness (or soundness) of UML State Machines is an active field of research. Pap *et al.* [9] have investigated applicable approaches. The presented techniques include checks based on the OCL, graph transformation, special programs and finally reachability analysis driven tests. Richters [10] has investigated different frameworks that can be used when it comes to working with OCL.

A wide range of available CASE tools provide OCL support, which is generally limited to gathering constraints. Beyond, Mutz and Huhn [8, 11] have developed the *Rule Checker* for the automated analysis of user-defined design rules on UML State Machines. They pursue an interpreter-based analysis for the evaluation of OCL. However, an interpretative approach is generally considered less flexible and slower than an executive. Additionally, simple syntactic checks are executed by Java programs. No sophisticated checks involving a theorem prover are performed.

Another approach to check the style guide conformance of Statecharts is *Mint*[9] by Ricardo which is focused on the MAAB style guide. The checker primarily aims at achieving a consistent look-and-feel, enhancing readability, and avoiding common modeling errors. The *Guideline-Checker* [12], coded in *Matlab*, is a no-cost/academic alternative to *Mint*. The range of the *Guideline-Checker*

[6] `http://java.sun.com/docs/codeconv/`
[7] `http://www.mathworks.com/industries/auto/maab.html`
[8] `http://vehicle.berkeley.edu/mobies/papers/stylev242.pdf`
[9] `http://www.ricardo.com/engineeringservices/controlelectronics.aspx?page=mint`

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” [12, page 26].

Moreover, special programs for the detection of specific problems have been developed. Here, the *State Analyzer* [13], developed within DaimlerChrysler’s R&D, is a prototypical software tool to check the “determinism” of *Statemate* Statecharts. 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 (cf. Section 5), *i. e.*, proving the satisfiability of a formula consisting of the conjunction of each pair of transition predicates. Approaches analyzing requirements specifications are introduced by, *e. g.*, Heitmeyer *et al.* [14] for the Software Cost Reduction (SCR) formalism; Heimdahl and Leveson [15] present a similar approach for the Requirements State Machine Language (RSML).

In summary, none of the discussed methods and tools fulfill all of our needs. Therefore, we present a Statechart robustness analysis approach, based on the execution of Java code synthesized from OCL rules, that combines the usability and flexibility of OCL and—beyond the approach of Mutz and Huhn—the mightiness of a theorem prover.

## 3 Errors and Error Prevention in the Modeling of Statecharts

To support the early detection and elimination of modeling errors, a design methodology must provide effective communication among the various design stages of the product. This section gives an overview of common error sources in developing Statecharts and how these may be avoided.

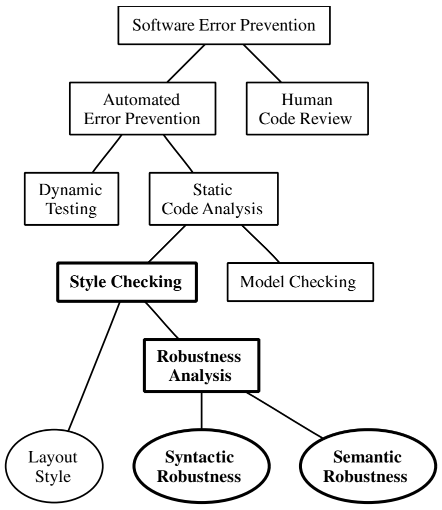
### 3.1 Sources of Errors

Errors in development of graphical models like Statecharts have a large diversity of types and reasons. A paramount cause of producing erroneous Statecharts is apparently a misunderstanding of the utilized modeling tools and their simulation behavior. This may have its source in *counterintuitive specifications* of the model semantics (*e. g.*, unbound behavior) and a lacking comprehension of the modeler.

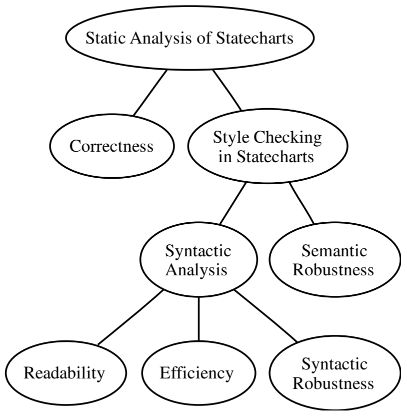
Errors also originate from the often *large size of graphical models*: Because of the extensive requirements in software design technology, the dimensions of graphical models can increase enormously. Moreover, Statecharts often are of great *complexity*: Because of the discrete nature of Statecharts, small changes not always have small effects. Beyond, Statecharts represent *interactive and distributed systems*: large collections of interconnected components usually involve interactive and concurrent processes. Therefore, potential errors can be very subtle and hard to locate for human developers.

## 3.2 Error Prevention

The approaches to error prevention in textual and visual languages face essentially the same problems. Due to this, we propose a common error prevention taxonomy and refine it in the following for Statecharts. Software error prevention in general encompasses a number of different techniques designed to identify programming flaws. As outlined in Figure 1a, we can basically distinguish between automated error prevention and human code review. As already pointed out, human code reviews are exceedingly time-consuming and often undependable in revealing errors. However, they may find conceptual problems that are impossible to detect automatically.



(a) Software Error Prevention in General and its Taxonomy.



(b) Taxonomy for Style Checking in Statecharts as a Refinement of General Software Error Prevention.

Fig. 1: Classification of Software Error Prevention.

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

*Style checking*, another aspect of static code analysis, is concerned with layout style, *i. e.,* common appearance, as well as syntactic and semantic style. The latter two are often collectively referred to as robustness analysis (see below). Style checking always requires the syntactic and semantic correctness of the code. *Robustness analysis* refers to the objective of eliminating certain types of errors and enforcing sound engineering practices. Robustness rules limit the general range of a given modeling/programming language, as they are entirely independent of what is being designed.

In the general context of static code analysis, one must distinguish syntactic and semantic correctness on the one hand and style checking on the other hand. On this foundation, as a first step toward systematically devising an extensive style guide for Statecharts, the following taxonomy, depicted in Figure 1b, was laid down:

**Syntactic Analysis:** The enforcement of syntax-related rules does, in general, not necessitate knowledge of model semantics.

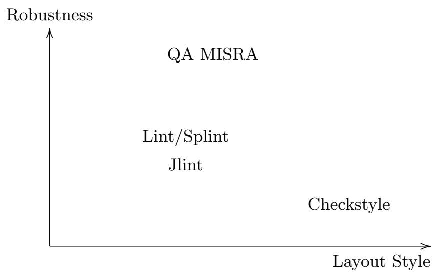
- *Readability* (or layout style) aims at a graphical normal form, *e. g.,* transitions connect states in a clockwise direction, charts contain a limited number of states, *etc.*
- *Efficiency* (or compactness, simplicity) emphasizes superfluous and redundant elements from the Statechart model.
- *Syntactic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

**Semantic Robustness:** Deriving and enforcing semantic robustness rules requires knowledge of specific aspects of the model semantics. Exact analysis typically requires the use of formal verification tools.

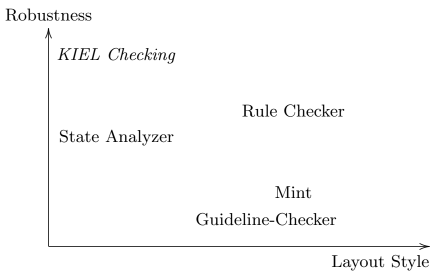
## 3.3 Existing Style Guides and Applications

Style checking is based upon *style guides*. They constitute a set of design rules, concerning textual programming, respectively the modeling of Statecharts. Style guides provide general instructions on how to use languages. They are commonly provided as (in-)formal specifications, containing lists of rules. Style guides concern human languages, textual programming languages, as well as visual programming languages, such as Statecharts. They define a subset of usable elements. The informal as well as the formal specifications are primarily *operational instructions for humans*. These affect the programmed or modeled result. Beyond, formal style guides act as the configuration for *automated style checking*, *i. e.,* style checkers.

Since programming style often depends on the programming language, different coding standards and related code checking tools exist for different programming languages. Akin to coding standards, most code checking tools are programming language-specific. Available code checkers for C are *e. g., Lint* [16], *LCLint* (aka. *Splint*) [17] and QA MISRA[10]; code checkers for Java are Jlint[11] and Checkstyle[12]. Figure 2a roughly classifies these code checkers according to their emphasis on layout style *vs.* robustness—a major distinction within style checking (see Section 3.2).



(a) Checking Tools for Textual Programming Languages.



(b) Checking Tools for Statecharts.

Fig. 2: Classification of Checking Tools according to their Emphasis of Layout Style *vs.* Robustness.

Statechart style checking is much less developed and less sophisticated as compared to style checking in textual computer programming. Nevertheless, when analyzing the dynamics of reactive systems, it is all the more important that models are designed according to approved rules. Therefore, several Statechart modeling tools, *e. g.*, *Stateflow* and *Statemate*, have been supplemented with a number of checks. Four representative checking tools—*Mint* and the *Guideline-Checker* related to *Stateflow*, the *State Analyzer* related to *Statemate*, and the *Rule Checker*—as well as our own robustness checker (see Section 5), are roughly classified according to their emphasis of layout style *vs.* robustness in Figure 2b.

The *Guideline-Checker* and the *State Analyzer* 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 particularly important since they eliminate possible non-trivial sources of error, which are very hard to discern for humans. The rules put forth in the next section aspire to fill this gap.

[10] `http://www.programmingresearch.com/`
[11] `http://jlint.sourceforge.net/`
[12] `http://checkstyle.sourceforge.net/`

## 4 Statechart Style Guide

Building on the aforementioned theoretical foundation, practical experience, and available prototypes, this section outlines a comprehensive Statechart style guide, striving for general applicability to Statechart dialects, within the limits of the UML State Machines specification. The rules presented below were formulated following the advice of Buck and Rau [18]: Clarity, Minimality, Consistency, Consensus, Flexibility, Adaptability, Stability and Testability.

As mentioned above (cf. Section 3.2), style guides for Statecharts can roughly be divided into two parts, namely syntactical analysis on the one hand and semantical analysis on the other hand. Syntactical analysis addresses the syntactical structure of Statecharts, such as layout, possible optimizations, and robustness problems. Therefore, in the context of syntactical rules, one basically has to focus on problems that deal with the relations of individual Statechart elements to each other. Furthermore, syntactical analysis opens up two fields of possible applications. One field analyses whether the syntactical relation of the elements used corresponds to the rules specified by a certain dialect (*i. e.*, syntactical correctness). Within the UML these kind of rules are called *well-formedness rule*s. The *well-formedness rule*s "[. . . ] specify constraints over attributes and associations defined [with]in the [Statechart] meta model" [19, Section 2.3.2.2].

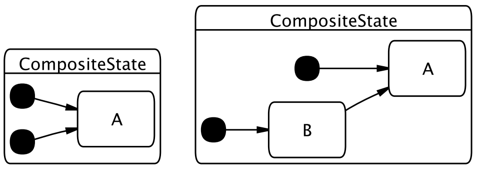
Nevertheless, locating problems from the part of syntactical correctness and syntactical robustness works the same way. Since Statecharts are directed graphs, one can use pattern matching here. If used for locating problems one would create a pattern that captures the problem.

In the following, we present the rules incorporated into our Statechart style guide. Following the proposed taxonomy (see Figure 1b), the rules are grouped in different sections. First of all, the rules dealing with the syntactical correctness, the *well-formedness rule*s, are presented. On that foundation, we extend the style guide by afterwards presenting the rules for syntactical robustness. Finally, the rules for semantical robustness are presented.

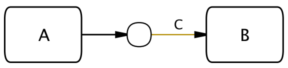
### UML Well-formedness Rules

As mentioned above, syntactical correctness is mandatory for robustness. Therefore, it is necessary to check, whether a Statechart is syntactically correct or not. For most Statechart dialects, this is done within a dialect dependent modeling tool. But when dealing with UML State Machines, one has to manually make sure that the above mentioned *well-formedness rule*s are preserved as some UML tools do not check those rules at all. Within the UML, the *well-formedness rule*s themselves are described using OCL. Given a context of application and the constraint itself, problems are detected fairly easy. In the following, we present some examples for violations of the *well-formedness rule*s. Section 5 elaborates on the OCL implementation of the presented examples.

The rule *CompositeState-1* denotes that "a composite can have at most one initial vertex" [19, Section 2.12.3.1]. Detecting violations of this rule, as presented in Figure 3a (left-hand side), is done by a two-part pattern. One part contains a composite state with no initial vertex and the other part contains a composite state with one initial vertex. If neither part matches the composite state is known to have more than one initial vertex. Fixing problems detected by this check has to be done with great care because the intended behavior has to be carefully remodeled as Statecharts can include parts in which it is not clear what to do as depicted in Figure 3a (right-hand side). The rule *Transition-5* denotes that "Transitions outgoing pseudostates may not have a trigger" [19, Section 2.12.3.8]. The violation detection pattern may just contain a transition with the type of the source set to pseudostate and no trigger specified (see Figure 3b).



(a) Violations of Rule *CompositeState-1*.



(b) Violation of Rule *Transition-5*.

Fig. 3: Violations of *well-formedness Rules*.

**Syntactical Robustness Rules**

The style guide for Statecharts proposed in this paper aims at covering a wide range of dialects. Therefore, we extracted syntactical rules from various other style guides (cf. Section 2) that are applicable to different dialects. Furthermore, we formulated rules based on our own experience in Statechart modeling.

The rules presented below were adopted from Mutz [8, p. 144f]. All of them apply to the area of syntactical robustness and are dialect-independent.

*MiracleStates*: All states except the root state and the initial states must have at least one incoming transition. Figure 4a depicts the violation of this rule.

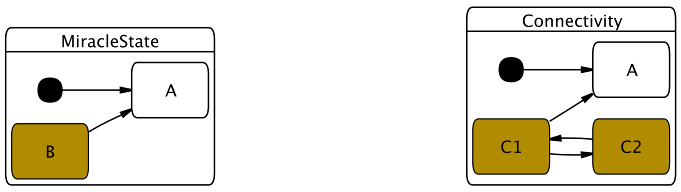
*IsolatedStates*: An even stronger version of *MiracleStates* is the check for isolated states. A state is isolated when it has neither incoming nor outgoing transitions.

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

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

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

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



(a) Violation of the Rule *MiracleStates*. (b) Violation of the Rule *Connectivity*.

Fig. 4: Violation of Syntactical Robustness Rules.

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

*DefaultFromJunction*: When using connective junctions to model decisions one shall always add an outgoing transition with no label. The unlabeled transition is then the default transition. The default transition is provided so the control flow does not stop when the other conditions do not hold.

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

*TransitionLabels*: Ensuring that all transitions are specified with a label makes the understanding of the model easier. This is especially important for dialects in which a default signal exists as it would be assigned invisibly to an unlabeled transition.

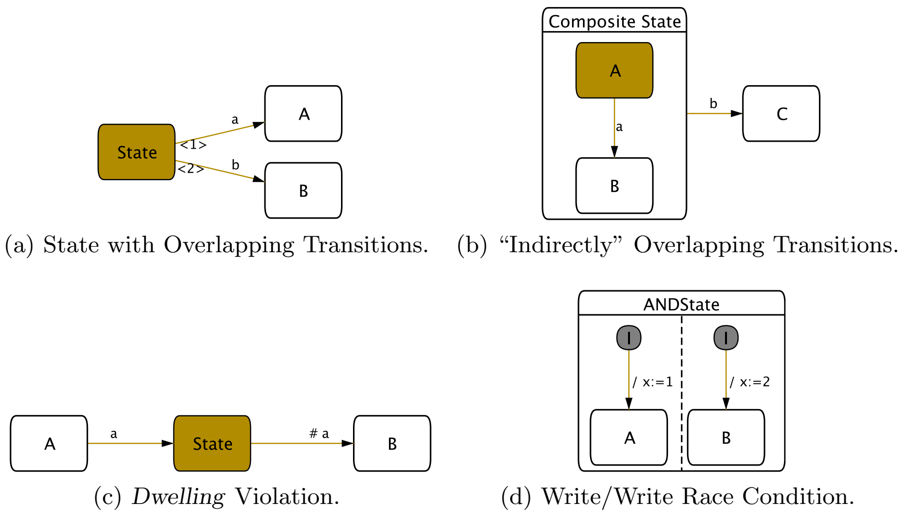
*InterlevelTransitions*: A Statechart should not contain interlevel transitions, *i. e.*, transitions bypassing level borders. The benefit is that understanding a Statechart without interlevel transitions is easier; especially novices tend to misunderstand the so expressed behavior, *e. g.*, the order of executed (entry) activities and the activation of parallel areas of execution.

*Connectivity*: Another aspect closely related to *MiracleStates* are states not connected by a sequence of transitions outgoing from any initial state. Such States are superfluous as they will never be entered while simulation. See Figure 4b where no path from the initial state to *C1* or *C2* exists. This rule extends the already mentioned *MiracleStates* as it also detects states that have incoming transitions and are still never entered as depicted in Figure 4b.

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

## Semantic Robustness Rules

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



(a) State with Overlapping Transitions. (b) "Indirectly" Overlapping Transitions.

(c) *Dwelling* Violation. (d) Write/Write Race Condition.

Fig. 5: Application Examples of the Semantic Robustness Rules.

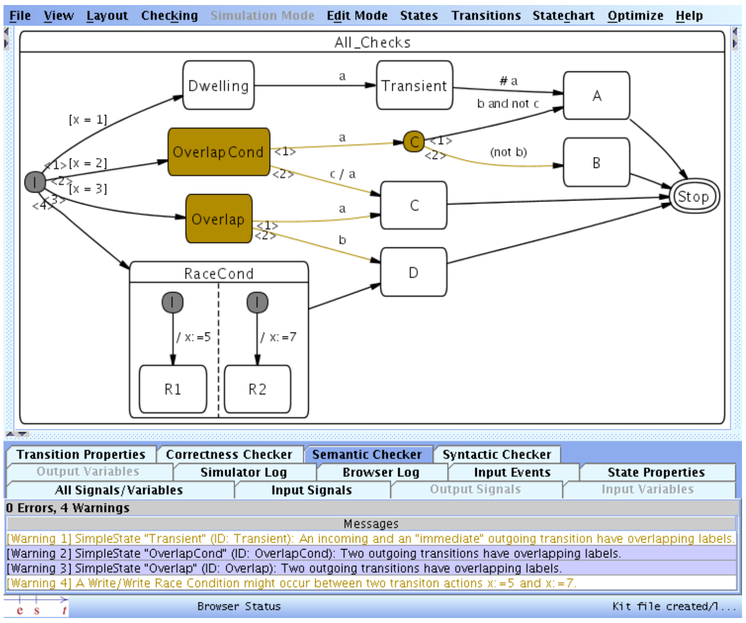
*Transition Overlap*: All transitions (directly or indirectly) outgoing from a state should have semantically disjoint predicates [20]. Ensuring this warrants that at most one transition is enabled at any time, *i. e.*, no transition shadowing can occur, leading to guaranteed deterministic behavior, independent of potential transition priorities. Figure 5a depicts a basic case of two transitions departing from a simple state. A *Transition Overlap* violation exists if $e_1$ and $c_1$ are not disjoint from $e_2$ and $c_2$. Such a violation may be eliminated by, *e. g.,* adding $\neg e_2$ and $\neg c_2$ to the predicates of $trans_1$, yielding $(e_1 \wedge \neg e_2)$ for the event expression and $(c_1 \wedge \neg c_2)$ for the condition expression. In addition to transitions departing directly from a state, transitions departing from an enclosing state may also be enabled (see Figure 5b). Overlaps are, however, resolved by transition priorities or hierarchy. Hence, this rule is primarily intended for Statechart dialects that do not provide a priority mechanism, such as *Statemate*.

*Dwelling*: The predicates of all incoming and outgoing transitions of a state should be pairwise disjoint or at least not completely overlapping [20]. This rule ensures that the system pauses at every state it reaches. A state in which the system cannot pause contradicts the concept of a *system state*. Careless use of *Esterel Studio*'s *immediate* flag, denoted by #, may lead to a *Dwelling* violation (see Figure 5c for an example). An immediate transition is evaluated in the same instant, in which its source state is reached; a non-immediate transition is not evaluated until the following instant.

*Race Conditions*: Concurrent writing or concurrent reading and writing of a variable should not exist in parallel states (cf. Figure 5d). Since race conditions are generally not detectable, we have chosen a conservative approximation. We detect a race condition in concurrent threads, if a variable is written in one thread and read or written in another. This rule, and the previous rule are aimed primarily at *Safe State Machine*s used in *Esterel Studio*.

## 5 The *KIEL* Modeling Environment

The *Kiel Integrated Environment for Layout* (*KIEL*) is a prototypical modeling environment that has been developed for the exploration of complex reactive system design [21]. As the name suggests, a central capability of *KIEL* is the automatic layout of graphical models. One can use *KIEL* to easily perform a layout of a given Statechart. However, the tool's main goal is to enhance the intuitive comprehension of the behavior of the System Under Development (SUD). While traditional Statechart development tools merely offer a static view of the SUD during simulation, in contrast, *KIEL* provides a simulation based on *dynamic focus-and-context* [21]. It employs a generic concept of Statecharts which can be adapted to specific notations and semantics, and it can import Statecharts that were created using other modeling tools. The currently supported dialects are those of *Esterel Studio*, *Stateflow*, and the UML via the XMI format, as, *e. g.,* generated by *ArgoUML*. *KIEL* further provides a structure-based editor to create Statecharts from scratch or to modify imported charts. A simulator is also part of the tool. The robustness checker, comprising checks for the rules presented above, has been integrated into *KIEL*. Figure 6 shows a screen-shot of *KIEL* as it checks particularly semantic robustness rules.



```
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.
```

Fig. 6: Screen-shot of *KIEL* Checking Robustness of a Statechart.

## The Checking Plug-in

The checking plug-in of *KIEL* was designed to be very flexible in usage. All checks have been implemented independently. Via an user interface it is easily possible to manually select which checks to apply. It is further possible to define Statechart dialect-specific profiles containing different sets of rules. Depending on the model loaded into *KIEL*, the plug-in automatically decides which profile to apply.

The plug-in was developed to be easily extendable. The user can extend the rule set by either adding an appropriate OCL constraint for a syntactical check or by adding a new Java class for semantical checks. Depending on the seriousness of a detected problem, the robustness checker delivers two kinds of messages. (1) *Errors* in modeling are violations of rules that have to be addressed because further actions such as simulating the model is impossible. (2) *Warnings* indicate that a problem was found which does not need to be fixed immediately for simulation, *i. e.*, possible sources of errors or ambiguous constructs. In the following, an overview of the implementation of the aforementioned rules is presented.

We have chosen the OCL because, as stated by Mutz, it allows to formulate checks on a high level of abstraction, and neither knowledge of a programming language nor of the underlying data structure is needed [8]. The executive approach towards the evaluation of OCL is preferable to an interpretative approach as the former one proved to be more flexible and faster in execution time.

Therefore, we chose to use the Dresden OCL Toolkit version 1.3[13] discussed by Richters [10] to transform OCL constraints to Java.

Our approach for the checking framework contains the possibility of returning customized messages when a violation is found. Therefore, the OCL constraint is wrapped by additional information as Java code snippets. The union of OCL and Java code snippets we named *KIEL wrapped OCL* (KOCL). The developed *KOCL* to Java translator utilizes the Dresden OCL Toolkit which is supplied with the according meta model of the *KIEL* data structure. Figure 7 basically shows how the different parts of the *KOCL* files are processed. The workflow and the specified rules were described in detail elsewhere [22].

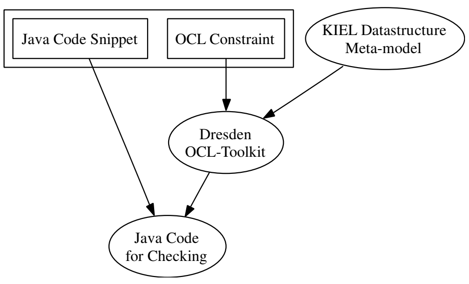


Fig. 7: Processing *KOCL* with *KIEL*.

As the framework is designed to handle rules formulated as OCL-constraints we have implemented the rules elaborated above (cf. Section 4). Most of the *well-formedness rule*s were specified in *KOCL*. The rules not specified in *KOCL* deal partly with features of UML diagrams. As the *KIEL* project so far is focused on simulating and modifying Statecharts only, the representations of classes and packages was left out for the sake of simplicity. Therefore, *e. g.*, rule StateMachine number 1 which states that "a State Machine is aggregated within either a classifier or a behavioral feature" from the UML specification was left out.

We will not present all of the transfered rules in detail. The example presented in the following gives an overview about how the additional information is capsuled within *KOCL* files. A relatively simple example is Rule *CompositeState-1* (cf. Section 4) as specified in Figure 8a. The OCL constraint states that the set `subvertex` of a composite state can contain at most one pseudostate of kind `#initial`. The Dot notation is used to access members of a class. An arrow ("`->`") is used to access properties or functions on sets.

The rule specified in *KOCL* is presented in Figure 8b. The separation of the message declaration, the constraint definition and the specification of the returning message is clearly seen in this example. The `declarations` part (lines 2–4) is designed to hold more than one message. The `fails` part (line 10) specifies

[13] `http://dresden-ocl.sourceforge.net/`

which message to return if a violation of the constraint is found. It is even possible to return different messages (if defined) depending on the context in the `fails` part by simply using a common `if-then-else`-statement. Due to the meta model the constraint itself (lines 7–9) is even shorter than specified in the UML.

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

(a) The OCL Representation.

```
 1  rule UML13CompositeStateRule1 {
 2    declarations {
 3      message "A composite state can have
 4            at most one initial vertex.";}
 5    constraint {
 6      context ORState or Region;
 7      "self.subnodes->select(
 8         v| v.oclIsTypeOf(InitialState))->
 9         size <= 1";}
10    fails {message;}
11  }
```

(b) The *KOCL* Representation.

Fig. 8: The Rule *CompositeState-1*.

As mentioned before, syntactic analysis is not the only field for which an automated checking framework for Statecharts is beneficial. The presented semantic rules can also be checked automatically. Although OCL is of great benefit in specifying and implementing robustness checks regarding the syntax of Statecharts, semantic analyses are generally beyond its scope because checking a Statechart with respect to these rules typically requires extensive knowledge of the model semantics. The *Transition Overlap* rule, the *Dwelling* rule, and the *Race Conditions* rule (see Section 4) cannot be specified using OCL constraints. Our framework still allows to incorporate such checks; for this purpose Java code is needed to formulate theorem-proving queries and sending them to an outside tool for analysis.

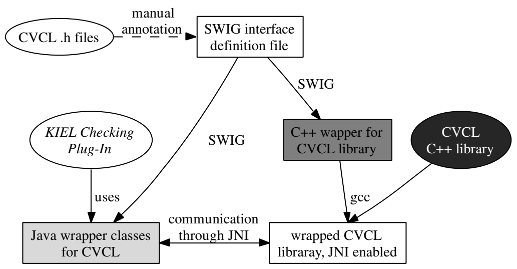
To perform the semantic robustness checks, a *satisfiability modulo theories* (SMT)[14] solver is needed. SMT problems are a variation of *automated theorem proving* [23], which in turn is part of automated reasoning. After an evaluation of available SMT solvers [24], CVC Lite [25] was chosen. Here, in order to determine whether, *e. g.*, two transitions $trans_1$ and $trans_2$ (cf. Section 4) have overlapping labels, satisfiability of the formula

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

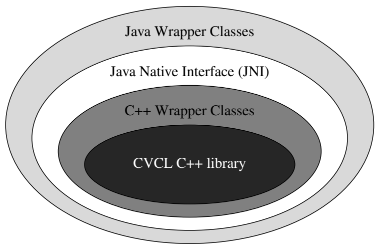
must be decided. Unsatisfiability implies that the predicates of $trans_1$ and $trans_2$ are disjoint. Such SMT problems are generally decidable as long as they contain only addition but no multiplication of variables.

[14] `http://combination.cs.uiowa.edu/smtlib/`

Further, the *Simplified Wrapper and Interface Generator* (*SWIG*) [26] was employed to generate wrappers and interface files for CVC Lite, enabling its immediate use from within Java. Here, the Java and C++ JNI wrappers are produced from CVC Lite's annotated C++ header files, as shown in Figure 9.


CVCL .h 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.

Fig. 9: Interfacing of *KIEL* and the CVC Lite Library via JNI and SWIG.

## 6 Experimental Results

Finally, we show the application of the checking framework on a well known example, the wristwatch presented by Harel [1]. As this example is well-established we did not expect to detect real modeling errors; our focus was to quantitatively asses the efficiency of our checking mechanism. We remodeled the wristwatch with ArgoUML which imposed some restrictions. *E. g.*, some transitions perform indexing over multiple states, which was replaced by according conditional constructs. However, the final model retains the originally modeled behavior. So far, the final model contains 120 transitions and 108 states.

The results from benchmarking are presented in Table 1. The number of returned hints and the run-time of each check are presented. The checking times were measured on a PC with GNU/Linux OS, a 2.6 GHz AMD Athlon 64 processor and 2 GB of RAM.

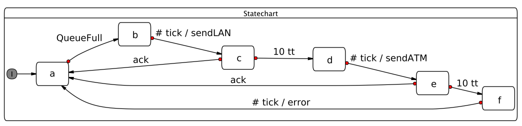
The application of the *well-formedness rule*s consumed the least time of all parts. Roughly 20 milli-seconds were needed to check those rules on the chart. Except for the check *EqualNames* the syntactical robustness checks roughly take twice as much time as the *well-formedness rule*s. The check *EqualNames* has a quadratic complexity in the number of states. This is caused by limitations of the OCL—all states have to be compared to the currently handled state. In comparison to the checks dealing with syntactical robustness, except *EqualNames*, the checks for semantical robustness take about 400 milli-seconds. Here, the check

Table 1: Experimental Results of Checking the Wristwatch Example.

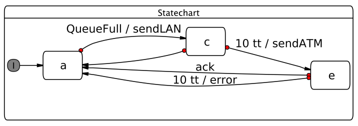
| **Checks** | **Hints** | **Time [ms]** |
|---|---:|---:|
| **well-formedness checks (total)** | **0** | **20** |
| *InterlevelTransition* | 17 | 14 |
| *Connectivity* | 7 | 2 |
| *EqualNames* | 33 | 587 |
| *InitialStateCount* | 7 | 1 |
| *TransitionLabels* | 6 | 9 |
| *IsolatedStates* | 1 | 4 |
| **syntactical checks (total)** | **71** | **617** |
| *Transition Overlap* | 598 | 352 |
| *Dwelling* | 0 | 2 |
| *Race Conditions* | 0 | 1 |
| **semantical checks (total)** | **598** | **355** |
| **total** | **669** | **992** |

*Transition Overlap* returns an enormous number of hints compared to the total number of transitions. This is due to the fact that almost no transition was designed with an opposing predicate of another outgoing transition.

As another example, the application of the framework on the Statechart presented in Figure 10a delivered the hint that violations of the rule *Dwelling* are present. Especially novices tend to produce unnecessarily large models with needless states, for example by splitting trigger and effect into separate transitions. Figure 10b shows a possible way how the violation can be fixed. Because the Statechart is rather small, all checks were applied in about 3 milli-seconds.



(a) A Statechart with Unnecessary, Transient States.



(b) Statechart after Removing the Transient States `b`, `d`, and `f`.

Fig. 10: Example for Removing Transient States.

## 7 Assessment

We gained the following results during the work with the framework. Not surprisingly, the time needed for specifying rules differs significantly depending on the complexity of the problem. The fairly simple *well-formedness rule*s from the UML were specified in *KOCL* in a rather short amount of time. All in all, it took less than one hour to specify them. The more sophisticated rules regarding problems from the field of syntactical robustness took not much longer, as the OCL proved to be an easy to apply language for these kind of problems, too. The time needed for specifying those rules varies between two minutes and half an hour per rule. The semantic robustness rules turned out to be the most demanding. Roughly two weeks were needed altogether for the implementation of the three rules presented. The main aspect of this task was to extract the needed data and afterwards to transform the data from the Statechart to the input language of the theorem prover.

The *well-formedness rule*s do not necessarily improve the quality of Statecharts in the sense of robustness. Those rules apply to the field of syntactical correctness only. Nevertheless, these rules are needed before any further checking can be applied to a Statechart, because the robustness checks rely on the correct syntax.

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

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

Finally, there is a trade-off between semantic robustness and minimality of Statecharts. *E. g.*, eliminating a *Transition Overlap* or *Dwelling* violation by adding the negation of the predicates of one transition to the predicates of the other transition, as suggested above, constitutes an infringement of the *write things once* principle of modeling [27].

In summary, on the one hand one can say that evaluating OCL statements as specified by the *well-formedness rule*s turned out to be a task very fast done. On the other hand one has to say that for statements of greater complexity the evaluation of OCL—as in rule *EqualNames*—can be much more time consuming.

## 8 Conclusion and Further Work

As the failure of safety-critical systems can have severe consequences, error prevention in the model-driven system development of such systems is vital. We have outlined an approach to make the model-driven system development with Statecharts less error prone, and have presented a general Statechart style guide that is not restricted to a single dialect. We implemented a flexible robustness checking framework within the *KIEL* modeling tool. The hints returned by our checking framework do not necessarily indicate errors; this typically still requires application-specific knowledge. However, as has been observed in earlier work, adhering to the robustness rules reduces the chance for errors. Beyond that, they serve to improve the readability and maintainability of a system.

Our framework permits to express robustness rules directly with the well-established OCL formalism, which facilitates an abstract rule formulation and allows to directly incorporate existing OCL rule sets. Our transformative approach for the evaluation of OCL statements has turned out superior to earlier, interpretative approaches, and the expressiveness of the OCL has been sufficient for most of our checks. However, the framework also allows to implement complex semantic checks in Java directly, which we have used to incorporate an off-the-shelf theorem prover. Our framework has been practically validated for the checking of UML State Machines; however, the framework could easily be adapted to other commercial Statechart modeling tools as well; provided that an appropriate import functionality exists.

Beyond the experimental results presented in this paper, we intend to utilize the *KIEL* checking framework to perform a systematic study of the effectiveness of robustness checking, both for novice users and experienced modelers. Furthermore, we plan to implement support for the recently published version 2.0 of the OCL, and to incorporate further rules into our checking framework.

## References

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

[2] Object Management Group: Unified Modeling Language: Superstructure, Version 2.0 (2005)

[3] Parnas, D.L.: Some theorems we should prove. In: HUG '93: Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications, London, UK, Springer-Verlag (1994) 155–162

[4] Object Management Group: (Unified Modeling Lanugage—UML Resource Page) `http://www.uml.org`.

[5] 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)

[6] Scaife, N., Sofronis, C., Caspi, P., Tripakis, S., Maraninchi, F.: Defining and translating a "safe" subset of simulink/stateflow into lustre. Technical Report 2004-16, Verimag, Centre Équation, 38610 Gières (2004)

[7] Kreppold, T.: 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 (2001)

[8] Mutz, M.: Eine durchgängige modellbasierte Entwurfsmethodik für eingebettete Systeme im Automobilbereich. Dissertation, Technische Universität Braunschweig (2005)

[9] Pap, Z., Majzik, I., Pataricza, A.: Checking general safety criteria on UML statecharts. Lecture Notes in Computer Science **2187** (2001)

[10] Richters, M.: A Precise Approach to Validating UML Models and OCL Constraints. PhD thesis, University of Bremen (2001)

[11] Mutz, M., Huhn, M.: Automated statechart analysis for user-defined design rules. Technical report, Technische Universität Braunschweig (2003)

[12] Moutos, M., Korn, A., Fisel, C.: Guideline-Checker. Studienarbeit, University of Applied Sciences in Esslingen (2000)

[13] Scheidler, C.: Systems Engineering for Time Triggered Architectures. SETTA Consortium (2002) Deliverable D7.3 – Final Document.

[14] Heitmeyer, C., Jeffords, R., Labaw, B.: Automated Consistency Checking of Requirements Specifications. ACM Transactions on Software Engineering and Methodology **5**(3) (1996) 231–261

[15] Heimdahl, M.P.E., Leveson, N.G.: Completeness and Consistency in Hierarchical State-Based Requirements. Software Engineering **22**(6) (1996) 363–377

[16] Johnson, S.C.: Lint, a C program checker. In Thompson, K., Ritchie, D.M., eds.: UNIX Programmer's Manual. Seventh edn. Bell Laboratories (1979)

[17] Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. IEEE Software **19**(1) (2002) 42–51

[18] Buck, D., Rau, A.: On Modelling Guidelines: Flowchart Patterns for STATEFLOW. Softwaretechnik-Trends **21**(2) (2001) 7–12

[19] Object Management Group: Unified Modeling Language (UML) 1.3 specification (2000) `http://www.omg.org/cgi-bin/apps/doc?formal/00-03-01.pdf`.

[20] Kossowan, K.: Automatisierte überprüfung semantischer modellierungsrichtlinien für statecharts. Diplomarbeit, Technische Universität Berlin (2000)

[21] Prochnow, S., von Hanxleden, R.: Comfortable Modeling of Complex Reactive Systems. In: Proceedings of Design, Automation and Test in Europe (DATE'06), Munich (2006)

[22] Bell, K.: Überprüfung der Syntaktischen Robustheit von Statecharts auf der Basis von OCL. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik (2006) unpublished.

[23] Gallier, J.H.: Logic for Computer Science: Foundations of Automatic Theorem Proving. Revised On-Line Version (2003), Philadelphia, PA (2003)

[24] Schaefer, G.: Statechart Style Checking – Automated Semantic Robustness Analysis of Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Institut für Informatik (2006)

[25] Barrett, C.W., Berezin, S.: CVC Lite: A new implementation of the Cooperating Validity Checker Category B. In Alur, R., Peled, D.A., eds.: Proceedings of Computer Aided Verification: 16th International Conference, CAV 2004, Boston. Volume 3114 of Lecture Notes in Computer Science., Springer (2004) 515–518

[26] Beazley, D.M.: SWIG: An easy to use tool for integrating scripting languages with C and C++. In: Proceedings of the Fourth Annual USENIX Tcl/Tk Workshop. (1996) 129–139

[27] Berry, G.: The Foundations of Esterel. Proof, Language and Interaction: Essays in Honour of Robin Milner (2000) Editors: G. Plotkin, C. Stirling and M. Tofte.