Model Checking for SCCharts

Andreas Achim Stange

Master's Thesis
May 2019

Real-Time and Embedded Systems Group
Prof. Dr. Reinhard von Hanxleden
Department of Computer Science
Kiel University

Advised by
M.Sc. Alexander Schulz-Rosengarten
Dipl.-Inf. Steven Smyth

Selbstständigkeitserklärung

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

Kiel,

Abstract

Correct and predictable behavior is mandatory for safety-critical applications and model checking can verify important properties of these systems. Synchronous languages with mathematically formalized semantics are particularly suited for model checking.

This thesis discusses means of model checking the synchronous language SCCharts. Characteristics of reactive models are considered and how temporal properties can change in a translation to another language. The model checkers SPIN and nuXmv have been integrated in the KIELER development environment using translations from SCCharts to the imperative language PROMELA and to the data-flow language SMV. Thereby, the translations utilize a low-level representation of SCCharts, namely the Sequentially Constructive Graph.

This work shares observations and experiences with the model checkers and illustrates characteristics of models that can affect the model checking performance. Further, an evaluation of the implementation on small examples demonstrates potential for practical use. There are models that SPIN could verify faster. However, the symbolic approach implemented in nuXmv performed better than SPIN in many cases. In particular, the IC3 algorithm, K-liveness, and interpolation-based model checking of nuXmv performed well.

Acknowledgements

I want to thank my advisers Alexander Schulz-Rosengarten and Steven Smyth as well as Professor Dr. Reinhard von Hanxleden for the time they invested to countercheck and guide this thesis. Moreover, I want to thank Dr. Eugene Yip and Professor Dr. Gerald Lüttgen from University of Bamberg for their support of this thesis and hospitality. In particular, I want to thank for helping with the translation to SMV, and with adapting temporal properties.

A special thanks is directed at all members of the Real-time and Embedded Stems Group of Kiel University for the cooperative, fun and professional atmosphere that I experienced during my time there.

Contents

1 Introduction **1**
1.1 SCCharts . . . 2
1.2 Model Checking . . . 3
1.3 Problem Statement . . . 4
1.4 Outline . . . 4

2 Used Technology **5**
2.1 KIELER . . . 5
2.1.1 SCCharts . . . 5
2.1.2 SCG . . . 7
2.1.3 KTrace . . . 9
2.2 Model Checking . . . 9
2.2.1 Temporal Logics . . . 10
2.2.2 Synchronous Observer . . . 10
2.2.3 SPIN . . . 11
2.2.4 nuXmv . . . 12

3 Related Work **15**
3.1 SCADE . . . 16
3.2 mbeddr . . . 17
3.3 Lesar . . . 17
3.4 Xeve . . . 18

4 Concept **19**
4.1 Model Checker Selection . . . 19
4.2 Translation Considerations . . . 20
4.2.1 Persisting Temporal Properties . . . 21
4.3 Translation to SMV . . . 22
4.3.1 SMV Constructs . . . 22
4.3.2 Modeling the Tick Logic via Program Counter . . . 23
4.3.3 Modeling the Tick Logic via SSA . . . 24
4.4 Translation to PROMELA . . . 27
4.4.1 Modeling the Tick Logic . . . 27
4.4.2 Modeling Reactive Systems . . . 29
4.4.3 Specifying Properties . . . 29
4.5 Synchronous Observers in SCCharts . . . 31

5 Implementation **33**
5.1 Plug-in Overview . . . 33
5.2 Common Data Structures . . . 34
5.3 Translation to PROMELA . . . 35
5.4 Translation to SMV . . . 36

5.5 Interfacing with the Model Checkers . . . 37
5.5.1 Parsing Counterexamples . . . 39
5.5.2 Generated Files . . . 40
5.6 Automated Tests . . . 41
5.7 Graphical User Interface . . . 42

6 Evaluation and Experience Report **43**
6.1 General Observations . . . 43
6.1.1 Model Checker Integration . . . 44
6.1.2 Model Characteristics that Affect Performance . . . 45
6.2 Comparison of Algorithms . . . 46
6.2.1 Tested Models . . . 46
6.2.2 Test Setup . . . 51
6.2.3 Results . . . 52
6.2.4 Comparison with Lesar . . . 55

7 Conclusion **57**
7.1 Summary . . . 57
7.2 Possible Applications . . . 58
7.2.1 Teaching Temporal Logics . . . 58
7.2.2 Testing of SCCharts Compiler . . . 59
7.3 Future Work . . . 59
7.3.1 Extending the Translation to SMV and PROMELA . . . 59
7.3.2 Further Model Checkers . . . 60
7.3.3 Further Synchronous Languages . . . 60
7.3.4 Visualization of State-Space . . . 60
7.3.5 Evaluation in Case Study . . . 60

Bibliography **61**

List of Acronyms **65**

List of Listings **67**

List of Figures **69**

List of Tables **71**

Chapter 1

Introduction

Characteristic for reactive systems is an interdependence with their surrounding environment. Sensors deliver inputs to the system, whereas calculated outputs control actuators that in turn influence the environment. Such systems can be found at the heart of safety-critical applications, for example in form of a controlling unit. It becomes obvious that correctness and reliability of such systems are of special importance when imagining the possible impact of a software error in a plane or nuclear power plant. This motivates the use of formal methods for specifying and verifying safety-critical applications.

Model checking is a solution for formally verifying system properties that can be automated and gives a counterexample in case a property is violated, which is useful for debugging [BK08]. To verify that a property holds, it is checked in all reachable states of the system. Limitations of model checking are imposed by large state-spaces in real-world applications (state-space explosion problem). However, using optimized models and efficient algorithms the technique can already be used in many applications or at least central parts of these. For example, model checking could have revealed bugs that are today famous for their consequences, e.g. in the Ariane-5 missile, the Mars Pathfinder and Intel's Pentium II processor [BK08]. There is still ongoing research to improve the performance and scope of application area for model checking.

Another tool to tackle the requirements of reactive systems development can be found in the synchronous approach as pointed out by Halbwachs [Hal93]. He argues that classic approaches for concurrency, such as threads, introduce an interleaving of execution paths that makes it difficult to reason about system behavior. Low-level models of programs, e. g., Petri-nets and basic deterministic automata, do not scale well because they lack means to abstract from concurrency or complex subsystems. On the other hand general-purpose parallel languages, such as Ada or Occam, provide high-level features that abstract away process communication internals. However, Halbwachs pointed out that their semantics are often vague in favor of portability. Berry has given an example for the vague semantics of Ada when trying to broadcast an event MINUTE after 60 seconds [Ber89]. He illustrates that the event will not be received by all listeners at the same instant, which contradicts the event.

In contrast to this, the synchronous approach relies on fully deterministic semantics and the idealized assumption that a reaction of the system happens in zero-delay. This is the *synchrony hypothesis*. A single such reaction is often referred to as a *tick* or *instant*. An upper bound for its execution time can be given because the system reaction is deterministic. This makes the implementation of the synchrony hypothesis feasible for a given hardware platform [Ray08]. The concepts of the synchronous approach are implemented in synchronous languages. Their formal and deterministic semantics make them well suited for model checking.

Therefore synchronous languages and model checking are used in industry for developing safety-critical applications [BCE+03]. For instance the synchronous language Lustre is the semantic root

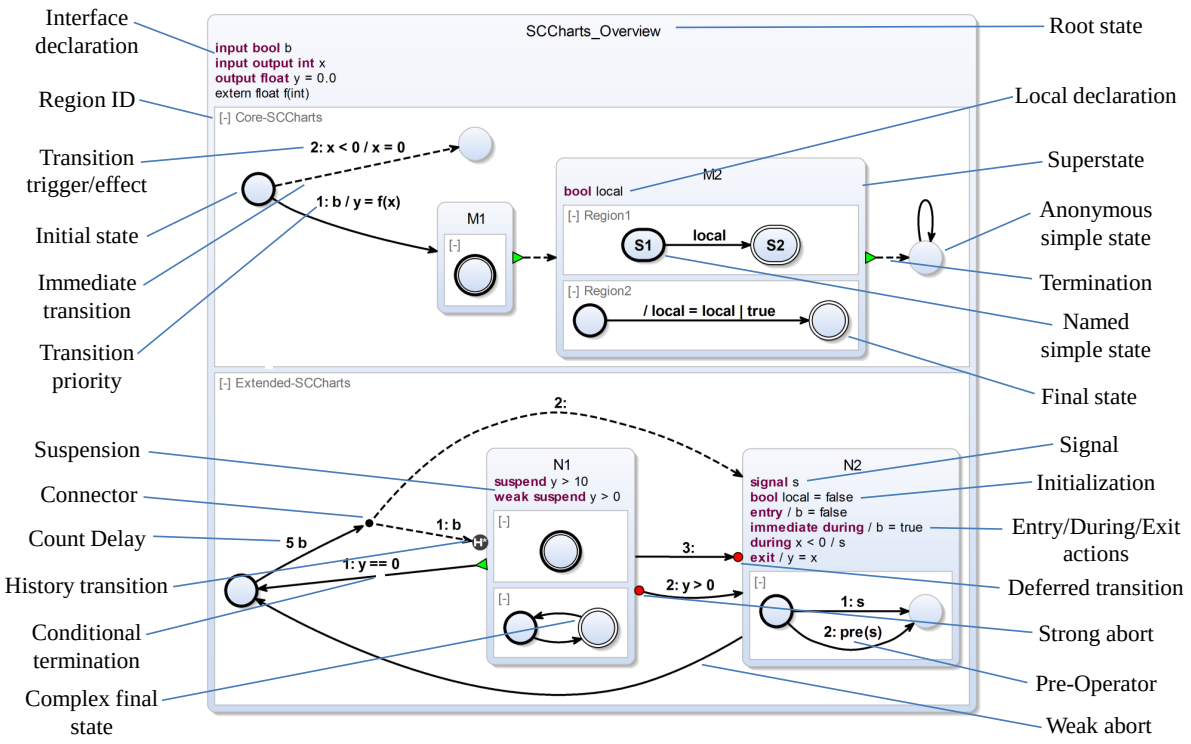


Figure 1.1. Syntax overview of SCCharts [HDM+14].

of SCADE[1], a development environment for safety-critical applications. SCADE supports automated verification of its models and is used in various industries such as avionics and railways.

1.1 SCCharts

Sequentially Constructive StateCharts (SCCharts) is a synchronous language with graphical syntax [HDM+14]. Syntactically SCCharts is based on SyncCharts [And95], in which program logic is expressed as a state-machine extended with constructs for, among others, concurrency, hierarchy and preemption. Figure 1.1 shows the syntactical elements of the language.

The semantics of SCCharts are based on the Sequentially Constructive Model of Computation (SC MoC) [HMA+14]. In this model concurrent accesses to variables are handled by scheduling them deterministically, such that writing of variables is done before reading them. Sequentially ordered accesses to variables on the other hand are scheduled as given in the program. This approach is less restrictive than classical Models of Computation for synchronous languages where a variable is only allowed to have one single value within a tick.

[1] www.esterel-technologies.com/products/scade-suite/

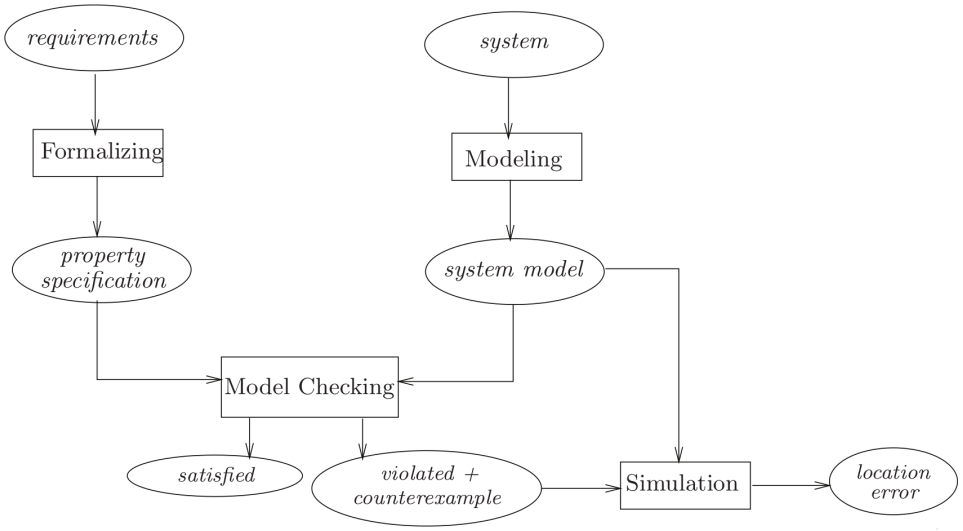


Figure 1.2. Schematic of model checking [BK08]. A model of the system to be verified together with the property to be checked are given as input to a model checker, which will output a counterexample or that the property holds in the system. The counterexample can be used to debug the system.

KIELER[2] is an IDE developed at Kiel University that implements SCCharts along with pragmatic modeling tools, for instance automatic layout and quick navigation of graphs. In KIELER an SCChart is written in a textual syntax and the corresponding visual representation is synthesized and laid out automatically. This provides the benefits of a visually understandable model without sacrificing efficient and common tools for coding [FH10].

Several approaches to compile SCCharts have been suggested [HDM+14] [Pei17] [SMH18]. KIELER implements these in a compiler framework that uses an incremental compilation approach and model-to-model transformations.

1.2 Model Checking

Model checking is a formal method used in practice to show that a property holds in a system model. For this, the system model and the property are specified and given as input to a model checker. A naive model checking solution could create the complete state space of the system model and check properties via exhaustive search. When a property is found to be violated, the traversed states can be output as counterexample of the property, so that a developer can retrace the cause and fix the broken system. Figure 1.2 illustrates this.

Simple properties that are independent of other states require traversing the system once. More complex properties, e. g., that there are no deadlocks in the system, requires finding a loop in the system, such that it holds in all states of the loop and thus infinitely often. There is a whole theory of the complexity and languages to formulate properties. In practice the set of *safety* properties (*"something bad will never happen"*) and *liveness* properties (*"something good will eventually happen"*) are the most important.

For formal verification it is necessary that the property to be checked is formulated without ambiguity. Temporal logics have been found to be suited for this task. Temporal logics extend

[2] http://rtsys.informatik.uni-kiel.de/kieler

traditional propositional logics with operators to refer to different states of the system in time. For example, one can imagine operators that refer to the next, the last or a future tick of a reactive system. Well known temporal logics are Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). Although these describe different sets of properties in general, their intersection is in many cases sufficient in practice.

The naive approach to model checking quickly suffers from the state space explosion problem, as the state space grows exponentially. Therefore different algorithms and abstraction techniques have been developed and extending the application area of model checking is still on-going research. Systems with more than 10^{20} states have been checked successfully using smart model checking techniques [BK08].

1.3 Problem Statement

The goal of this thesis is to find ways for model checking SCCharts that integrate with the KIELER tooling. This can be achieved by re-using existing model checkers or by a custom model checker implementation for SCCharts. Further, different abstraction levels of SCCharts in the KIELER compile chain can be re-used. This thesis discusses these different approaches. For a translation to existing model checker languages, it is necessary to identify characteristics of the synchronous MoC and how they can be preserved.

The new tools have to be integrated in the KIELER tooling to enable a complete model checking work-flow. As part of this, counterexamples from a model checker have to be refined for the original high-level SCChart such that they can be used for debugging.

1.4 Outline

In Chapter 2 the technology used in this thesis is explained. Related work is presented in Chapter 3. This includes papers that compare model checkers in case studies as well as other projects that use model checking in a similar application area. Chapter 4 explains concepts for model checking a synchronous language on an abstract level. Requirements for model checking a synchronous language in general and SCCharts in particular are discussed. Afterwards the model checkers nuXmv and SPIN that fit the requirements are identified so that a broad range of model checking techniques and input languages can be evaluated for this task. Chapter 5 goes into details of the implementation in the KIELER tool. In Chapter 6 the implementation is evaluated. First, general observations of the presented translation and its limits when model checking SCCharts are discussed. Second, different options and algorithms provided by the chosen model checkers are evaluated in a systematic comparison.

Finally, Chapter 7 summarizes the thesis and presents possible future work.

Chapter 2

Used Technology

The following presents technologies and concepts used in this thesis. First, KIELER and related tools are introduced. Afterwards, temporal logics and the synchronous observer pattern are explained, which can be used to specify properties for model checking. Finally, the model checkers nuXmv and SPIN are introduced.

2.1 KIELER

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) is an open source IDE developed at Kiel University. It is based on Eclipse[1], which was originally started as a Java IDE but developed to a general purpose rich client platform. An extensible plug-in mechanism in Eclipse enables to implement large, modular applications within this framework. For instance, KIELER contributes several plug-ins containing its compiler framework and SCCharts related tools. Furthermore, the Eclipse Public License[2] supports a wide application area for academic and commercial use. As a result, the Eclipse platform has thrived to an ecosystem with many open source projects from academia and industry.

The KIELER project builds on these technologies, among other things, an implementation of SCCharts. Thereby, heavy use is made of the Eclipse Modeling Framework (EMF)[3] and Xtext[4] to develop metamodels and language tooling. For instance, parsers, editors, and static code analyzers are created from an Xtext grammar of the textual SCCharts language.

Furthermore, KIELER implements an extensible framework for interactive compilation, namely the KIELER Compiler (KiCo) [SSH18]. Compilation and simulation of SCCharts have been developed using this framework.

A focus of the KIELER project is to enhance the pragmatics of model-based design by using automatic layout and filtering mechanisms [FH10]. These enable graphical modeling from a textual syntax, thus combining benefits of both approaches. KIELER SCCharts is a demonstration of this combined approach.

2.1.1 SCCharts

An SCChart is at its core a state-machine similar to a Mealy-machine extended with hierarchy and concurrency. Figure 2.1 shows a simple SCChart, namely the *ABRO* model. States are linked through transitions that can have a *trigger* and an *effect*, which are separated by a slash. The transition can be taken only when the trigger condition evaluates to true. When the transition is taken then the effect is executed before entering the next state. Concurrency is added using regions. Every region has a state-machine on its own that starts in one initial state. In the ABRO example, the state `WaitAB` has two regions, each waiting for one input.

[1] `https://eclipse.org`
[2] `https://www.eclipse.org/legal/epl-2.0/`
[3] `https://www.eclipse.org/modeling/emf`
[4] `https://www.eclipse.org/Xtext`

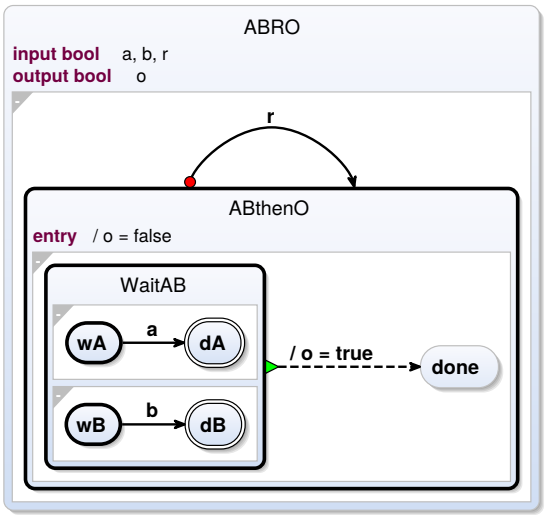


Figure 2.1. Example for an SCChart. The ABRO model waits concurrently for the inputs `a` and `b` and sets the output `o` to true when both have been received. Further, the input `r` is used to reset the system, which takes precedence in this model.

There are two kinds of transitions namely *delayed* and *immediate* transitions. Immediate transitions conceptually do not take time in the synchronous MoC and can always be taken. In contrast to this, a delayed transition can only be taken when the control-flow of the previous reaction has ended in its source state. Thus, delayed transitions separate one system reaction from the next. The ABRO model uses delayed transitions in `WaitAB`, such that these will not react to inputs in the initial tick.

A state in SCCharts can contain further regions with one initial and multiple final states. Such a state with inner behavior is called a *superstate*. A superstate is left trough a *termination* transition when all of its regions have reached a final state. In the ABRO model, the termination transition that sets the output to true is taken when the state `WaitAB` has reached the final states `dA` and `dB`, i. e., when the inputs `a` and `b` have been received.

Furthermore an SCChart contains declarations that define which variables are used and which of them are inputs or outputs to the model.

Most other language constructs of SCCharts can be expressed in terms of the ones described above. Figure 1.1 gives an overview of SCCharts elements. One can differentiate *extended* SCCharts that can be simplified from *core* SCCharts that cannot be expressed in terms of simpler features.

The ABRO model contains extended features in form of an entry action that is used to reset the output variable `o` when entering the state `ABthenO`. Other actions that are available in SCCharts are *exit* actions and *during* actions. These are evaluated when a state is left, respectively as long as control-flow stays in a state. Furthermore, in ABRO a *strong abort* is used to reset the model. Inner behavior of the `ABthenO` superstate will be preempted when the strong abort transition is taken. A *weak abort* on the other hand would allow the execution of inner behavior of a superstate before moving control-flow to a new position.

In the synchronous MoC a single reaction happens in zero-time. For SCCharts this means that any change to a variable is broadcast to all concurrent regions and immediately visible. This is implemented in SCCharts semantics by scheduling rules for concurrent variable accesses, namely the Initialize-Update-Read (IUR) protocol. This means that a concurrent *initialization* of a variable (i. e., setting a value that does not depend on the current variable value) is done before any concurrent *updates* to a variable,

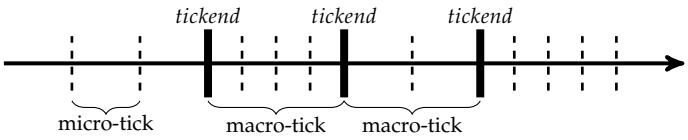


Figure 2.2. A timeline of reactions. A bold vertical line corresponds to the end of a macro-tick. Such a computation can be split into multiple micro-ticks depending on the abstraction level, illustrated here as smaller dashed vertical lines.

which in turn are done before concurrently *reading* a variable. SCCharts that cannot be scheduled consistently using these rules are rejected by the compiler.

In addition to the explained features, there are referencing mechanisms to instantiate re-usable modules. In KIELER these are implemented similar to a macro expansion, i. e., by putting concrete SCCharts instances in place of parameterized references. Furthermore, experimental data-flow regions allow the definition of data-flow equations with sequentially constructive semantics. In a low-level model each equation corresponds to a concurrent control-flow region that performs its corresponding assignment every tick. As a result, dependencies between equations are handled by the IUR-protocol when scheduling the concurrent regions. Other language constructs of SCCharts are introduced in this thesis when needed.

A single discrete reaction of a synchronous model is called a *macro-tick*. In reality the conceptually instantaneous macro-tick is implemented by a number of smaller elementary actions, which are called *micro-ticks* or *microsteps*. A timeline of reactions with macro-ticks and micro-ticks is illustrated in Figure 2.2. What is considered a micro-tick depends on the current point of view. On a hardware-level, the propagation of a signal through a logic gate can be considered a micro-tick. Other possible abstraction levels to look at micro-ticks are the nodes of an SCG, or lines of code in the textual SCCharts model. However, on the high-level synchronous MoC, only macro-ticks are considered as observable. As a result, temporal properties for SCCharts are formulated on macro-tick level.

The implementation of SCCharts in KIELER is using a textual syntax to create models. In this textual language it is possible to add meta-information in form of annotations to many elements, for instance states, variables and transitions. This is a general mechanism to augment a model with information that does not affect its semantics, but is relevant for the tooling. For example, this mechanism can be used in KIELER to add information about how the synthesized diagram should be laid out.

KIELER includes different interactive compile chains for SCCharts that include data-flow-based, priority-based, and state-based compilation approaches [HDM+14] [Pei17] [**smyth2018synthesizing**]. Thereby, compilation is done incrementally. For example, there are semantics-preserving model-to-model transformations to reduce complex SCCharts features to simpler ones. After these transformations have been applied and the model contains only core language features of SCCharts, it can be further processed and compiled more easily to other languages such as C or VHDL.

2.1.2 SCG

The Sequentially Constructive Graph (SCG) is a control-flow based data structure to express sequentially constructive programs. As a result, it can be seen as low-level representation of SCCharts. An SCG is written in a textual syntax in KIELER, namely the Sequentially Constructive Language (SCL)[HMA+14]. Figure 2.3 illustrates the syntax of an SCG and its semantic equivalence to SCL and other data-structures.

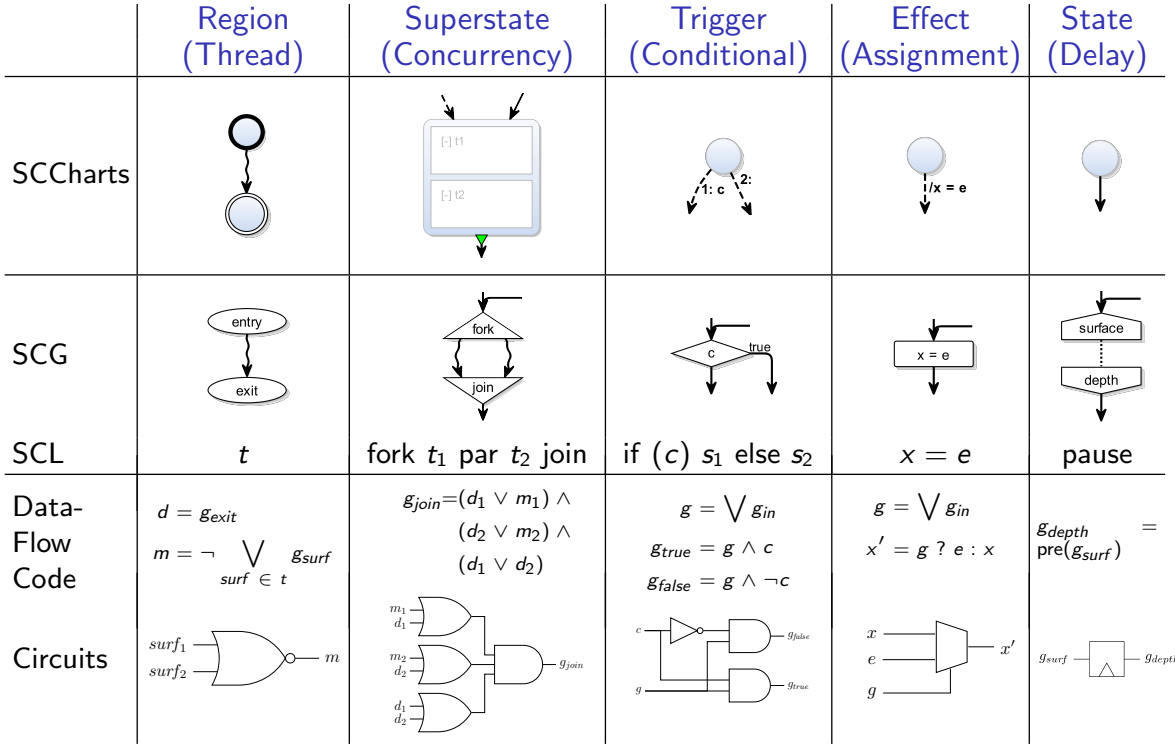


Figure 2.3. Transformation matrix of normalized SCCharts, SCG, data-flow and circuits [HDM+14].

The main nodes of an SCG are entry, exit, fork and join, surface, depth, conditional, and assignment. Each concurrent thread has an entry-node and an exit-node in which the control-flow starts, respectively terminates. A fork splits the control-flow into multiple threads that are conceptually executed at the same time. When all forked threads have reached their exit-node, control flow continues in the corresponding join-node. Surface and depth represent a tick border. This means that when the surface-node of a thread is reached then the control-flow will continue in the corresponding depth-node in the next tick. Conditional-nodes have two outgoing control-flow edges. One is used when the condition evaluates to true, whereas the other is used when it evaluates to false. Assignment-nodes set the value of variables.

Another SCG construct is the pre-operator, which can be used on the right side of an assignment. Its operand is a variable. The pre-operator refers to the variable's value at the end of the previous tick. In the initial tick, the returned value is `false` for Booleans and `0` for integers. This operator is typically implemented by an additional register or variable that stores the operand's value at the end of a tick to make it accessible in the following tick. Such a variable that implements a pre-operator is called *pre-variable*.

Normalized SCCharts can be mapped directly to SCG constructs, which can be expressed in data-flow and hardware circuits. The variables that are introduced by a translation to data-flow are called *guards*.

```
1 a=true => o=1;
2 => o=2;
3 loop_start:
4 a=false => o=2
5 goto loop_start;
```

Listing 2.1. A small KTrace example. In the first tick, the input `a` is `true` and the output `o` is `1`. In the second tick, `a` stays `true` and `o` is `2`. The third tick has a label, which is used in line 5 as jump target. In this tick, `a` is false and `o` stays `2`. The following `goto` indicates that the trace should be continued with the tick that has the corresponding label, namely the third tick. In this trace, the visible state of the system does not change anymore in following ticks.

They encode which parts of the SCG are executed in a reaction. For instance, consider the data-flow row from Figure 2.3 for a Conditional. The guard g_{true} determines when the true-branch of the conditional is executed, whereas g_{false} determines when the false-branch of the conditional is executed. The conditional itself is executed when any of its incoming transitions are taken. This is expressed by a disjunction over the guards that represent the transitions.

A special input to the SCG in data-flow form is the GO-signal. It marks the entry point to the tick logic, i. e., the initial reaction. Therefore this variable must be set to `true` in the first tick and to `false` in all following ticks. More elements are needed to create a complete SCG that can be compiled, e. g., input and output declarations. However, these elements are of little relevance for understanding its behavior and are thus not further explained here.

The netlist-based compilation approach implemented in KIELER maps SCG constructs to their data-flow equivalent. Semantics of sequential constructiveness are then applied to schedule concurrent statements, thus resulting in a *sequentialized SCG*. Compared to general SCGs, the sequentialized version is free of concurrency and pauses. Stepping through this control-flow graph from entry to exit corresponds to a single system reaction.

2.1.3 KTrace

SCCharts in KIELER can be simulated. The behavior of such a simulation can be recorded and later re-played. A recording of visible behavior is called a *trace*. A trace is thus a sequence of input-output pairs. A pair (I, O) at position n in the sequence means that when given the inputs I in the n-th reaction to the model then it will produce the outputs O.

The format for traces in KIELER is called *KTrace*. It is a simple textual language that is illustrated in Listing 2.1. Inputs are separated from outputs using an equals-sign followed by greater-than (`=>`). Multiple ticks are separated using a semicolon. The value of variables is carried over to the next tick if not explicitly given. Thus, in the example the input `a` has the value `true` in the second tick. Optionally, a label can precede a tick. Afterwards the label can be used as jump target to create infinite traces.

2.2 Model Checking

The following introduces concepts and technologies related to model checking that are used in this thesis. First, temporal logics are explained with a focus on LTL. Afterwards, the synchronous observer pattern is discussed, which can be used instead temporal logics to formulate certain temporal properties in synchronous languages. Finally, the model checkers NuXmv and SPIN are introduced.

2.2.1 Temporal Logics

Temporal logics are an extension to propositional logic. The term *temporal*, however, does not necessarily mean that they formulate statements relative to real-time. Instead, the abstract *order* in which events occur is expressed, e. g., *event B occurs after event A*, rather-than *event A occurs and 300ms later event B occurs*. However, some temporal logics focus on real-time properties, for example Timed Computation Tree Logic (CTL). In this thesis Linear Temporal Logic (LTL) is used in examples because it is well supported by model checkers and is well suited to describe discrete system reactions of synchronous programs.

LTL formulas over the set AP of atomic propositions and $a \in AP$ are formed by the following abstract syntax rules:

$$\varphi ::= true \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid X\varphi \mid \varphi_1 \, U \, \varphi_2$$

X is representing the next-operator, whereas U is representing the until-operator. Atomic propositions that occur in this thesis follow intuitively from the context and are not further defined. Formal semantics of LTL are not discussed here but are explained in detail in other literature, for example by Baier and Katoen [BK08]. Figure 2.4 illustrates the intuitive semantics of LTL. Operators G for "always" (*now and forever in the future*) and F for "eventually" (*eventually in the future*) can be expressed using the until operator as follows:

$$F\varphi \equiv true \, U \, \varphi \quad G\varphi \equiv \neg F \neg \varphi$$

The following illustrates the use of LTL to define the key properties of the *mutual exclusion problem*. Thereby, the first property is a safety-property, whereas the second is a liveness-property.

- Access to the critical section is given only to one actor at a time:
  $G(\neg crit_1 \vee \neg crit_2)$
- Access to the critical section is always given at some point in the future:
  $(G \, F \, crit_1) \wedge (G \, F \, crit_2)$

2.2.2 Synchronous Observer

The synchronous observer pattern is an alternative to temporal logics for formulation of safety-properties when working with synchronous languages [Rus14]. In this pattern the system to be verified is monitored by a program that runs in parallel and which will raise an error flag if it observes undesired behavior. Running the observer in parallel to the main system works in the synchronous MoC because the system state is broadcast immediately and thus no behavior is missed because of timing issues or lost messages. The task of verification is then to ensure that the error flag is never raised.

The pattern has the advantage that the same language to describe the system can be used to formulate the desired properties. Thus, no new formalism has to be learned, which can be useful to establish formal methods in practice. Furthermore, temporal aspects of properties, e. g., *in the **next** tick there shall be no error*, is handled by the compiler of the synchronous language. This enables the use of relatively simple algorithms for invariant checking instead of complex ones for, e. g., LTL. Halbwachs et al. have shown how to use the synchronous data-flow language Lustre for checking non-trivial safety-properties [HLR92]. The use of algorithms for checking invariants instead temporal properties could improve performance.

Rushby points out that the synchronous observer pattern does not only allow to formulate safety-properties but also to specify assumptions and axioms [Rus14]. For instance, it is possible to ignore misbehavior or undefined behavior that would not occur in the real target environment of the system.

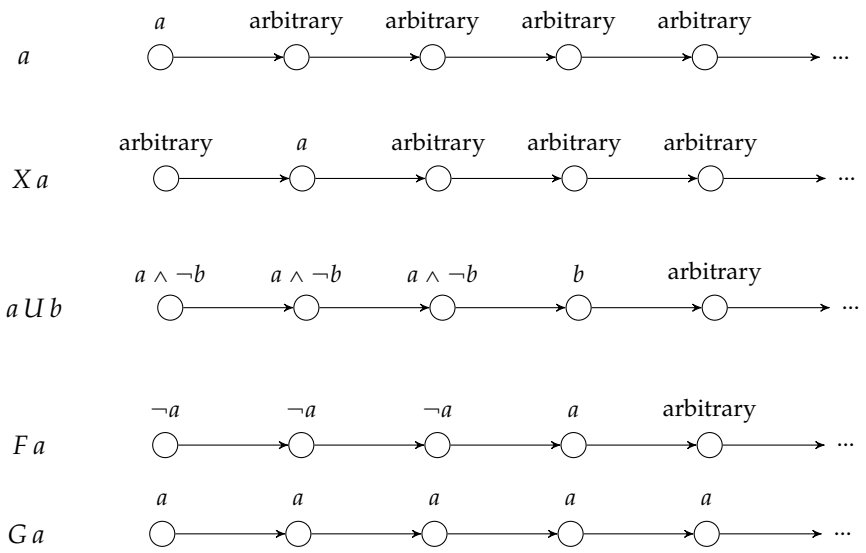


Figure 2.4. Illustration of the intuitive semantics of LTL for atomic propositions a and b [BK08].

A downside of the synchronous observer is that not all properties known from temporal logics can be expressed. Only past and current behavior can be observed but not the future in general. As a result, it is not possible to formulate unbounded liveness properties using this approach. Furthermore, the system that is given to a model checker is a composition of the main model and its observer. This composition is more complex than the main model alone, which can have a negative impact on the performance.

2.2.3 SPIN

The Simple PROMELA Interpreter (SPIN)[5] is a model checker for the the Process Meta Language (PROMELA). SPIN is one of the oldest model checkers and uses an explicit model checking algorithm. The state-space is constructed on-the-fly for checking a property. The verification of assertions and LTL formulas is supported. In SPIN this is done by first creating a verifier in C code, which is then compiled and executed to perform the model checking. The user can chose whether a breadth-first or a depth-first (default) search should be performed to find counterexamples. Unreachable states do not have a significant performance impact because they are not traversed in the constructed state-space.

The PROMELA syntax has been inspired by C and is designed to model parallel processes that communicate via shared variables and message passing. The program logic is described imperatively. However, if-statements and loops allow multiple parallel conditions and the next statement to be executed is chosen non-deterministically from all running processes.

The explicit model checking of SPIN allows to check dynamic behavior, meaning that processes can be spawned and started at run-time. SPIN has been used successfully to verify and detect bugs in real-world protocols. However, it is limited by the reachable state-space. For example, when an integer can reach all its possible values, then this will dramatically impact the time and memory required for

[5] `https://spinroot.com`

checking a property using SPIN. Thus, it is necessary to abstract from irrelevant states when writing PROMELA models.

2.2.4 nuXmv

nuXmv[6] is a symbolic model checker for synchronous finite-state and infinite-state systems. It extends the open source model checker NuSMV, which in turn was a re-implementation of SMV, the first model checker using Binary Decision Diagrams (BDDs). nuXmv offers a wide range of algorithms for model checking and its performance in certain categories is well placed when compared to other modern model checking systems [CCD+14]. Among others, available techniques make use of BDDs, Satisfiability (SAT) solving or Satisfiability Modulo Theories (SMT) solving.

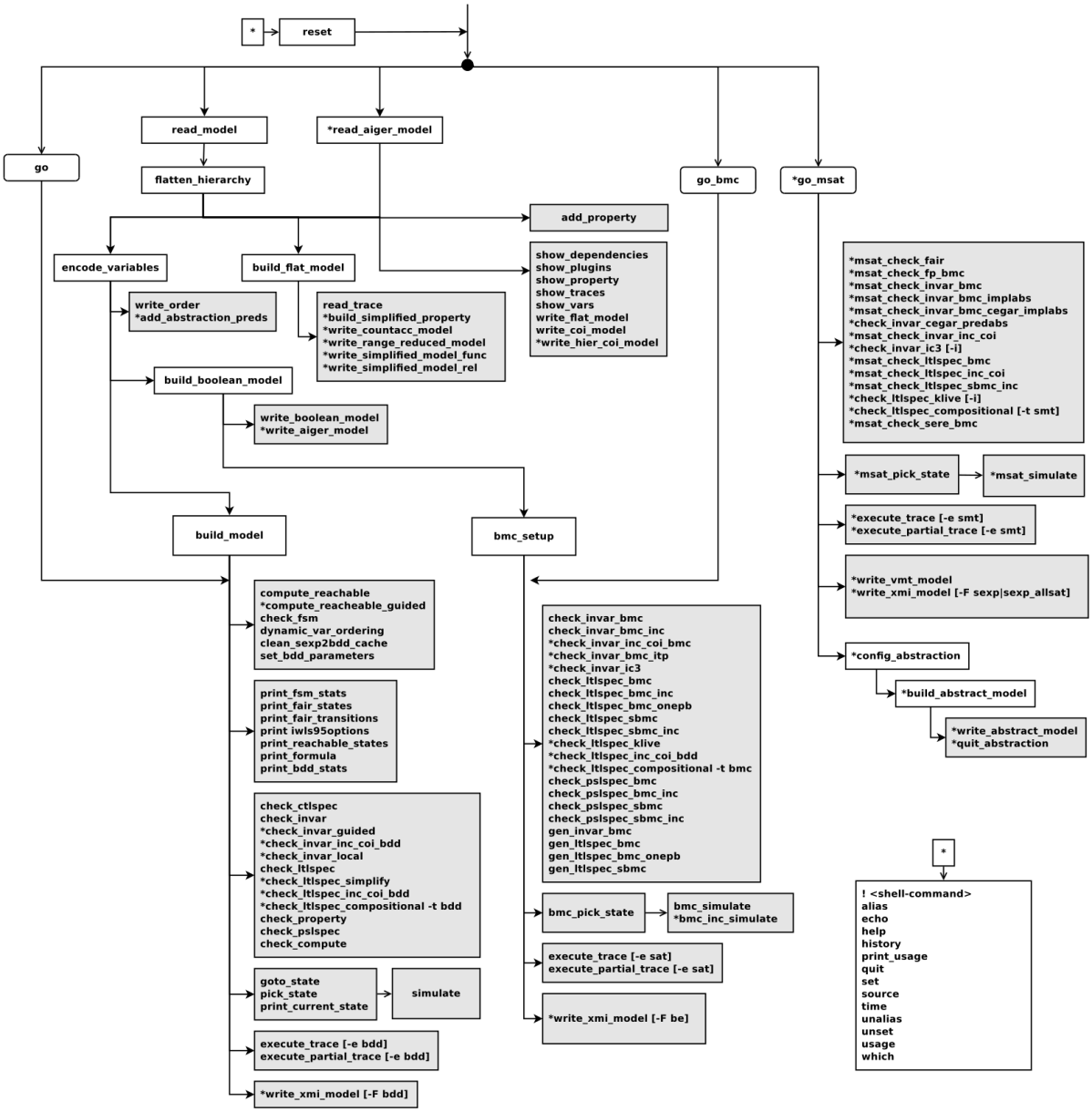
This is reflected in the nuXmv command palette, which is illustrated in Figure 2.5. Model checking algorithms that are reachable via the `go` command or ending in `_bmc` typically use BDDs. Algorithms reachable via the `go_bmc` command or ending in `_bmc` typically use bounded model checking and SAT techniques. Algorithms reachable via the `go_msat` command or starting with `msat_` typically use SMT techniques.

Moreover, abstraction/refinement algorithms are available, which is reflected in command names that contain the term `_cegar` (i. e., Counterexample Guided Abstraction Refinement) or `_predabs` (Predicate Abstraction). The suffix `_inc` indicates an incremental algorithm and the suffix `_inc_coi` indicates an incremental cone of influence approach. Several commands of nuXmv implement a combination of different approaches, for instance, `check_ltlspec_inc_coi_bdd`. Some of the new commands in nuXmv allow to check models with infinite domain variables, namely integers and real numbers. However, the explicit model checking approach that is used in SPIN is not available in nuXmv.

The SMV language has been designed to describe synchronous transition systems. This is done logically by giving equations that define the initial and following value of variables. The equations must not have cycles in their dependencies and must be deterministic. SMV allows for hierarchical models by abstracting and instantiating modules.

Depending on the model and used algorithm, nuXmv can check properties that are expressed in CTL, LTL (optionally extended with Past Operators), or the standardized Property Specification Language (PSL) [IEEE1850][EF07].

[6] `https://nuxmv.fbk.eu`



`https://es.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf`

Figure 2.5. Overview of the available commands in nuXmv and in which order they can be used. Commands with an asterisk (*) are new compared to the commands available in NuSMV. Most of the commands can be further parameterized and fine-tuned.

Chapter 3

Related Work

There are model checkers designed for different logics, models, and domains. A comparison of some model checkers can be obtained from various case studies.

Mazzanti et al. present a case study, in which a deadlock avoidance algorithm for train scheduling has been checked using ten different model checking systems [MF18]. Thereby, characteristics of the modeling languages and their impact on the algorithm implementation are identified. They did not focus on comparing the performance of model checkers. A fair comparison is difficult due to the strength and weaknesses of different model checking techniques. In general, model checking performance depends on the concrete model to be checked and the used algorithm and its configuration, for instance variable order in BDDs. However, their experiences with the different model checking systems can give a rough approximation of their potential. The fastest times were measured for NuSMV/nuXmv and SPIN.

Frappier et al. used 6 model checkers to verify an information system for a library [FFC+10]. They highlight the different aspects of the modeling as well as property specification languages and give an overview of the used model checking techniques. Also they compare the run-time needed to check increasingly complex instances of their problem. In this case study, the model checkers Fdr4 and Alloy outperformed the others.

An indirect comparison of model checkers can also be found in papers migrating from one tool to another, for example in the work of Choi, who migrated from NuSMV to SPIN for the design of a Flight Guidance System at Rockwell-Collins [Cho07]. SPIN was able to verify properties after applying optimizations to the SPIN model. NuSMV on the other hand was not able to verify these properties. However, it is noted that on unoptimized models NuSMV often performed better. Furthermore, results of SPIN may be unsound when bit-state hashing is used and the performance depends not only on the model, but also on the property to be checked. The author argues that creating efficient models for automatic verification requires expertise of the used model checking tool. This makes it difficult for developers to choose the optimal model checker for a specific task and thus to establish formal methods in the industry. Creating an optimized model for every property is not ideal because it is costly and can lead to inconsistencies between models.

A case study that compared model checking tools to verify a real-world example of a critical real-time system were done by Boniol, Wiels and Ledinot [BWL06]. For this case study three different versions of the system, two versions of the property and two different sets of hypothesis have been formulated to achieve a better comparison of the tools. Further, the target environment of the model has been designed as well to get a realistic setting. The model checkers that were tested are Lesar, SMV, Uppaal and the Prover Plug-in for SCADE. Lesar is a symbolic model checker for the synchronous language Lustre and is explained in Section 3.3. Uppaal[1] is a model checker and modeling environment for real-time systems specified in a visual state-machine language. SCADE and the Prover Plug-in are explained in Section 3.1.

[1] uppaal.org

The original specifications of the models were provided by a company in form of Esterel code. Motivation for the case study was that, at the time, the Esterel model checker failed to verify some properties. This is why the models have been re-implemented in Uppaal and Lustre to make use of other verification systems. As Lustre is the semantic root of SCADE, it is supported by the Prover Plug-in. Further, an automated translation from Lustre to SMV exists. Thus a Lustre and Uppaal model were sufficient to use all model checkers of the case study.

SMV was the only tool that successfully checked all properties of the study. SMV outperformed Lesar, although both use symbolic model checking techniques. The authors were not able to benefit from the real-time modeling of Uppaal. A possible explanation for this is given. They triggered the controller model with a fixed frequency so that the continuous clock domain of Uppaal is discretized. This could be a reason why the real-time domain of Uppaal was not beneficial in the case study. The authors were not able to get positive results from the Prover Plug-in. They argue that this may be due to the number of induction steps needed by the Prover Plug-in to verify these particular properties of the models. In another experiment that did not have these hurdle, the Prover Plug-in showed very good performance, much better than Lesar.

Another kind of case study can be found by investigating solutions for the Steam-boiler Control Specification Problem [ABL96]. It gives an informal, realistic specification of a steam-boiler as an example for a safety-critical system. The idea was to have a practical, non-trivial, non-academic example to compare techniques for formal specification and verification in a competition. The authors received contributions from all over the world. Models of the steam-boiler specification have been written in various languages including Lustre [CD96], Esterel [Bou97b] and StateCharts [BW96].

3.1 SCADE

The Safety-Critical Application Development Environment (SCADE)[2] is a commercial tool that is used in various industries with high requirements for correctness and reliability, e. g., avionics, railway and nuclear power plants. SCADE uses a visual language for modeling system behavior, which is also done in KIELER using SCCharts. However, the semantic root of SCADE is the synchronous language Lustre. A certified compiler can generate C and Ada code from SCADE models.

SCADE features tools for the specification of system properties and their formal verification. This can be done by modeling a synchronous observer that monitors the property to be checked. There exists a library of predefined elements to ease the creation of synchronous observers in SCADE. Included are common patterns such as logical implication. The library can be extended by users to express complex properties using the synchronous observer pattern. In a similar way, an extensible library of models could be created that ease the use of synchronous observers in SCCharts.

Model checking in SCADE is done via the Prover Plug-in, which is provided by Prover Technologies[3]. Two modes are available in this model checker, namely *proof* and *debug*. The debug mode can quickly find counterexamples when they exist. This has been experienced in the case study by Boniol, Wiels and Ledinot [BWL06]. Thus this mode is suited in early development stages, where the developer is not sure whether the designed property is true or even well expressed. However, the *proof* mode should be used when the developer thinks that the property holds to get a complete verification.

[2] esterel-technologies.com/products/scade-suite
[3] prover.com

3.2 mbeddr

The open-source tool mbeddr[4] aims at providing a complete solution for embedded system development from requirements specification to implementation, testing and verification. Therefore, mbeddr provides an extensible set of Domain Specific Languages (DSLs) for the C language. For instance, there is an extension that enables to add physical unit information to members of C structs. Expressions with these members are then checked to produce sound units, e. g., when used in an assignment. This can help in avoiding bugs. The language tooling of mbeddr builds on the Meta Programming System from JetBrains, which is based on Projectional Editing. This means the Abstract Syntax Tree (AST) of code is modified directly instead of textual languages that have to be parsed first. This allows for very flexible code representations and editing in mbeddr. In contrast to this, KIELER uses Xtext for implementation of its DSLs, thus using a traditional parser-based approach.

Noteworthy of mbeddr in the context of this thesis is how the verification is implemented. The backend tools for verification that are used in mbeddr are NuSMV, CBMC and Yices. A paper illustrates how CBMC has been used to automatically create inductive proofs about mbeddr's state-machine extensions for C [MVR14]. Such state-machines and certain other code constructs can be automatically visualized in the mbeddr IDE. This is similar to how SCCharts are visualized from a textual syntax in KIELER. However, concurrent state-machines with a synchronous semantics are not part of the mbeddr system.

Verification in mbeddr is not limited to state-machines. For instance, an interface can be defined with methods to be implemented and pre and post conditions that define a contract for the methods. Implementations of that interface can then be checked against that contract using formal methods. An execution stack trace is given as counterexample if a property does not hold.

3.3 Lesar

Lesar is a model checker for the synchronous language Lustre. Lesar implements both, an explicit approach that enumerates all states, and a symbolic approach that uses BDDs for model checking. The internals of Lesar's symbolic model checking approach are explained by Raymond [Ray08]. The paper gives technical details about the BDD algorithms. Furthermore, it is explained how Boolean abstraction can be used to reduce the amount of system configurations when checking safety properties. The discussed algorithms could be adapted to implement model checkers for other synchronous languages, e. g., SCCharts. Raymond also illustrates that the synchronous observer approach is used in Lesar instead temporal logics to specify safety properties. As a result, it is not possible to check, e. g., liveness properties using Lesar as these cannot be expressed using the synchronous observer approach. This limitation has been accepted in favor of handling real-life safety properties by means of abstraction techniques.

How Lesar can be used for specification and verification of safety properties is illustrated on a small example by Halbwachs et al. [HLR92]. The authors pointed out that there are cases in which the explicit approach implemented in Lesar performs better than the symbolic approach and vice versa. Anyhow, the version of Lesar that was used in the paper was able to check about 1,000,000 states in less than an hour, which was considered a *"reasonable time"*. Since computing hardware has seen noteworthy improvements over the last decades, the Lesar model checker is likely to handle more states in the same time today. Furthermore, the paper explains how modular verification of a composed system S can be achieved for suited properties. This is done by first verifying properties of

[4] `mbeddr.com`

smaller components that are used in *S*. Afterwards this knowledge can be used as assumption about the components instead of their concrete implementation, thus resulting in an abstraction of *S*, which can be much easier to prove.

3.4 Xeve

Xeve is a graphical verification environment for the synchronous language Esterel [Bou97a]. It utilizes a library for handling BDDs to minimize and verify the finite state machines that are implicitly encoded as sequential circuits during the compilation of Esterel programs. Xeve takes the circuit in the Berkeley Logic Interchange Format (BLIF). Minimized state machines are stored in a textual format (Fc2) that can be visualized for graphical navigation of the state space.

Esterel semantics are based on signals. A signal is either *present* or *absent* in any given tick. A signal is present when it must be emitted and it is absent when it cannot be emitted. Xeve verifies that an input or output signal of the program can be emitted or not. A minimal execution trace is given that shows how to reach a state with the corresponding emission status of the signal. Such a trace can be loaded in the graphical Esterel simulator to replay the program logic. Furthermore, the graphical tools of Xeve can be used to restrict inputs and outputs to a fixed value (present or absent) when doing a verification. Inputs and outputs can be marked as ignored when determining the equivalence of states for minimization of the transition system.

Xeve requires the synchronous observer approach to formulate properties, similar to Lesar.

Chapter 4

Concept

This chapter discusses the integration of model checking features for SCCharts in KIELER. The implementation of a dedicated model checker for SCCharts is considered but dismissed in favor of re-using and evaluating existing model checker implementations. The model checkers nuXmv and SPIN have been identified for this task. Properties of a reactive model such as SCCharts are considered and how temporal properties change when a model is translated to another language.

These considerations are respected when giving a translation from a low-level representation of SCCharts, namely the sequentialized SCG, to the input languages of the chosen model checkers. nuXmv uses the SMV language, in which a transition system is described using data-flow equations. Two possible translations of the tick logic to this language are discussed. The first uses a program counter approach to implement the control-flow of the SCG. The second uses an Static Single Assignment (SSA) form of the sequentialized SCG to express its underlying data-flow in the SMV language. Thereby, this thesis focuses on the SSA approach because it preserves temporal properties of the original model.

SPIN is a model checker for PROMELA. Thus, a translation from the sequentialized SCG to PROMELA is presented to enable the use of this model checker for SCCharts.

Finally, it is illustrated how the synchronous observer pattern can be implemented in SCCharts to express the model, its safety-properties and assumptions within the same language.

4.1 Model Checker Selection

There are various model checking algorithms and implementations as explained in Chapter 3. Model checkers are often dedicated to do verification on a specific problem. For instance, the model checkers Xeve and Lesar have been implemented to verify programs written in Esterel, respectively Lustre. In a similar way it is possible to write a dedicated model checker for SCCharts. However, it is not trivial to find an optimal model checking algorithm for this task. The explicit and symbolic approach can both be more efficient in some cases, which has been experienced in several case studies. Further, efficient model checking algorithms are typically complex to handle large state-spaces. As a result, they are not trivial to implement.

An alternative to the implementation of a particular model checking algorithm for SCCharts is re-using existing model checkers. SCCharts can be translated to a language that is supported by existing model checking solutions to evaluate different algorithms. In the scope of this thesis, the model

Table 4.1. Comparison of nuXmv and SPIN. The model checkers differ in their modeling language, semantics and model checking algorithm.

	nuXmv	**SPIN**
Language	Logical (data-flow)	Imperative
Semantics	Synchronous	Asynchronous
Model Checking Approach	Symbolic (various)	Explicit

checkers nuXmv and SPIN have been identified to evaluate different model checking approaches for SCCharts. Table 4.1 names differences between SPIN and nuXmv.

KIELER is a cross-platform tool that runs on Windows, MacOS and Linux. As a result, it is desirable that related tooling such as model checkers are available for these platforms as well. Both nuXmv and SPIN meet this requirement.

nuXmv is a model checker that implements high-performance symbolic model checking techniques, which has been demonstrated in the hardware model checking competition of 2013 [CCD+14]. SMV has been evaluated before and performed well in different case studies as discussed in Chapter 3. Furthermore, translations from the synchronous language Signal to SMV have been proposed to leverage the model checking limitations of its dedicated model checker [PG09].

nuXmv provides a wide range of model checking algorithms. This is useful in the evaluation to find an optimal model checking algorithm for SCCharts. An advanced feature of nuXmv is the support for unbounded integers and real numbers. Properties can be formulated in different temporal logics, including LTL and CTL.

Thus, nuXmv allows for a wide range of models and properties. Different algorithms can be evaluated and it meets the requirements for cross-platform tooling. As a result, it has been chosen in this thesis for model checking of SCCharts.

On the other hand, SPIN complements nuXmv and the SMV language in several ways. First, it uses an explicit model checking algorithm that constructs the traversed state-space on-the-fly. This method is not available in nuXmv. Second, the modeling languages of SPIN and nuXmv are different. PROMELA is an imperative language to model asynchronous processes that communicate via messages and shared memory. In contrast to this, the SMV language is a data-flow language to describe synchronous transition systems. As a result, a translation from SCCharts to PROMELA and SMV allows to evaluate different approaches.

Both model checkers are used as verification back-end in other tools [CCD+14][Hol97]. There are plugins to translate the object oriented, reactive language Rebeca[1] to SMV and PROMELA. SATABS is a tool for model checking ANSI-C and supports (among others) NuSMV and SPIN as back-end [CKS+05]. NuSMV is used for model checking state-machines in mbeddr [VRK+13].

4.2 Translation Considerations

Reactive systems have some features that must be considered when doing model checking. First, the reaction can happen infinitely often. Second, inputs to the system can be different for every reaction, but they are constant within a reaction. Depending on the Model of Computation, these properties are either implicit or must be modeled explicitly.

An important consideration for model checking a language is the abstraction level on which it is done. The model is more abstract on a high-level. Using this information can help to implement efficient model checking. The abstraction is lost when translating high-level features to simpler ones. On the other hand, models with high-level features are still subject to compilation. Therefore, a bug in the compiler could change the behavior of the system, such that properties of the original model do not hold anymore.

The exact semantics of language constructs must be preserved in a translation when doing model checking. A direct mapping of high-level features to another language is not possible in most cases such that a non-trivial translation must be given. Thereby, semantic equivalence is difficult to establish.

[1] rebeca-lang.org

Semantics of low-level features on the other hand are typically easier to grasp and thus to translate. Additionally, there are typically fewer low-level than high-level features. As a result, giving a translation on this level has less potential for errors. A bug in the compiler can lead to code in which properties of the original model do not hold. In contrast to this, doing model checking on the final code meets the principle *"What You Prove Is What You Execute"* as mentioned by Berry [Ber89].

As a result, this thesis focuses on a low-level representation of SCCharts, namely the sequentialized SCG that is explained in Section 2.1.2. In this form, only conditionals, assignments, pre-operators and the sequential execution of statements remain. This drastically eases the translation to other languages, including existing model checking systems. Another advantage is that other high-level languages are also compiled to a sequentialized SCG in KIELER. For instance there is a sequentially constructive version of Esterel (SCEst) [SMR+17]. As a result, doing model checking on this level can be beneficial not only for SCCharts.

4.2.1 Persisting Temporal Properties

It is important to note how the number of visible reactions and thus temporal properties changes when a model is translated to another language or abstraction level. In SCCharts only the system behavior of macro-ticks is relevant and visible from the outside as discussed in Section 2.1.1. Thus, invariants of the system hold when the reaction is complete but not necessarily during its computation. Similarly, a next-operator in LTL refers to the *end* of the next tick in the context of SCCharts. The complete code of a sequentialized SCG is meant to be executed as a single macro-tick. Thus, invariants and temporal properties must be checked after the last statement of the SCG has been executed. This must be considered in a translation. However, in PROMELA every statement is per default considered a reaction and can be referenced by a next-operator in LTL.

Temporal properties formulated from a macro-tick perspective can be adapted to models with observable micro-ticks. A timeline of reactions containing micro-ticks and macro-ticks is illustrated in Figure 2.2. Intuitively, an observable state that does not correspond to the end of a macro-tick must be ignored because at this point the computation of the reaction is incomplete. This leads to the following LTL operators:

$\triangleright\ F'\,a := F\,(\mathit{tickend} \wedge a)$

$\triangleright\ G'\,a := G\,(\neg\mathit{tickend} \vee a)$

$\triangleright\ a\,U'\,b := ((\neg\mathit{tickend} \vee a)\,U\,(\mathit{tickend} \wedge b))$

$\triangleright\ X'\,p := X\,(\neg\mathit{tickend}\,U\,(\mathit{tickend} \wedge p))$

For example, consider the LTL property $G\,X\,(a \rightarrow o)$ of the ao-model from Listing 5.1. This property can be translated to ignore intermediate system configurations, for example, $G\,(\neg\mathit{tickend} \vee X\,(\neg\mathit{tickend}\,U\,(\mathit{tickend} \wedge (a \rightarrow o))))$.

A proof for the correctness of the adapted LTL operators is not given here as they have not been used in the implementation of this thesis. However, intuitively temporal properties get more complex when losing a one-to-one relation between reactions of the original high-level model and its translation.

An alternative to adapting temporal properties is to preserve what is considered a macro-tick. For instance, PROMELA features an `atomic` keyword to consider multiple statements as a single reaction.

4.3 Translation to SMV

SMV is a data-flow language in which a transition system is expressed using equations. Thereby, the order of equations is irrelevant for the semantics. However, no cycle in the dependencies between equations and only one defining equation per variable is allowed. There is an equivalence between data-flow code and hardware circuits. Each equation represents one wire of the circuit and operators in equations correspond to logic gates.

Sequentially writing to the same variable, which can be done in imperative languages, must be split to different equations in data-flow. Why this is necessary becomes clear when thinking about the corresponding hardware circuit. Each wire of the circuit can only carry one distinct value in a reaction. Otherwise the circuit would be unstable.

Defining a dedicated variable for each distinct value of a sequentially written variable is done when creating the SSA form of a program. This has been done for the SCG by Schulz-Rosengarten [Sch16]. In the following, a sequentialized SCG in SSA form is called an SSA SCG.

A translation from control-flow to data-flow can be done in several ways. The following first discusses general constructs of the SMV language. Afterwards two approaches are discussed to express the tick logic of the sequentialized SCG in SMV. The first is using a *program counter* whereas the second is using the *SSA form* of a sequentialized SCG. Thereby, this thesis focuses on the SSA approach because it preserves temporal properties of the sequentialized SCG.

4.3.1 SMV Constructs

An SMV program for nuXmv is structured using keywords. A `MODULE` with the name `main` is used as entry point to the system, similar to how the `main` function is the entry point to a program in C.

In a module, `VAR` is used to start the declaration of variables that contribute to the state-space. Undefined variables in SMV can take any value of their domain non-deterministically. Thus, to model all possible values for an input variable in SCCharts, it can be declared in SMV without following defining equations. This is different from PROMELA semantics. In PROMELA variables are persisted and initialized with `false` for Booleans and with `0` for integers such that they must be assigned non-deterministically to model all possible inputs.

The `ASSIGN` keyword starts the code block in which variables can be defined. This is done using one equation for the initial reaction (`init(x) := ...`), and one equation for all following reactions (`next(x) := ...`). If both are the same for the initial and following reactions then this can be written in a single equation (`x := ...`). However, the variables that are defined in the `ASSIGN`-block must have been declared in the `VAR`-block.

The `DEFINE`-block works differently. It associates an identifier on the left side with an expression on the right side. The nuXmv manual explains it as follows:

> A define statement can be considered as a macro. Whenever a define identifier occurs in an expression, the identifier is syntactically replaced by the expression it is associated with.

This means the expressions from a `DEFINE`-block do not contribute to the actual state-space. They are merely used to express the logic of the reaction. All variables from a sequentialized SCG that do not impact the following tick can be defined this way.

A special signal named `GO` is used in the sequentialized SCG to provide the entry point of the tick logic as discussed in Section 2.1.2. It must be set to `true` in the initial tick and to `false` in all following ticks. When translating a sequentialized SCG, the `GO` variable must be set accordingly in the `ASSIGN`-block.

```
@Invariant "i -> !o"
scchart ivar_example {
  input bool i
  output bool o = i
}
```

```
o0 = pre(o);
if (_GO) {
  o1 = i;
  o = o1;
} else {
  o = o0;
}
```

Listing 4.2. Textual SCChart with a failing property and the resulting logic in SCL

```
-> State: 1.1 <-
  i = TRUE
  _GO = TRUE
  _po = FALSE
  o = TRUE
  o1 = TRUE
  o0 = FALSE
```

Listing 4.3. SMV counterexample when using VAR for inputs

```
-> State: 1.1 <-
  _GO = TRUE
  _po = FALSE
  o0 = FALSE
  o = TRUE
-> Input: 1.2 <-
  i = TRUE
  o1 = TRUE
-> State: 1.2 <-
  _GO = FALSE
  _po = TRUE
  o0 = TRUE
```

Listing 4.4. SMV counterexample when using IVAR for inputs

In SMV it is possible to define inputs in a dedicated block, namely as part of `IVAR` instead with all other variables in the `VAR`-block. However, there are some restrictions on variables defined as part of `IVAR`. They cannot be used in CTL formulas, init-statements of the `ASSIGN`-block, and invariants. Furthermore, counterexamples change when `IVAR` variables are present. Listing 4.3 and Listing 4.4 show the difference for a small counterexample. When using `IVAR`, the input variables are named separately, conceptually for the following tick. Per definition, the initial state in SMV is not allowed to depend on `IVAR` variables (they cannot be used in init-statements). `DEFINE` identifiers that depend on input variables are named as part of the input-block in the counterexample. However, from an SCCharts perspective `IVAR` variables belong to the previous SMV state. In Listing 4.4 the input `i` is considered an input to the state `1.2` but from an SCCharts perspective it belongs to the previous state `1.1`. Thus, when parsing SMV counterexamples the separation marks `-> Input: ... <-` can be ignored as the following variables still belong to the same state from an SCCharts perspective.

This thesis focuses on a translation where inputs of the system are declared in the `VAR`-block of SMV modules because of the limitations on `IVAR` variables and how they are handled in SMV, which breaks with the SCCharts perspective on input variables. Nonetheless, a translation from SCCharts to SMV that uses `IVAR` for input variables has been implemented and can be enabled from the graphical user interface.

The interested reader can find further features of the SMV language in the nuXmv user manual[2].

4.3.2 Modeling the Tick Logic via Program Counter

A program counter is used in processors to sequentially execute instructions on synchronous hardware. Similarly, it is possible to express control-flow statements in a synchronous data-flow language. This is illustrated in Listing 4.6. A new program counter variable is introduced. The domain of this variable are the statements that can be executed. Data-flow equations are then written to reflect the statement that is selected by the program counter. The next value of the program counter is selected to reflect the control-flow of the original model.

[2] `https://es.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf`

```
x = 1;      // l1
if (x > 0)
    x = 2; // l2
else
    x = 3; // l3
halt;       // l4
```

Listing 4.5. A program with control-flow.

```
// pc is one of l1, l2, l3, l4
pc = if (pc == l1) { if (x > 0) pc = l2 else pc = l3 }
     else if (pc == l2) { pc = l4 }
     else if (pc == l3) { pc = l4 };

x = if (pc == l1) 1
    else if (pc == l2) 2
    else if (pc == l3) 3;
```

Listing 4.6. The control-flow program in data-flow using a program counter (pc).

An advantage of the program counter approach is that it can express any control-flow, including jumps. Furthermore, no variables besides the program counter need to be added. In contrast to this, the translation to an SSA form introduces a new variable for every distinct value that is assigned to a variable. However, these variables can be defined in the `DEFINE`-block of SMV and do not contribute to the state-space of the model.

A disadvantage of the program counter approach is that the number of reactions increase because each program counter step requires one reaction. Thus, temporal properties must be adapted to express that they hold only when the reaction of the original model has been completed as explained in Section 4.2.1.

In the sequentialized SCG there are many Boolean guards that define which statements should be executed in the current tick. However, only pre-operators carry information about guards across tick borders and must be persisted in the form of flip-flops or registers. When using the program counter approach, all variables must be persisted, which results in an increased number of variables in the `VAR`-block. The evaluation in Chapter 6 shows that this can have a negative effect on performance, for instance, in algorithms that make use of Binary Decision Diagrams. The impact becomes clear when thinking about the underlying hardware circuit that is represented by data-flow equations. Values of guards correspond to wires and gates. Now, when using the program counter approach these must be persisted as well, requiring a register for what was modeled as a simple wire before.

In conclusion the program counter approach can express any control-flow but increases the number of global variables and requires adaptation of temporal properties. As a result, this thesis focuses on a translation to SMV without program counters by using an SSA form of the sequentialized SCG.

4.3.3 Modeling the Tick Logic via SSA

The sequentialized SCG encodes its tick logic using data-flow guards as explained in Section 2.1.2. This eases the translation to a data-flow language such as SMV without using a program counter. In SSA form there is one distinct variable for each value. This makes it possible to use data-flow equations to describe the logic of a sequentialized SCG in a single reaction. Listing 4.7 gives an example of the resulting code using the translation presented in this section.

Few changes to the SSA SCG that is created from the work of Schulz-Rosengarten were necessary for the translation to work. First, the pre-operator must return the value of a variable at the *end* of the previous tick. Thus, the operand is always the *last* version of a variable in case the SSA form has introduced multiple versions for different values.

Second, the SSA form implemented by Schulz-Rosengarten does not retain the variable names of the original model. However, it is easier to work with an SSA SCG that retains the variable names

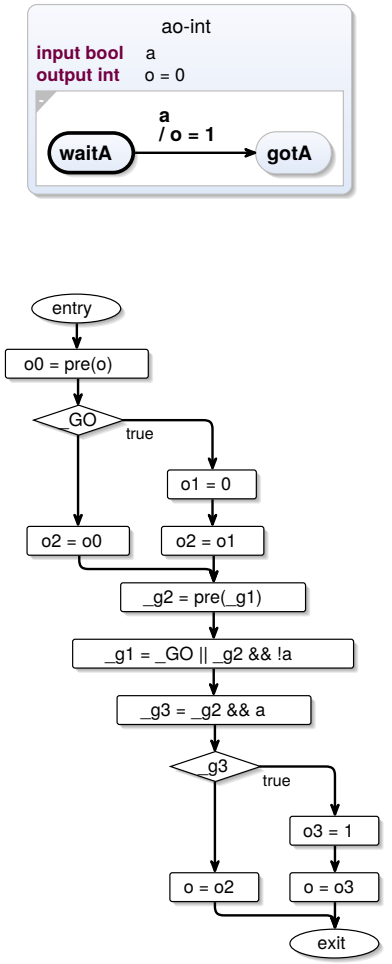


Figure 4.1. The ao-int model and its sequentialized SCG in SSA form

```
1  MODULE main
2    VAR
3    a : boolean;     -- input a
4    _pg1 : boolean; -- pre(g1)
5    _po : 0..1;      -- pre(o)
6    _GO : boolean;   -- GO signal
7
8    DEFINE
9    o0 := _po;
10   o1 := 0;
11   o2 :=
12     case
13       _GO : o1;
14       TRUE : o0;
15     esac;
16   o3 := 1;
17   _g1 := _GO | _g2 & !a;
18   _g2 := _pg1;
19   _g3 := _g2 & a;
20   o :=
21     case
22       _g3 : o3;
23       TRUE : o2;
24     esac;
25
26   ASSIGN
27   init(_GO) := TRUE;
28   next(_GO) := FALSE;
29
30   init(_pg1) := FALSE;
31   next(_pg1) := _g1;
32
33   init(_po) := FALSE;
34   next(_po) := o;
35
36 LTLSPEC
37   G X(a -> o=1);
```

Listing 4.7. SMV code for the SSA SCG

of the original SCChart. Thus, a renaming of the last variable version in SSA form to the name of its counterpart in the original SCCharts model has been added. This makes sense, because the observable system reaction of an SCCharts is always at the end of a tick, such that only the last SSA version of a variable corresponds to an output of the original model.

Third, the first version of a variable in the SSA SCG refers to its last version from the previous tick. This relation was implicit in the original SSA SCG implementation and has been made explicit using an assignment with a pre-operator at the beginning of the reaction.

The resulting SSA SCG with these additional changes is presented for the ao-int model in Figure 4.1. Four intermediate versions of the variable `o` have been introduced, namely `o0` to `o3`. The last version of the variable is called `o`, just as in the original SCChart. The first version `o0` is equal to the last version from the previous tick, which is made explicit using an assignment with the pre-operator. Also, `g2` depends on the *last version* of `g1` from the previous tick.

The SSA SCG is still a control-flow representation of the reaction with multiple assignments to the same variable depending on the branch that is taken in conditionals. However, these assignments are mutually exclusive and thus can be expressed in data-flow using a single equation. Therefore, the value to be written is selected depending on which assignment would be executed. For instance, `if (c) {a=1} else {a=2}` can be written as `a = if (c) 1 else 2`. In SMV the `case`-construct is used for such selections. Thereby, the condition `TRUE` can be used to define a default case. As soon as there is only a single defining equation for each variable, they can be written in data-flow. At this point conditionals have been encoded in the equations and control-flow edges of the SCG are handled by dependencies between the data-flow equations.

The data-flow guards and all variables that are introduced via SSA can be defined in the `DEFINE`-block of an SMV module because they do not need to be persisted and thus do not contribute to the state-space. This makes sense as guards represent wires of a hardware circuit. The following state of a well-formed circuit depends only on inputs and the current values of its registers. In SMV, these are the variables that are declared inside a `VAR`-block.

In the translation presented here, variables that implement the pre-operator correspond to registers of the circuit because they must be persisted across tick-borders. As a result, all pre-variables must be listed in the `VAR`-block. When resetting a hardware circuit, registers are reset to their default value. This can be implemented in SMV by defining the initial value of pre-variables. For Boolean pre-variables the initial value is `false` whereas for integers the value is `0`. Thus, a pre-operator `pre(e)` can be implemented using a pre-variable `pe` that is initialized in the `ASSIGN`-block as `init(pe) = FALSE`, respectively `init(pe) = 0`. In all following reactions the current value of the operand is persisted, thus `next(pe) = e`. Thereby, `pe` must be declared as global variable in the `VAR`-block.

The presented translation from the SSA SCG to SMV cannot handle variables that are input and output at the same time. The variable would be added to the `VAR`-block because it is an input to the model. There would also be an equation that sets the variable according to the tick logic in the original model. Thus, there are either two definitions of the same variable and the SMV model is rejected, or the input is defined, such that not all input configurations are considered and model checking would be incomplete. From a hardware perspective, having the very same variable as input and output is only possible for the trivial case that it stays constant inside the reaction, thus representing a wire. Having the same variable as input and output works in SCCharts and the PROMELA model, because the reaction is computed sequentially in control-flow. A translation to hardware-like SMV code as presented in this thesis cannot handle this. Anyhow, this restriction can easily be worked around by having two separate variables in the original model, one for the input, and one for the output.

The SSA SCG approach to translate SCCharts to SMV is limited to models that can be transformed into an equivalent hardware circuit. This includes all models that are accepted in the netlist-based

Table 4.2. Translation patterns from SCL to PROMELA.

	SCL	PROMELA
Assignment	`x = e;`	`x = e;`
Conditional 1	`if (c) { stmt1 } else { stmt2 }`	`if :: (c) -> stmt1; :: else -> stmt2; fi`
Conditional 2	`if (c) { stmt }`	`if :: (c) -> stmt; :: else -> skip; fi`
Ternary operator	`p ? q : r`	`(p -> q : r)`

Pre-operator	`pre(e)`	`// Introduce new pre-variable`
		`TYPE_OF_E pe;`
		`...`
		`// After tick logic (set pre-variable)`
		`pe = e;`
		`...`

compilation approach in KIELER. A more general set of SCCharts can be compiled using the priority-based compilation approach [HMA+14]. However, the resulting control-flow code can contain jumps that are taken multiple times within the same tick to dynamically schedule the nodes of an SCG. Such control-flow can be translated to an SMV model via the program counter approach explained in Section 4.3.2.

4.4 Translation to PROMELA

The following presents a translation from the sequentialized SCG to PROMELA. First, it is presented how the tick logic from the SCG can be expressed in PROMELA. Second, general features of reactive systems are considered and expressed using PROMELA constructs. Finally, it is discussed how assumptions and temporal properties of the original model can be mapped to PROMELA.

Figure 4.2 shows an SCChart and its corresponding sequentialized SCG. The PROMELA code that results from the translation presented in this section is shown in Listing 4.8.

4.4.1 Modeling the Tick Logic

The sequentialized SCG that is created when doing netlist-based compilation of KIELER SCCharts represents the tick logic using control-flow. Language constructs required to express this are conditionals, assignments, variables, and sequentiality. These are available in imperative languages such as PROMELA or C. The translation patterns for conditionals, assignments and the pre-operator are presented in Table 4.2.

The translation of the pre-operator for a variable `e` requires a new variable `pe` that must be declared globally because it contributes to the state-space. Setting the new variable to the value of `e` can then be done after the tick logic is complete. For this to work, `pe` must be declared with the same type

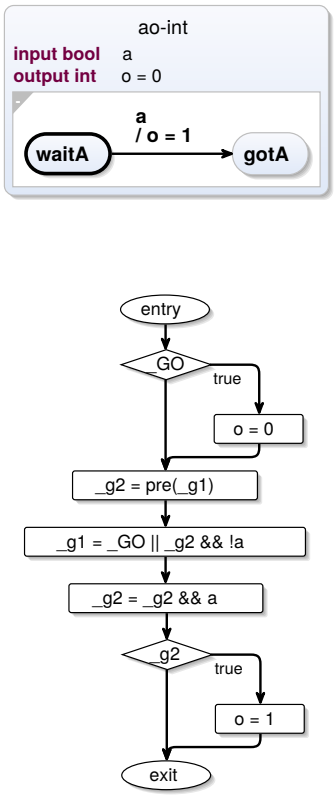


Figure 4.2. The ao-int model and its sequentialized SCG.

```
1  ltl always_in_next_tick_a_implies_o
       { X( [](X(!a || o==1)) ) }
2
3  bool a; // input a
4  int o;  // output o
5  bool _pg1 = 0; // pre(g1)
6  bool _GO = 1; // GO-signal
7
8  init {
9    do
10   ::
11   atomic {
12     bool pmltickend = 0;
13
14     // Set random inputs
15     if
16     :: (1) -> a = true;
17     :: (1) -> a = false;
18     fi
19
20     d_step {
21       bool _g1;
22       bool _g2;
23       bool _cg2;
24
25       // Tick logic
26       if :: (_GO) ->
27       o = 0;
28       :: else -> skip;
29       fi
30       _g2 = _pg1;
31       _g1 = _GO || _g2 && !a;
32       _g2 = _g2 && a;
33       if :: (_g2) ->
34       o = 1;
35       :: else -> skip;
36       fi
37       // After tick logic
38       _pg1 = _g1;
39       _GO = 0;
40
41       pmltickend = 1;
42     }
43   }
44   od
45 }
```

Listing 4.8. PROMELA code for the sequentialized SCG. The LTL property at the top is $G\,X\,(a \rightarrow o = 1)$ plus an initial next-operator to enter the tick-loop. A variable `pmltickend` is used to identify the start and end of a tick for parsing counterexamples.

as `e`. This way `pe` stores the value of `e` from the previous tick, thus implementing the pre-operator. Occurrences of `pre(e)` are replaced with a reference to `pe`.

When using a modeling language that supports non-determinism, care must be taken that the tick logic from the SCG is kept deterministic. For a translation to PROMELA this means that every if-statement must have an else-branch. Otherwise the PROMELA model will wait until the if-condition evaluates to true. PROMELA semantics differ in this case from if-statement semantics known from C or SCL.

A special signal of the sequentialized SCG model is the `GO` variable as explained in Section 2.1.2. It must be set to `true` in the initial tick and to `false` in all following ticks. PROMELA uses Boolean values in the same way as C. Thus, `0` represents `false` and all other values represent `true`. As a result, the `GO` variable is set accordingly in the translated PROMELA code. Thereby, it is important that the variable is set to zero after the pre-variables because a pre-operator can depend on `GO` (i. e., `pre(GO)` in the SCG).

4.4.2 Modeling Reactive Systems

It is necessary to encode the characteristics of reactive systems in the PROMELA model. The tick logic must be called infinitely often and with random inputs. Code patterns that achieve this are presented in Table 4.3. To keep temporal properties of the original model, the reaction has been made atomic inside the infinite loop. This keeps the one-to-one relation of an SCCharts reaction and PROMELA such that an adaptation of temporal properties as discussed in Section 4.2.1 is not necessary. However, an exception to this is an initial setup needed to enter the tick loop. SPIN needs one reaction for this. As a result, the LTL formula for an SCCharts model is prepended by a next-operator in the PROMELA model.

There are two language constructs in PROMELA to make multiple statements atomic, namely the `atomic`-block and the `d_step`-block. The SPIN manual states that `d_step` is more efficient but it can be used only for code that is fully *deterministic*[3]. However, inside the `atomic`-block of the presented tick loop translation there are non-deterministic constructs to set random inputs. Nonetheless, `d_step` can be used for the tick logic of the sequentialized SCG in PROMELA, which can improve performance. For instance, the time required for model checking the DVD-Player model that is described in Section 6.2.1 dropped from 5.18 seconds to 2.00 seconds when using `d_step` to indicate the deterministic reaction of the tick logic.

There are potentially many guards in the sequentialized SCG that encode the statements to be executed as explained in Section 2.1.2. These are re-calculated in every tick and do not contribute to the global state-space. As a result, they do not need to be declared in the global scope. Instead, they can be declared locally for the reaction. It has been found that declaring the guards locally reduces the traversed number of states by SPIN when using a breadth-first-search. For instance, in the `ao-int` example it reduces the number of stored states from 66 to 53 and the number of traversed transitions from 80 to 66.

4.4.3 Specifying Properties

The semantics of LTL formulas in SPIN depends on how they are given to the system. When given an LTL formula from the command line, SPIN will check that the *negation* of this formula holds. This is because the formula is directly translated to a PROMELA *never-claim*, which describes a property that should never hold in the model. Another option to declare an LTL formula to be checked is inside the PROMELA code using the keyword `ltl`. Formulas provided this way are checked by SPIN in their

[3] `http://spinroot.com/spin/Man/d_step.html`

Table 4.3. Translation patterns for reactive systems in PROMELA.

	PROMELA

Infinite loop with atomic reaction	`do`
	`:: atomic {`
	`  ...`
	`  }`
	`od`

Set random Boolean	`if`
	`:: (1) -> b = true;`
	`:: (1) -> b = false;`
	`fi`

Set random integer that ranges from MIN to MAX	`select(i : MIN..MAX);`

Set random integer without range assumptions	`do`
	`:: i++;`
	`:: i--;`
	`:: break;`
	`od`

Complete file structure	`// LTL-formula to be checked (if any)`
	`ltl PROP_NAME { X( ORIGINAL_PROPERTY ) }`
	
	`// Variable declarations`
	`...`
	
	`init {  // The main process in PROMELA`
	`  do  // Infinite loop`
	`  :: atomic {  // Atomicity of the reaction`
	`  // Set random inputs`
	`  ...`
	`  d_step {  // Deterministic step`
	`  // Guard declarations`
	`  ...`
	
	`  // Tick logic`
	`  ...`
	`  // After tick logic (set pre-variables)`
	`  ...`
	`  // assert-statement for invariant`
	`  // to be checked (if any)`
	`  ...`
	`  }`
	`  }`
	`  od`
	`}`

Table 4.4. Translation pattern for assumptions in PROMELA.

	PROMELA

Assertion under Assumption	`if`
	`:: (ASSUMPTION) -> assert(PROPERTY);`
	`:: else -> skip;`
	`fi`

positive form. Formulating properties is more natural in a positive form, such that the variant inside the PROMELA code has been used in this thesis.

Invariants to be checked of the original model can be added as assert-statement after the tick logic has been fully computed. The new model checking features for SCCharts use an annotation to add assumptions about the model. System configurations in which these assumptions do not hold have to be ignored in model checking. This can be implemented in the translation to PROMELA by putting the assert-statement in the corresponding branch of an if-statement. This way, the assertion is only reachable when the assumptions are met. The pattern is illustrated in Table 4.4. However, this does not prevent SPIN from visiting all system configurations and thus does not improve performance. It only calls the assert-statement in a subset of the system configurations that are traversed in the explicit model checking approach of SPIN. Thus, system configurations in which the assumption holds are still checked, even though the previous configuration violated the assumption. A pattern to avoid such inconsistent states is initializing a variable *`assumption`* with true and updating it every tick as `assumption &= ASSUMED_PROPERTY`.

The presented approach for translating assumptions to PROMELA only works with simple propositional formulas. For an LTL property an implication can be used to ignore invalid system states, for instance, `(G assumption) -> (G property)`.

4.5 Synchronous Observers in SCCharts

The synchronous observer pattern can be implemented in SCCharts using concurrent regions, which is illustrated in Figure 4.4. One region contains the behavior of the main model, for instance, using referenced SCCharts. A second concurrent region contains the observer model that raises an error flag if undesired behavior has been detected in the main model, optionally under certain assumptions.

Assumptions for a model can be formulated in SCCharts using another concurrent region. These assumptions can then be given to the model checker to rule out counterexamples from inconsistent system configurations. One approach to formulate assumptions is by creating a concrete model of the surrounding target environment. Also thinkable is the definition of properties that define the environment's behavior in an abstract way.

The latter approach is used in the example to formulate the assumption that no error signal is received. Model checking the example shows that the variable *`ok`* stays true in all cases when the assumption holds.

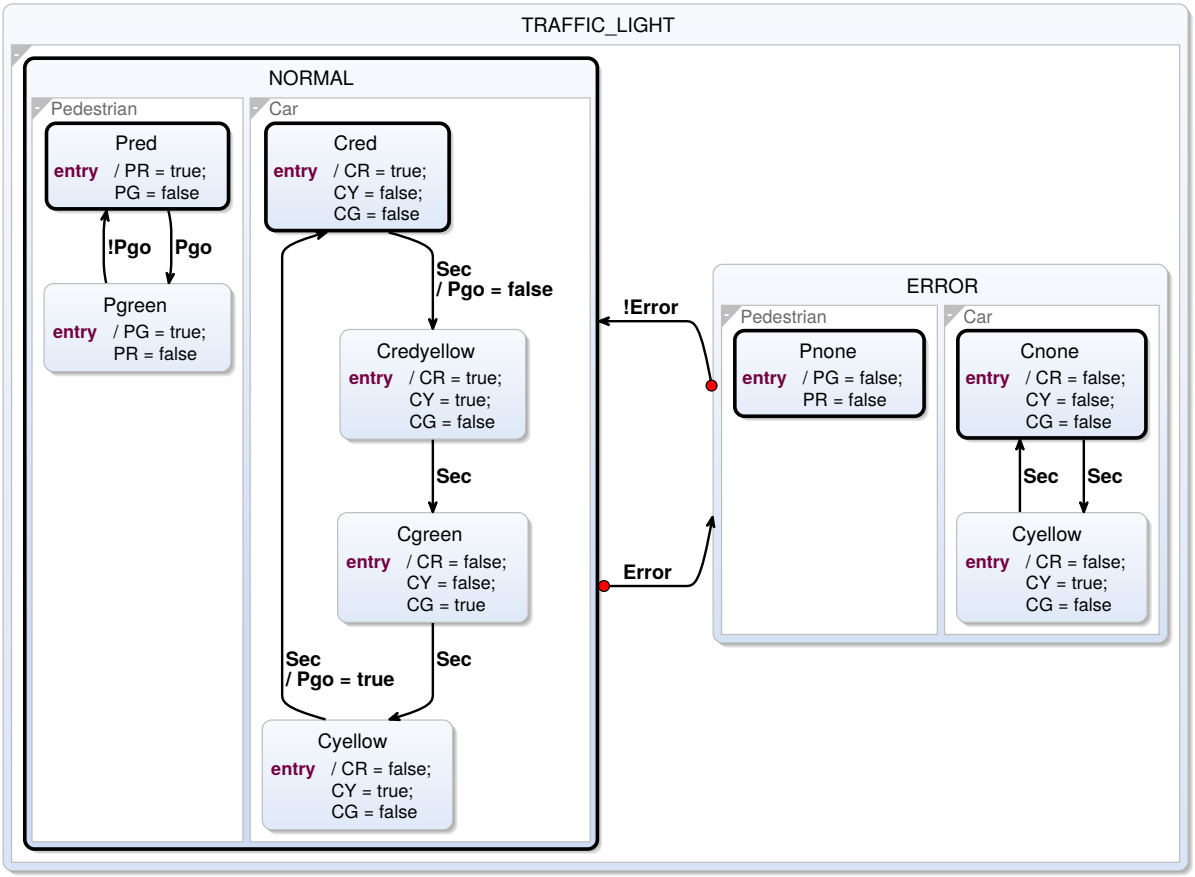


Figure 4.3. A model of a simple traffic light in SCCharts [HLM+12]. As long as no error is received, the lights of cars and pedestrians are controlled normally. Thereby, the variable `Pgo` is set to signal when the cars have red, such that the pedestrian light is set accordingly. As soon as an error is received, it changes its mode, turns off the pedestrian light, and toggles the yellow light for cars.

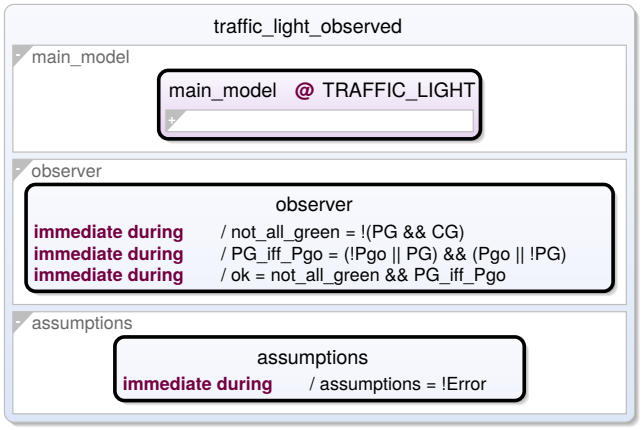


Figure 4.4. The synchronous observer pattern applied to the traffic light model from Figure 4.3. The invariant to be checked is *ok*. An assumed property of the model is that *assumptions* is always true, i. e., that no error signal is received.

Chapter 5

Implementation

The following describes the implementation of concepts discussed in Chapter 4. Thereby, only a subset of SCCharts is considered due to a fixed time frame for this thesis. The basic data types integer and Boolean are allowed in models but no arrays, floating point numbers or strings. Mechanisms to include code in the model from a hosting target environment are not considered. KIELER is open source and the code that has been contributed as part of this thesis is available in the corresponding KIELER repositories.

The two model checkers nuXmv and SPIN have been integrated into the KIELER tool for model checking SCCharts. Therefore, the discussed translation from the sequentialized SCG to PROMELA and SMV have been implemented. The translation to SMV uses the SSA SCG approach because it preserves temporal properties of the original model.

After the model has been translated, the corresponding model checker is started. Next, the model checker's output is parsed to present the result inside KIELER. In case the model checker gives a counterexample, it is translated to the KTrace format, such that it can be re-played on the original model.

Properties to be checked and meta-information about the model (e. g., integer ranges) are added via annotations directly inside an SCChart. An Eclipse view for the new model checking features has been implemented. It contains controls to start the model checking task, present the results and also to start counterexamples if a property fails. This enables a smooth work-flow for model checking. Furthermore, options to configure the code generation as well as model checker arguments are available from the GUI.

5.1 Plug-in Overview

An overview of relevant plug-ins is presented in Figure 5.1.The presented separation in plug-ins fits the existing application architecture, separates UI code from non-UI code, and makes it possible to use the new model checking features for other languages supported by KIELER.

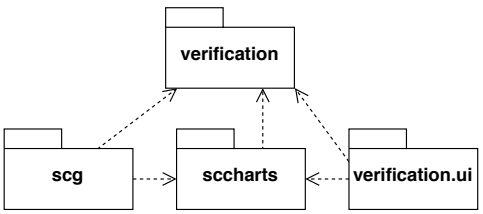


Figure 5.1. Overview of relevant plug-ins. The base plug-in is `verification`. The UI plug-in uses a class of the `sccharts` plug-in to read annotations from models. The `scg` plug-in references the `sccharts` plug-in to implement the translation from SCCharts to the SCG. Similarly, the translation from the sequentialized SCG to SMV and PROMELA has been implemented in the `scg` plug-in.

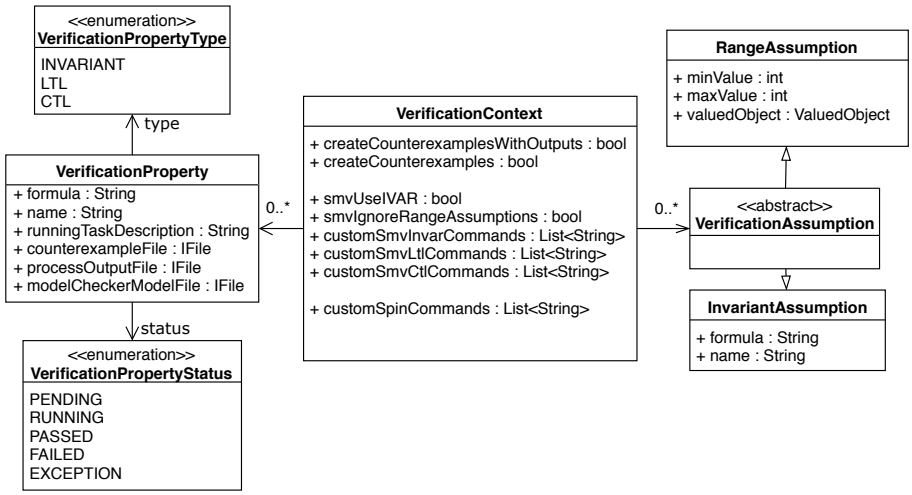


Figure 5.2. Common data structures that hold information for model checking tasks.

Two new plug-ins have been contributed to the KIELER project. The `verification` plug-in contains common data structures and code to interface with the model checkers, whereas the `verification.ui` plug-in contains code that contributes to the GUI. The translation from SCG to PROMELA and SMV has been added to the `scg` plug-in, which is also the host for other already existing code generations, e. g., from SCG to Java. Code specific for SCCharts has been added in the `sccharts` plug-in. This code is for reading annotations in the model to create instances of common data structures. Using abstract data-structures rather-than concrete annotations in the model simplifies the reuse of the implementation for other languages that can be compiled to a sequentialized SCG in KIELER.

5.2 Common Data Structures

There are some common data structures to keep the implementation independent from the concrete SCCharts language and model checker that is used. Important ones are shown as UML class diagram in Figure 5.2. Model checking requires three main inputs, namely the model itself, properties to be checked, and assumptions that are made about the model. In the KIELER compiler the model is represented as an EMF data structure that is processed in a compile chain. `VerificationProperty` and `VerificationAssumption` represent the other two inputs for model checking.

`VerificationProperty` holds the formula to be checked and an optional name for the property. A name is useful when having multiple properties for the same model, which is a common use case. Furthermore, it holds the status of the property, such as *to-be-checked*, *passed*, or *failed*. A counterexample or exception that occurred during the model checking task is also held in this container.

`VerificationAssumption` is defined abstract and is specialized in other classes that provide concrete assumptions of some kind. For instance, a *`RangeAssumption`* has been implemented that holds the minimum and maximum value for integers. Another useful assumption is the *`InvariantAssumption`* that stores a propositional logic formula, which is assumed to always hold in the model.

Instances of these classes are created from Annotations in the model. Listing 5.1 illustrates the use of annotations to augment an SCChart with model checking information. Thus, the three main components for model checking are declared in the model itself. They are handled depending on the concrete model checker to be used. For instance, the `RangeAssumption` is used in SMV as domain for

```
@LTL "G(X(a -> o==1))"
scchart ao_int {
  input bool a
  @AssumeRange 0, 1
  output int o = 0

  initial state waitA
  if a do o = 1 go to gotA

  state gotA
}
```

Listing 5.1. Textual SCChart with annotations for model checking. Starting with the second tick, it will set the output `o` to 1 when the input `a` is true. This is expressed using an LTL annotation. An assumption about the integer range is provided in form of another annotation. Such properties are often trivial for a developer to see but can drastically reduce the resources required for model checking.

Figure 5.3. Compile chain to PROMELA.

the variable, whereas in the PROMELA code it is used as range in a non-deterministic assignment of a PROMELA int.

Another important data structure is the `VerificationContext`, which is also presented in Figure 5.1. It is the container for all information about a concrete model checking run. As such, it contains a list of `VerificationProperty` as well as a list of `VerificationAssumption`. Furthermore, there are fields to fine-tune the code generation. User-defined commands that should be sent to the model checker are also persisted in this class. The `VerificationContext` extends `CompilationContext`, which is the central container for information about a concrete compilation run in KIELER. As a result, the model, the compile chain, and additional configuration and information for model checking is available in the `VerificationContext`.

The presented implementation for model checking is similar to how simulation is handled in KIELER. A `SimulationContext` is set up and started for a new simulation, thereby utilizing the existing compile chain infrastructure of KIELER. The `VerificationContext` has been inspired by this approach.

5.3 Translation to PROMELA

The compile chain that is used in the translation to PROMELA is illustrated in Figure 5.3. It re-uses the translation from SCCharts to the sequentialized SCG that has already been present in KIELER. As a result, only the translation patterns discussed in Section 4.4 needed to be implemented. For instance, Listing 5.2 shows an excerpt from the translation that is used to create the infinite loop, which corresponds to the pattern presented in Table 4.3.

The SPIN syntax for LTL operators uses brackets and angle brackets for *globally*, respectively *eventually*. The expected syntax for these operators in annotations on SCCharts level is `G` for globally,

```
def void generateTickLoop() {
  appendIndentedLine("do") // loop
  appendIndentedLine("::")
  appendIndentedLine("atomic { ") // atomic
  incIndentationLevel()
  appendIndentedLine('''bool «TICK_END» = 0;''')
  generateSettingRandomInputs()
  appendIndentedLine("d_step {") // d_step
  incIndentationLevel()
  generateLocalDeclarations()
  generateSequentialScgLogic()
  generateAfterTickLogic()
  appendIndentedLine('''«TICK_END» = 1;''')
  generateAssertions()
  decIndentationLevel()
  appendIndentedLine("}") // d_step end
  decIndentationLevel()
  appendIndentedLine("}") // atomic end
  appendIndentedLine("od") // loop end
}
```

Listing 5.2. Main method for the translation of the SCG tick logic to PROMELA.

```
def String toSpinLtlFormula(String
    ltlFormula) {
  // Replace with SPIN syntax
  val spinLtlFormula =
    ltlFormula
      .replaceAll('''\bG\b''', "[]")
      .replaceAll('''\bF\b''', "<>")
  // Prepended next operator
  val spinLtlFormulaWithNext =
    '''X( «spinLtlFormula» )'''
  return spinLtlFormulaWithNext
}
```

Listing 5.3. Adapting LTL formulas to SPIN syntax with an initial setup reaction.

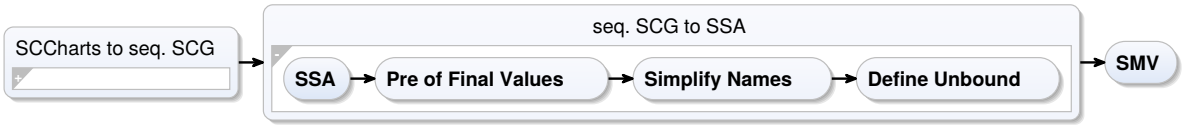


Figure 5.4. Compile chain to SMV.

and `F` for eventually. Therefore, they must be translated to the SPIN syntax. Further, SPIN needs one reaction to enter the tick loop, which is why an LTL formula must be prepended with an initial *next* operator. The code that implements the required changes to LTL formulas is presented in Listing 5.3. The current implementation uses a string replacement. However, this could be enhanced in future work by providing LTL constructs as part of the textual SCCharts grammar. This would enable to do a translation of the operators on the level of an abstract syntax tree.

5.4 Translation to SMV

The compile chain that is used in the translation to SMV is illustrated in Figure 5.4. The first part, namely the translation to a sequentialized SCG, is the same as in the translation to PROMELA and was present in the KIELER tool before this thesis. Similarly, a translation from the sequentialized SCG to an SSA form existed. However, it was not directly fit for a translation to SMV and thus has been adapted as discussed in Section 4.3.

First, a pre-operator always refers to the *last* version of a variable. Thus, a processor has been implemented that replaces references to variables that are used in a pre-operator with the correspond-

ing last version. For instance, consider two sequential assignments to the variable x and reading the previous value of x in between, i. e., `x=1; y=pre(x); x=2`. The SSA transformation will introduce multiple variables for x and y still should be set to the *last* version of x from the previous tick, thus it should be translated to `x1=1; y=pre(x2); x2=2`.

Second, it is easier to work with models that preserve the original names of variables, which is why the last version of a variable is renamed to its original form. In the example above this means that `x2` is renamed to `x`, which results in `x1=1; y=pre(x);x=2`.

Third, there are undefined variables in the resulting SSA form. These implicitly represent the last version of a variable from the previous tick. This relation has been made explicit with an additional assignment to a pre-operator. For instance, `x0=pre(x)`. After these changes to the SSA SCG have been executed, it can be translated to SMV code.

As discussed in Section 4.3.3, there are still mutually exclusive assignments to the same variable in the SSA SCG depending on which path is taken in conditionals. These are expressed in SMV using a single equation with the `case`-construct. For instance, the control flow `if(cond) { x = 1 } else { x = 2 }` can be translated to `x := case cond : 1; TRUE : 2; esac;`. Note, that the else-branch of the if-statement is implemented as the default case in SMV, which is expressed using the condition `TRUE`. Creating a single defining equation for variables has been implemented in two steps. First, the complete SSA SCG is traversed. Thereby, two `HashMap` instances are filled. One instance maps valued objects to their assignment-nodes, whereas the other instance maps assignment-nodes to their corresponding conditionals. Afterwards, the keys of the first HashMap are iterated over, which are the valued objects to be defined. If there is only a single assignment-node in the `HashMap` for this key with no associated conditionals then it can be written without `case`-construct. Otherwise the `case`-construct is used and the conditionals that lead to the corresponding assignment-nodes are fetched from the second `HashMap`.

General assumptions about the model can be added in SCCharts using an annotation `@Assume`. These annotations will be translated to an invariant in the SMV code. Such invariants are specified in SMV using the keyword `INVAR`. For instance, when having two Boolean inputs `a` and `b` then the annotation `@Assume "a != b"` could be used to declare that these two inputs will never have the same value. The annotation will be translated to the SMV model as `INVAR a != b;`. Range assumptions on the other hand are translated to the domain of integer variables. For instance, an annotation `@AssumeRange 0, 255` for a variable `x` will be translated to `x : 0..255;` in the SMV module.

In contrast to SPIN, the syntax for the LTL operators *globally* and *eventually* is G, respectively F in SMV. Thus, these do not need to be adapted. However, expressions in SCCharts and PROMELA use a C-like syntax. For instance, equals is written as `==` and logical conjunction is written as `&&`. In SMV such expressions are written using a single character, i. e., `=` and `&` respectively. Listing 5.4 shows a code excerpt that adapts expressions to SMV syntax. A string replacement takes care of adapting the expressions taken from annotations inside the original SCChart to the expression syntax of SMV. This could be improved by extending the SCCharts grammar with constructs for model checking, e. g., LTL expressions. The translation to SMV could then be done on the level of an abstract syntax tree. Further, the Boolean literals `true` and `false` are written in upper-case in SMV. As a result, these also need to be adapted.

5.5 Interfacing with the Model Checkers

After the original model has been translated to SMV or PROMELA, the model checker needs to be executed. Both nuXmv and SPIN are command line tools. As such, running the model checkers can be done manually from the console. However, for this thesis communication with the model checkers

```
def String toSmvExpression(CharSequence kexpression) {
  val result = exp.toString
    .replace("==", "=").replace("&&", "&").replace("||", "|")
    .replaceAll('''\bfalse\b''', " FALSE ")
    .replaceAll('''\btrue\b''', " TRUE ")
    .replace("%", " mod ")
  return result
}
```

Listing 5.4. Method that adapts expressions to SMV syntax using a string replacement. Note that *`replace`* and *`replaceAll`* both replace all occurences of the given pattern. The difference is that `replaceAll` uses a regular expression as pattern, whereas `replace` uses a simple String.

has been integrated in the KIELER tool. Therefore, the model checking process is started with suitable arguments and communication with it is done using *stdin* and *stdout*.

NUXMV and SPIN require different commands and interactions to perform model checking. For instance, in SPIN counterexamples are saved to a separate file. A counterexample can be printed in different variations by issuing corresponding commands to the SPIN model checker. In contrast to this, NUXMV prints counterexamples directly to stdout. Further, model checking using SPIN can be started from a single command that contains all relevant options. This is also true for NUXMV in some cases. However to access all options and algorithms that NUXMV provides, it is necessary to issue commands sequentially in a shell-like interface.

Communication with the model checkers has been implemented in the classes `RunSpinProcessor`, respectively `RunNuxmvProcessor`. These implement general processing units of the KIELER compiler infrastructure. This way running the model checkers is a general building block that can be integrated in a compile chain inside KIELER. For instance, a compile chain can start with the original SCCharts model as input. The following compilation units will transform the model to the sequentialized SCG. This is the input for the PROMELA code generation. The resulting PROMELA code is the input to the `RunSpinProcessor`, which will save the code in a suitable place, start the SPIN model checker, parse its output and issue events according to the result. The central configuration unit for such a model checking task is the `VerificationContext` as explained in Section 5.2.

It is assumed that NUXMV and SPIN have been added to the `PATH` environment variable to start the model checking process from within KIELER. The implemented command to start the NUXMV process is *`nuXmv -int SMVFILE`*. The `-int` option will start NUXMV in *interactive mode*. In this mode, further commands can be issued to configure the tool and choose which model checking algorithm to be executed. Figure 2.5 gives an overview of available commands. The commands that are sent to NUXMV in interactive mode are configured in the `VerificationContext`.

The implemented command to start the SPIN process is *`spin -run PMLFILE`*. The `-run` argument will prepare and perform a complete verification of the given PROMELA file. Additional arguments can be configured in the `VerificationContext`. They will be added after `-run` and before the path to the PROMELA file. Arguments that can be set this way include `-bfs` (use breadth-first search) and `-m100000` (sets the maximal search-depth to 100000).

The model checking process that is started from within KIELER can be canceled from the graphical user interface. When the user clicks the corresponding button, a flag is set to cancel following compilation units of the currently executed compile chain. Furthermore, if the model checking process is already running it will be stopped. Such an option is important as model checking can be costly with respect to time and memory. In a first implementation, it has been found that the default API of

Java 8 on Linux was not reliably killing the process together with all sub-processes. Instead, the model checking process continued to use RAM and CPU time as an orphan. As a result, killing the process on Linux was implemented in three steps. First, the ID of the process that was started from within KIELER is identified. Second, the IDs of child processes are identified recursively. Third, processes with the corresponding IDs are terminated. The first step can be done using Java-API and reflection. The second and third step have been implemented by issuing common Linux system commands, namely `pgrep -P PARENTID` to find the ID of child processes and `kill ID` for killing the process. The implementation can be found in the class `ProcessExtensions`. On other operating systems, the standard Java API is used to kill processes.

5.5.1 Parsing Counterexamples

In this thesis, SCCharts are translated to a language that is supported by model checkers. As a result, any counterexample from the model checkers will be in terms of the translated model. Presenting the counterexample in terms of the original model is an important step to get a practical work-flow because it is the original model that users work with. Therefore, the output of the model checker is parsed and counterexamples are translated to the KTrace format from Section 2.1.3. This way the counterexample from a model checker can be replayed on the original SCCharts.

A counterexample for SCCharts corresponds to a list of system reactions. It can contain a loop marker to indicate repeating system reactions that violate a liveness property. A data-structure to store such counterexamples has been implemented. It contains a list of key-value maps. Each key-value map holds the variable-value pairs for a single system reaction. Further, the data-structure can store the index of the reaction that marks the beginning of a loop.

Listing 4.3 shows a counterexample from nuXmv. Parsing the model checker output can be done line-by-line using regular expressions. nuXmv clearly states whether a property is true or false. Parsing the counterexample is started in case a false property is reported. In this case `-> State:...<-` indicates the beginning of a new reaction. Thus, a new key-value map will be added to the counterexample data-structure when such a line is found. A line containing `Loop starts here` indicates that the next reported reaction is the first of the loop. The loop index in the counterexample data-structure is set accordingly. Finally, the current value of a variable is reported by nuXmv as `VARIABLE = VALUE`. Thus, when finding such a line, the corresponding pair will be added to the key-value map of the counterexample data-structure.

SPIN does not print its counterexample directly to stdout. Instead, it saves it in a separate file called *trail-file*. Using specific SPIN commands, such a trail can be replayed or printed to stdout in different levels of detail. Parsing the SPIN output is done line-by-line using regular expressions. When it is found that a trail-file has been created then a second SPIN process is started that prints the counterexample to stdout. Therefor, the model checker is called as `spin -t -p -g PMLFILE`. The argument `-t` is used to printout a trail-file, `-p` is used to print the statements that were executed, and `-g` is used to print the changed values of global variables after an executed statement. Note that inputs and outputs of the original SCCharts are declared globally in the generated PROMELA code. Thus, the values of locally defined variables in PROMELA are irrelevant from an SCCharts perspective.

To separate the counterexample into discrete reactions of the original model, a special variable `pmltickend` is added to the PROMELA model during translation. It is set before and after each reaction to indicate its start, respectively end. An example of the resulting PROMELA code is given in Listing 4.8.

The data-structure for storing counterexamples is filled from the output that SPIN prints for the trail-file. A change of `pmltickend` from `1` to `0` and its very first assignment to `0` indicate a new system reaction in the counterexample. A new key-value map is created in the counterexample data-structure

```
1 i = true => o = true ;
2 i = false ;
3 loop_start:
4 goto loop_start;
```

Listing 5.5. KTrace counterexample for Listing 4.3

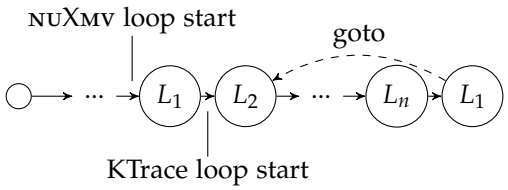


Figure 5.5. Schematic of a counterexample from nuXmv with n different looping states. nuXmv prints the first state of the loop again at the end of the counterexample. The loop-start in the KTrace format is adjusted accordingly.

when a new reaction is indicated this way. When an assignment to a variable is found then the variable-value pair is added to the current map. SPIN will print `<<<<<START OF CYCLE>>>>>` to mark the beginning of a loop in the counterexample. When this marker is found then the index in the data-structure is set accordingly.

The filled counterexample data-structure can be printed in the KTrace format with little effort. Listing 5.5 shows a generated KTrace. Each entry in the list of the counterexample data-structure is printed as a tick in the KTrace format, which is *INPUTASSIGNMENTS* => *OUTPUTASSIGNMENTS*;. The separator => is omitted when there are no changed outputs in this tick. Recorded variable-value pairs are printed as *VARIABLE* = *VALUE*. Thereby, the syntax of *TRUE* and *FALSE* from SMV has to be adapted to *true* and *false* in KTrace.

Any variables that are neither input nor output of the original SCChart are omitted in the KTrace because they are not necessary to understand and replay the counterexample on the original high-level model. Low-level variables such as guards, will be recomputed when providing the same inputs to the model. This follows from the deterministic nature of the synchronous semantics.

The presented way of obtaining counterexamples from the model checkers will only record variables that have been changed from the last reaction to the current. This matches the semantics of the KTrace format such that no additional information has to be added.

A loop-label will be added to the KTrace file when the loop-start index is set in the counterexample data-structure. More specifically, a loop-label is prepended to the corresponding tick and a goto-statement to this label is added at the end of the KTrace. Thereby, an important detail in nuXmv counterexamples has to be considered. nuXmv repeats the first state of a loop at the end of the counterexample. The loop-start label in KTrace format has to be positioned accordingly. This is illustrated in Figure 5.5.

5.5.2 Generated Files

Several files are created during a model checking task. First, the model to be checked is saved in a PROMELA or SMV file. Second, the output of the model checker is saved. This is not needed most of the time because the results are presented within KIELER. Nonetheless, it can be useful in debugging and understanding the tools. Third, a counterexample will be created in KTrace format if necessary. Further files can be created by the model checker itself. For instance, SPIN saves counterexamples in a separate trail-file.

Files that are generated by KIELER when doing a simulation are stored using a common file structure. More specifically, generated files are placed inside the folder *kieler-gen*, which is created separately for each project.

There are several advantages to this approach.

▷ It keeps logically connected files in one place.

▷ It indicates why and when files have been generated.

▷ KIELER developers can view and check their content to get a better understanding of the tooling.

As a result, this approach has also been used for files that are created as part of the model checking task. They are placed in a sub-folder called *verification* inside `kieler-gen`. Thereby, a naming scheme for files is used to avoid name conflicts.

The base name of generated files is taken from the original model to be checked. For instance, the `ao-int` model would result in SMV and PROMELA files named `ao-int.smv` and `ao-int.pml` respectively. Output of model checkers is indicated by the suffix *.log*. However, a model can contain multiple properties. Thus, the name of the checked property is added to the corresponding output file name to make it unique. Additionally, white-space and special characters are replaced in the file name, such that only alphanumeric letters and dots remain.

In conclusion, the output of NUXMV for the property "o is a boolean" in `ao-int` will result in a file called `ao-int-o_is_a_boolean.smv.log`, which is saved in the `verification` sub-folder inside `kieler-gen`.

5.6 Automated Tests

The new model checking features presented in this thesis are tested automatically using the test infrastructure of KIELER. JUnit is used for automated tests in KIELER. A custom JUnit `TestRunner` is available that searches for models (e. g., SCCharts) in a repository and executes tests with these. Metadata about the models is stored alongside them in the repository. This metadata is used, for instance, to filter the models that should be used in a specific test run.

To test the new model checking features, this framework has been reused. Models to be tested have been marked as such inside the models-repository. An abstract JUnit test class has been written that performs a model checking task and compares the actual result with the expected result for the current property. This abstract test class is the bases for concrete test classes that perform model checking of SCCharts using NUXMV and SPIN. A `VerificationContext` is configured depending on the concrete test case.

There are tests for both, properties that must fail and properties that are true. Each test has a time-out that will cancel the model checking task and fail the current test in case the time is exceeded. This is useful as the tests are run frequently on a test server and are potentially costly in case of a bug.

The test performs a complete model checking task, starting with reading properties and assumptions from the model, translating it to SMV and PROMELA, running the corresponding model checker and comparing the actual and the expected result. Thus, the core features discussed in this thesis for model checking SCCharts in KIELER are tested.

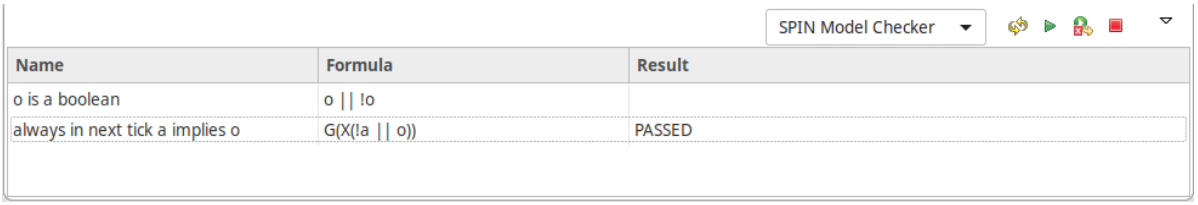


Figure 5.6. Screenshot of the model checking view. The properties and the model checking result are presented in a table. In the screenshot, SPIN is currently selected as back-end for model checking. From left to right the toolbar buttons are to *reload properties and assumptions*, *start model checking*, *replay a counterexample*, *stop model checking*. The small down arrow at the right-hand side opens a menu with further options.

5.7 Graphical User Interface

The code that contributes to the UI of KIELER has been added in the plugin `verification.ui`. It contains an Eclipse view for model checking. The view is shown in Figure 5.6. The toolbar contains a drop-down menu to select the model checking back-end. Toolbar buttons provide access to most commonly used functionality. These are loading properties and assumptions from a model, starting or stopping a model checking task, and re-playing a counterexample of the selected property.

Furthermore, a context menu provides options to define how a `VerificationContext` will be configured for a concrete verification run. For instance, there are menu items that open dialogs to configure the commands for NuXmv and SPIN.

An observer pattern (often referred to as *listener* in Java) has been implemented on top of `VerificationProperty` to update the view dynamically when the status of a property changes. More specifically, the event `VerificationPropertyChanged` is issued as part of the model checking task to notify listeners about its progress. The graphical user interface implements such a listener to update the status of a property in the *`Result`* column.

Chapter 6

Evaluation and Experience Report

The following presents experiences with NuXmv (version 1.1.1) and SPIN (version 6.4.8) on different models that have been created from SCCharts using the translations discussed in Chapter 4. First, general observations are presented in Section 6.1. This includes strengths and weaknesses of the model checkers. Furthermore, characteristics of models are illustrated that can affect the performance when doing model checking. Afterwards, Section 6.2 explains the setup that has been used to evaluate options and algorithms of the model checkers. The results of this evaluation are presented in Section 6.2.3.

6.1 General Observations

NuXmv can detect division-by-zero errors. Thereby, it states the problem and line where it occurred. In contrast to this, SPIN crashes with a floating point exception when encountering a division-by-zero and no information in which line the error occurred is given. This has been experienced when model checking the SCChart from Listing 6.1. However, NuXmv also gives a division-by-zero error for this listing when the variable `x` is initialized to 1. In this case no such error will occur when using SPIN. NuXmv on the other hand seems to do an analysis in which the *potential* for a division-by-zero is detected and as a result an error is returned. Changing the transition in the example to `if x > 0 do y = (10 / x) go to init` still results in a division-by-zero error from NuXmv.

```
@Invariant "y <= 255"
scchart div_by_zero {
  @AssumeRange 0, 255
  output int x=0, y = 0

  initial state init
  do y = (10 / x) go to done

  state done
}
```

Listing 6.1. Textual SCChart that performs a division-by-zero.

```
int a;
bool o;

init {
    do
    :: a++;
    :: a--;
    :: break;
    od;

    if
    :: (a == 0) -> o = 1;
    :: else -> skip;
    fi;

    assert(!(a==0) || o);
}
```

Listing 6.2. PROMELA model exploring the full range of an int.

The BDD construction in NuXmv can detect range violations of integers. This happens for example, when the go-command is processed. Checking the ranges of integers this way can be useful to verify the sufficiency of range assumptions in the original SCChart. However, NuXmv can give false errors when the integer range is not trivially bounded. Consider the following transition that increments an integer: `initial state S0 do x++ go to S1`. NuXmv will give an error for the resulting SMV code when processing the go-command even though this transition can only be taken once. However, it works when adding a range check before doing the increment. For example, when x is an unsigned 8 bit integer: `initial state S0 if x < 255 do x++ go to S1`. Here, the condition `x < 255` will result in SMV code that gives no out-of-range error when doing the BDD construction. BMC algorithms of NuXmv do not abort model checking when encountering an out-of-range violation. Instead, a warning is given. As a result, these algorithms can also handle the first transition without range check.

Big integers can affect the time required for model checking in NuXmv. This is also the case when using an algorithm that can handle unbounded integers (e. g., IC3 with -i option). For instance, changing the range assumption of the output in the ao-int model of Listing 5.1 to @AssumeRange 0, 10000 will increase the time required to do model checking considerably when using IC3 with the -i option. On the other hand, omitting the range assumption (such that an unbounded integer is used in the SMV model) and using the same algorithm will perform model checking in fractions of a second.

The explicit model checking approach of SPIN has difficulties when the full state-space of an integer must be explored. The unbounded non-deterministic selection of an int in PROMELA creates such a state-space. This has been modeled in Listing 6.2. A similar model will be created by the translation presented in this thesis when using an integer without range annotation as input in SCCharts. In general, when using SPIN it is better for model checking performance to declare the bounds of input integers in an SCChart as small as possible via a range annotation. However, on output and internal integers they do not have an effect when doing model checking with SPIN because the explicit model checking approach will traverse all reachable values. In contrast to this, the translation to SMV also uses the range assumptions on output and internal integers to declare their bounds.

An interesting behavior can be observed when specifying invariants that are false in the initial tick. In this case, there is no system configuration that satisfies the assumption, such that a checked property is trivially true. For instance, consider the assumption `pre(cond) && cond`. The presented translation will initialize the pre-variable with false, such that this assumption is always false in the initial tick. Now, when checking a model with this assumption using NuXmv, every property will trivially be true. This is because the set of system configurations in which the assumption has always been true is empty, and thus all of them satisfy the property to be checked. The model checker will give no warning in this case. As a result, care must be taken that specified assumptions do not rule out all possible system configurations. This can be achieved by checking a false property that is known to fail. The translation to PROMELA handles assumptions differently as discussed in Section 4.4.3. SPIN will ignore system configurations in model checking that do not satisfy the assumption. In the example above this means SPIN will ignore the system configuration for the initial tick but could check the property in following ticks. To specify a proper invariant assumption for SPIN, the formula must be adapted to stay false when it has been false once.

6.1.1 Model Checker Integration

This thesis integrates the model checkers NuXmv and SPIN in the KIELER development environment for SCCharts. This is a different use case than the manual usage. The model checking output is parsed to present the results within the IDE, such that the back-end for verification is hidden from the user. Thus,

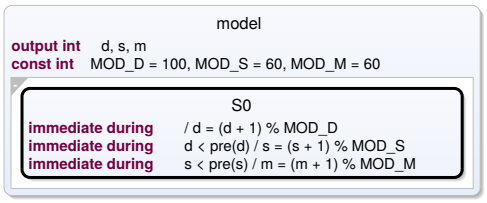


Figure 6.1. An SCChart implementing counting. The variable `d` is increased in every tick, whereas `s` counts the falling flanks of `d` and `m` counts the falling flanks of `s`.

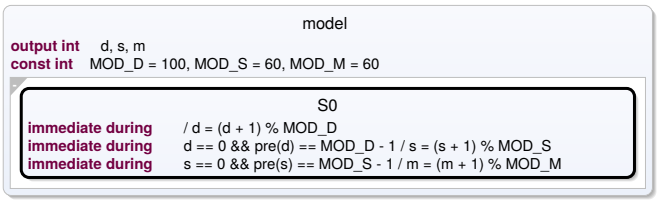


Figure 6.2. The model Figure 6.1 re-implemented using a comparison with constants instead with another variable. This model has better performance when doing model checking.

a regular and unambiguous output from the model checker is useful. In this thesis, understanding and parsing the output of SPIN has been perceived as more challenging than the output of nuXmv.

nuXmv gives a clear statement whether a property is true, or false, or could not be proved. Furthermore, a counterexample is presented in a way that reflects discrete reactions and only changed variables in a reaction are printed. This fits the KTrace format of KIELER and allows for simple yet complete parsing rules to present relevant model checking results to the user. When a property is not clearly stated as true by nuXmv, then it will not be presented as such to the user.

The output of SPIN on the other side only gives a clear statement when a property fails. In this case the name of a trail file is printed, which contains the counterexample. However, a passing property or a property that has not been fully verified is not clearly stated as such. This information must be taken from the context. For instance, the -bfs option of SPIN will not perform a detection of acceptance cycles, which means that a liveness property will not be disproved. The information of an incomplete liveness check is printed in the output as *`cycle checks - (disabled by -DSAFETY)`*. However, there is no statement about the property and that the -bfs option implies the -DSAFETY option. Furthermore, when the SPIN process terminates forcibly (e. g., because of a division-by-zero error), then the output may not contain the name of a trail file and thus may wrongly be interpreted as a true property. Thus, the SPIN output must not only be parsed for a statement about a created trail file, but also that it corresponds to a successful and complete verification.

Besides this, the output for a counterexample from SPIN is not given in discrete reactions of the original SCChart. This makes sense because PROMELA is not dedicated to synchronous systems. As a result, from an SCCharts perspective a SPIN counterexample contains more irrelevant information than a nuXmv counterexample.

6.1.2 Model Characteristics that Affect Performance

Some features of models have been found to affect the performance when doing model checking using nuXmv. First, the comparison of variables with constants can yield better performance than comparing with other variables. For example, consider the SCChart in Figure 6.1. This model compares a variable with its previous version to detect falling flanks (`d < pre(d)`). In contrast to this, Figure 6.2 is doing a comparison with constants to detect the falling flank. Note that both versions use the pre-operator but at different places. It has been found that the IC3 algorithm with `-i` option can check the property `m <= 60` in Figure 6.2 within fractions of a second, whereas the version that uses the pre-operator requires minutes.

It seems that the constant can be used by the algorithm to determine the bounds of the integers, which greatly improves model checking performance. This information seems to be missing in the

Table 6.1. Statistics about the tested models. The abbreviations stand for *Lines of Code* (LoC), *global variables* (GV), *local variables* (LV). Global variables in PROMELA are the ones defined outside of any process, whereas for SMV models it refers to the number of variables in the `VAR`-block. This includes inputs and the GO-signal. Local variables are the data-flow guards from the sequentialized SCG and the `pmltickend` variable. *Defines* refers to the number of symbols in the `DEFINE`-block of SMV.

	PROMELA			SMV		
Model	GV	LV	LoC	GV	Defines	LoC
ABRO	8	16	90	8	20	61
Counting 2 ints	9	9	77	9	18	66
Counting 3 ints	13	13	102	13	27	92
Counting 6 ints	25	25	177	25	54	170
DVD Player	43	142	523	33	226	465
Chrono	22	49	226	21	104	260
TLCS constant wait-time	22	52	270	22	152	407
TLCS variable wait-time	30	52	286	30	176	495
UMS	176	152	1090	127	545	1474

model checking process when comparing with other variables. As a result, comparisons of integer variables should be done with constants instead of variables when possible.

Second, the number of integer variables affects the performance. This is true even in the case where an integer always holds the same value when it is used, such that it could be declared as constant. This has been experienced in the evaluation of the TLCS, which is described in Section 6.2. Two versions of this model have been evaluated. The first uses a constant, whereas the second uses a normal integer. The increased number of integers in the second model resulted in a timeout with all evaluated nuXmv commands. The explicit model checking approach of SPIN was not affected as much by this change.

As a result, the number of integers should be kept small in models and constants should be declared as such to avoid unnecessary complexity when doing model checking.

6.2 Comparison of Algorithms

Different options and algorithms of the model checkers SPIN and nuXmv have been tested on models that have been created from SCCharts using the translation presented in Chapter 4. The following first explains the models and test setup that were used. Each model was tested with one invariant and one LTL property. Statistics of the models are presented in Table 6.1. The comparison results are presented in Section 6.2.3. The UMS model has been re-created in SCCharts from a Lustre model such that the results for the UMS are especially suited for a comparison with Lesar. This is done in Section 6.2.4.

6.2.1 Tested Models

The following discusses the models that have been used for the evaluation of different model checking options. Further, a true invariant and LTL property is given for each model. These are the properties that have been verified by the model checkers.

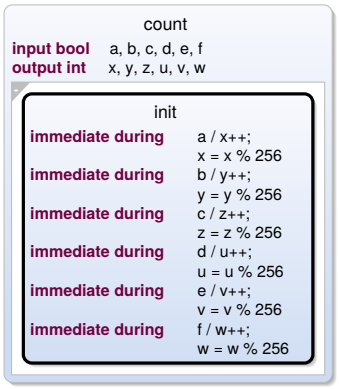


Figure 6.3. An SCChart that counts concurrently from 0 to 255 with 6 integers.

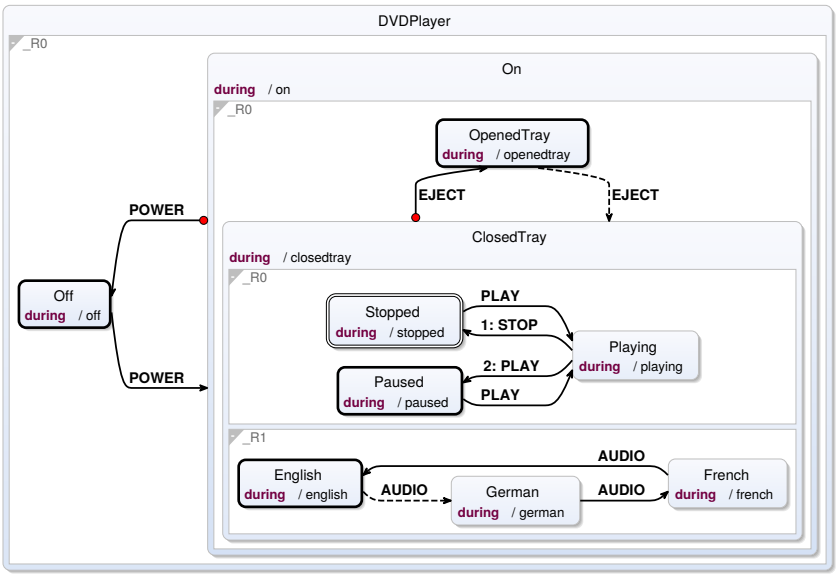


Figure 6.4. The DVD-Player example adapted from the work of Fuhrmann [Fuh11].

ABRO

The ABRO model is shown in Figure 2.1. It is a well-known synchronous program that is often used to illustrate concurrency, hierarchy and strong aborts. The model waits concurrently for the inputs `a` and `b` to be received. When both have been received then the output `o` is set to true. The input signal `r` is used to reset the model, such that `o` is (re-)set to false and `a` and `b` must be received again before the output is set to true. ABRO has a very limited state-space such that a model checking solution for SCCharts should handle this easily.

▷ Invariant Property: `r -> !o`

▷ LTL Property: `G X ((a && b && !r) -> o)`

Counting 8 Bit Integers in Parallel

Models with increasing state-spaces have been tested by incrementing integers in parallel. More specifically, it is counted from 0 to 255, which is the range that can be represented by 8 bit. This has been done with 2, 3 and 6 integers. Figure 6.3 shows the version for 6 integers. Note that only one variable is relevant for the tested property.

▷ Invariant Property: `x <= 255`

▷ LTL Property: `G (x <= 255)`

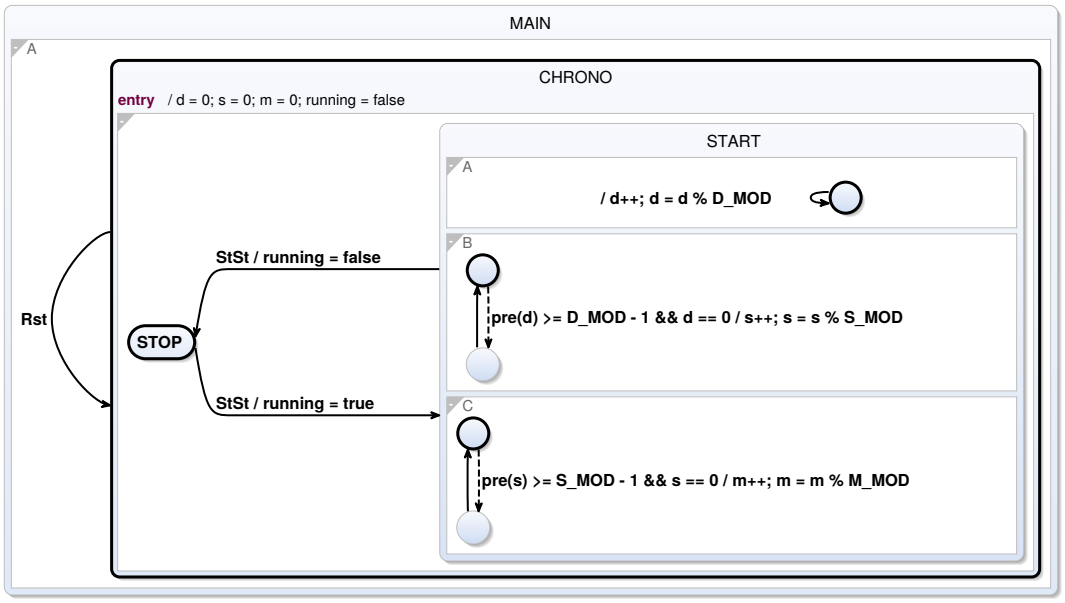


Figure 6.5. A simple example for a chronometer that counts seconds and minutes.

DVD-Player

The DVD-Player presented in Figure 6.4 has been taken from the SCCharts website[1] and is based on an example from Hauke Fuhrmann [Fuh11]. The model starts in the state `Off` and switches to `On` when the `POWER` button is pressed. A second press of the `POWER` button will transition back to `Off`. There is further behavior inside the `On` state. The tray can be opened and closed. In the closed state, the inputs `PLAY` and `STOP` control the playback by switching between `Paused`, `Stopped` and `Playing`. Further, the language can be toggled with the input `AUDIO`. During actions emit signals depending on the current state.

The DVD-Player model uses aborts and during actions, which are extended SCCharts features. Further, there are multiple hierarchy levels. The model does not make use of integers. However, different states in which the control-flow can stay, the hierarchy and extended features result in an SCChart that is more complex than the ABRO example.

▻ Invariant Property: `!(on && off)`

▻ LTL Property: `G ((on && POWER) -> off)`

Chrono

The Chrono model is shown in Figure 6.5. This model counts seconds and minutes. Thereby, a second is increased every 100 ticks that pass in the `START` state, which is counted in the variable `d`. The constants `S_MOD` and `M_MOD` are set to 60, whereas `D_MOD` is set to 100 to achieve the desired counting behavior. Two inputs are used to control the model, namely `Rst` for *reset* and `StSt` for *start*, respectively *stop*. The Chrono model has an increased state-space because it counts with integers sequentially.

▻ Invariant Property: `Rst -> (s == 0 && m == 0)`

[1] `https://sccharts.com`

▷ LTL Property: `G (Rst -> (s == 0 && m == 0))`

Traffic Light Control System

A Traffic Light Control System (TLCS) has been modeled based on a paper by Yu et al. [YDT14]. The authors describe it as an example for a "simple but practical TLCS" to demonstrate their model checking technique. The model is for a crossing of roads in south-north and east-west direction. It has two modes: one normal mode, and one for the rush-our, which requires slightly different pauses between light phases. The description of the system in the paper is as follows:

> (1) The system starts at 0 o'clock and the next step is (2);
>
> (2) The mode of the system is set as 0. The green light of the east-west direction and the red light of the south-north direction are on. This state lasts 25 seconds and the next step is (3);
>
> (3) The yellow light of the east-west direction flashes and the red light of the south-north direction is on. This state lasts 5 seconds and the next step is (4);
>
> (4) The red light of the east-west direction and the green light of the south-north direction are on. This state lasts 25 seconds and the next step is (5);
>
> (5) The red light of the east-west direction is on and the yellow light of the south-north direction flashes. This state lasts 5 seconds. According to the current time, the mode is set for the next state. If the current time is between 7 o'clock and 9 o'clock or between 17 o'clock and 19 o'clock, the next step is (6), otherwise the next step is (2);
>
> (6) The mode of the system is set as 1. The green light of the east-west direction and the red light of the south-north direction are on. This state lasts 30 seconds and the next step is (7);
>
> (7) The yellow light of the east-west direction flashes and the red light of the south-north direction is on. This state lasts 5 seconds and the next step is (8);
>
> (8) The red light of the east-west direction and the green light of the south-north direction are on. This state lasts 20 seconds and the next step is (9);
>
> (9) The red light of the east-west direction is on and the yellow light of the south-north direction flashes. This state lasts 5 seconds. According to the current time, the mode is set for the next state. If the current time is between 7 o'clock and 9 o'clock or between 17 o'clock and 19 o'clock, the next step is (6), otherwise the next step is (2).

The SCCharts that have been created based on this description are shown in Figure 6.6. The modeled SCCharts make use of referencing features that are implemented similar to a macro-expansion during compilation. The encapsulation of a single light-phase makes it possible to add further phases with little effort. The graphical syntax of SCCharts helps to visualize the textual specification from the paper.

This model has been tested twice in the evaluation of this paper with a subtle difference. The first version declares the `waitTime` in the `lightPhase` model as a constant. As a result, occurrences of this variable will be replaced during compilation with the constant literal that it is bound to. In the second version, the integer is not explicitly marked as a constant, such that local variables will be set to the bounded parameter when entering the light-phases in the main model. The evaluation shows that this makes a significant difference in the model checking performance when using nuXmv but not as much when using SPIN.

▷ Invariant Property: `!(snG && ewG)`

▷ LTL Property: `G F (ewG)`

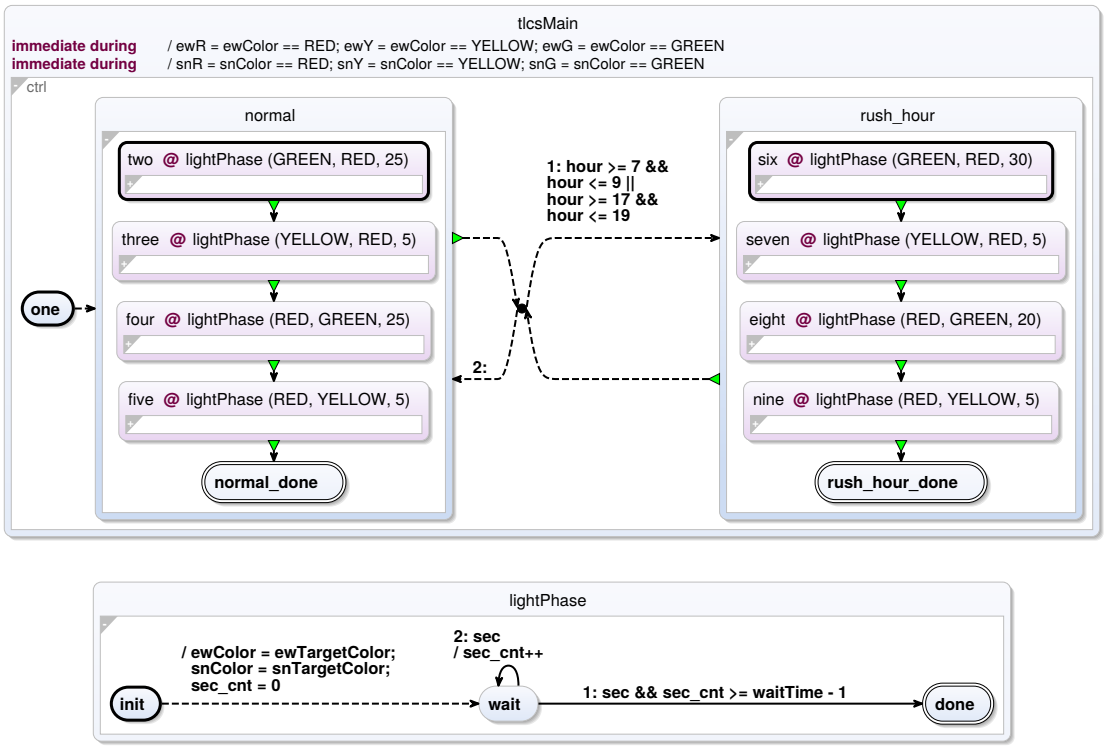


Figure 6.6. SCChart for the TLCS described by Yu et al. [YDT14]. The main model references the SCChart for a single light-phase multiple times with different arguments. The first argument is `ewTargetColor`, the second argument is `snTargetColor`, and the third argument is `waitTime`. The variables `sec` and `hour` are inputs to the model. `sec` is assumed to be true once per second, whereas `hour` is set to the current hour of the day ranging from 0 to 23. For simplicity, `sec` has been assumed to be true in every tick in this evaluation.

U-turn Management System

A U-turn Management System (UMS) for trains is used in a paper by Halbwachs et al. to demonstrate model checking with Lesar [HLR92]. Thereby, the synchronous observer pattern is used to formulate the properties to be checked and assumptions about the environment. A schematic of the UMS is illustrated in Figure 6.7. When a train enters through segment A then the switch S is set to connect it with segment B. As soon as the train arrives there, the switch is adjusted to connect it with segment C, such that the train can leave this way.

The UMS model has been re-created in SCCharts to compare the model checking features introduced in this thesis with Lesar. Lustre is a data-flow language such that the SCCharts version makes heavy use of data-flow regions. These are an extended feature in KIELER SCCharts. Each equation in a data-flow region is translated to its own concurrent control-flow region during compilation. Dependencies between the equations are thus handled by the IUR-protocol and the sequentially constructive MoC as described in Section 2.1.1.

The UMS itself consists only of 6 equations with no references to external modules. This model is presented in Listing 6.3. However, the synchronous observer to formulate its properties and assumptions about the environment add to the complexity of the final model that is given to the model checker. As a result, it is the largest model that has been used for the evaluation of this thesis. The statistics presented in Table 6.1 illustrate its complexity.

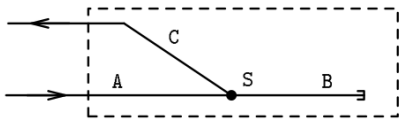


Figure 6.7. Schematic of a U-turn area for trains by Halbwachs et al. [HLR92]. A train enters through A, moves to B, and leaves through C.

```
1  scchart ums {
2    input bool on_A, on_B, on_C, ack_AB, ack_BC
3    output bool grant_access, grant_exit,
       do_AB, do_BC
4    bool empty_section, only_on_B
5
6    dataflow:
7    empty_section = !(on_A || on_B || on_C)
8    only_on_B = on_B && !(on_A || on_C)
9    grant_access = empty_section && ack_AB
10   grant_exit = only_on_B && ack_BC
11   do_AB = !ack_AB && empty_section
12   do_BC = !ack_BC && only_on_B
13 }
```

Listing 6.3. The UMS model implemented in SCCharts using a data-flow region.

The final property to be checked is called `ok`. It is the conjunction of the properties discussed in the Lustre paper, namely `no_collision`, `exclusive_req`, `no_derail_AB`, `no_derail_BC`. Similarly, the assumptions about the environment are expressed in a single Boolean by combining separate assumptions via logical AND.

▷ Invariant Property: `ok`

▷ LTL Property: `G (ok)`

6.2.2 Test Setup

The described SCCharts have been translated to SMV and PROMELA. These models were then checked by nuXmv and SPIN using different options. Thereby, the required time and memory for model checking have been measured using *GNU time* (`/usr/bin/time` on Ubuntu). A timeout of 2 minutes was set, such that the model checking process was killed after this duration. The fastest model checking algorithms finished verification within fractions of a second, such that a timeout of 2 minutes can be considered reasonable for the given models. The evaluation was done on server hardware with more than 40GB of RAM and 2.53 GHz CPUs. Thereby, only a single core has been used for the model checking process.

PROMELA and SMV models differ in their MoC. Further, different translation approaches have been used in their creation. For instance, the PROMELA code is generated without translating the sequentialized SCG to SSA form. As a result, no direct conclusion of the general model checking capabilities of nuXmv and SPIN can be deduced from this evaluation. However, the goal is to find a suited model checker option for automated verification of SCCharts. Both translations preserve the features of the original model as they originate from the same sequentialized SCG. Comparing model checking options of SPIN and nuXmv on these models can indicate sensible defaults for future use.

Each of the described models was tested using one true invariant and one true LTL property. Failing properties were not tested as this typically reduces the required resources for model checking. For instance, the explicit model checking approach implemented in SPIN will traverse more states when a

true property is checked. As soon as a violation of a property is found, it will stop and output the counterexample.

This thesis focuses on LTL, hence CTL and other temporal logic dialects of nuXmv were not considered. Further, not all options of the nuXmv command palette that is illustrated in Section 2.2.4 have been used in this evaluation for different reasons. First, compositional reasoning algorithms (e.g., `check_ltlspec_compositional`) require a proof file and thus have been considered as unfit for fully automated verification of SCCharts.

Second, some commands have been tried but discarded. They were not able to verify the models as expected, which could be related to the concrete translation from SCCharts to SMV that has been used to perform model checking with nuXmv.

- `msat_check_invar_bmc_implabs` finds an abstract counterexample in ABRO. The command requires predicates specified in the input file and thus has been considered unfit for fully automated verification of SCCharts. For comparison, `msat_check_invar_bmc_cegar_implabs` does automatically refine its property when finding a spurious counterexample, which is why this command has been evaluated.

- `check_invar_bmc_itp` performs interpolation-based BMC. There are several algorithms available for verification with this command. The algorithms `falsification` and `mcmillan` gave a wrong counterexample for the trivial property `o || !o` in ABRO. As a result, only the algorithms `mcmillan2`, `avy`, `itp_seq` and `itp_seq2` have been evaluated.

- `check_invar_bmc -a classic` did not prove all properties of the evaluated models. For instance, in the DVD-Player model it returns `cannot prove the invariant !(off & on) is true or false`. Similarly, the command `msat_check_invar_bmc -a classic` fails to prove the invariant `r -> !o` of ABRO.

- Several nuXmv commands for LTL model checking were not able to verify ABRO and counting with two 8-bit integers in parallel. As a result, they were not further evaluated on larger models. The corresponding commands are: `check_ltlspec_bmc`, `check_ltlspec_bmc_inc`, `check_ltlspec_inc_coi_bmc`, `msat_check_ltlspec_bmc`, `msat_check_ltlspec_inc_coi`, `msat_check_ltlspec_sbmc_inc`, `check_ltlspec_sbmc`, `check_ltlspec_sbmc_inc`.

6.2.3 Results

The run-time of evaluated algorithms are presented in Figure 6.8 for LTL properties and in Figure 6.9 for invariant properties. In these figures, the models are labeled using abbreviations. *tlcs-const* represents the TLCS where `waitTime` has been declared as constant. On the other hand, *tlcs-var* represents the model in which `waitTime` is declared as normal variable. The labels that end with *int-par* represent the models for counting 8-bit integers in parallel.

The plots do not show every tested combination of arguments. Instead, representatives that performed well have been chosen for better readability. The omitted results are explained in the following.

- `check_invar` has a parameter to select whether a forward search (i. e., from initial states to errors) or backward search (i. e., from errors to initial states) or a mixture of both should be done. The measured results are similar in most cases. However, the pure forward search required approximately 70 seconds in the Chrono model, whereas the other two options required 0.7 seconds. A mixture of both approaches, namely `forward-backward` has been chosen as representative in the diagram.

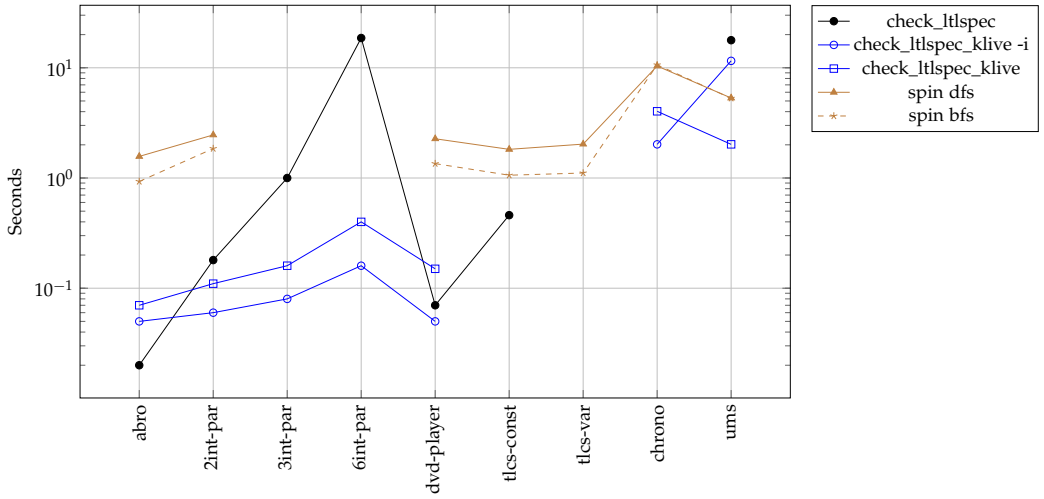


Figure 6.8. Run-time of model checking algorithms for LTL properties. Missing points represent a timeout. Black lines are BDD algorithms, blue lines are K-liveness algorithms, and brown lines are SPIN algorithms.

- `check_invar_cegar_predabs` performed similar to `msat_check_invar_bmc_cegar_implabs`. A noteworthy difference was measured in *tlcs-const* where `msat_check_invar_bmc_cegar_implabs` required approximately 48 seconds whereas `check_invar_cegar_predabs` required 0.9 seconds. Thus, `check_invar_cegar_predabs` is shown in Figure 6.9.

- There are multiple options to choose from when doing interpolation-based BMC (`check_invar_bmc_itp`). The `itp_seq` option timed out in the DVD-player, Chrono, and UMS models. The other evaluated options `avy`, `mcmillan2`, and `itp_seq2` performed similar in the test and did not time out. The `mcmillan2` option has been chosen as representative in the diagram.

- Several BMC commands for invariant properties have shown similar results. `check_invar_bmc -a een-sorensson`, `check_invar_bmc_inc`, `check_invar_inc_coi_bmc`, `msat_check_invar_bmc -a een-sorensson`, `msat_check_invar_inc_coi` all performed similar and failed in the same models. The interpolation-based BMC algorithms were able to handle more models without timing out, which is why the others are omitted in the graphic.

The evaluation shows that there are several commands of nuXmv that are superior to SPIN in many cases when performing model checking of SCCharts using the translations to SMV and PROMELA discussed in Chapter 4. Model checking of invariants using IC3, and `check_invar_bmc_itp` with `avy` or `mcmillan2` option performed best.

However, the TLCS with variable `waitTime` is an example that there are models that SPIN can handle more efficiently than nuXmv. There is a significant impact on model checking performance when `waitTime` is not declared as constant in the TLCS. In this version of the model, all algorithms of nuXmv timed out. This shows that integers should be declared as constant when possible.

For LTL model checking, K-liveness and BDD algorithms performed best in nuXmv and seem to complement each other. K-liveness timed out in the TLCS, whereas BDD algorithms timed out in the Chrono model. At least one of these approaches were faster and required less memory than SPIN in nearly all cases.

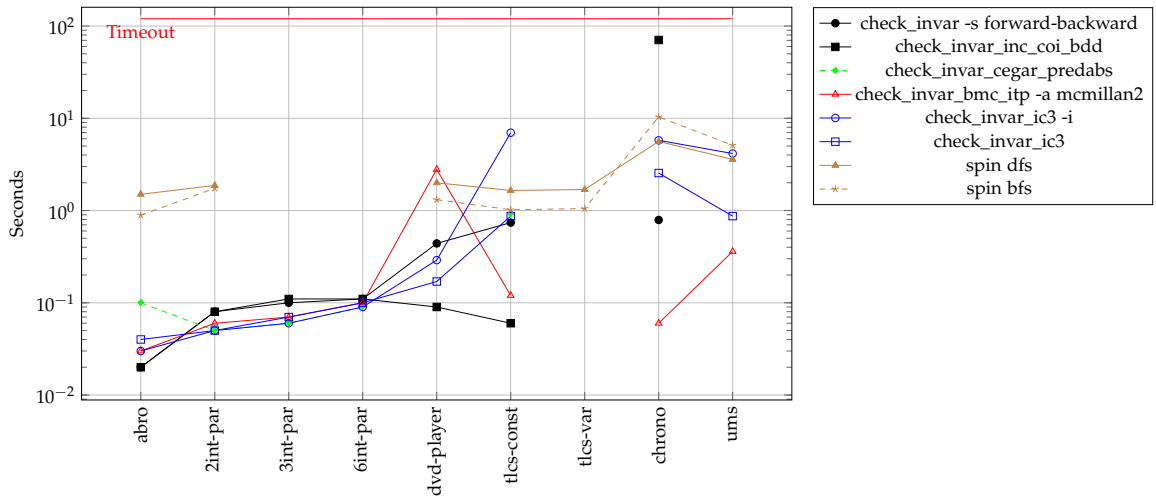


Figure 6.9. Run-time of model checking algorithms for invariant properties. Missing points represent a timeout. Black lines are BDD algorithms, the green line is an abstraction/refinement algorithm, the red line is for an interpolation-based BMC algorithm, blue lines are IC3 algorithms, and brown lines are algorithms of SPIN.

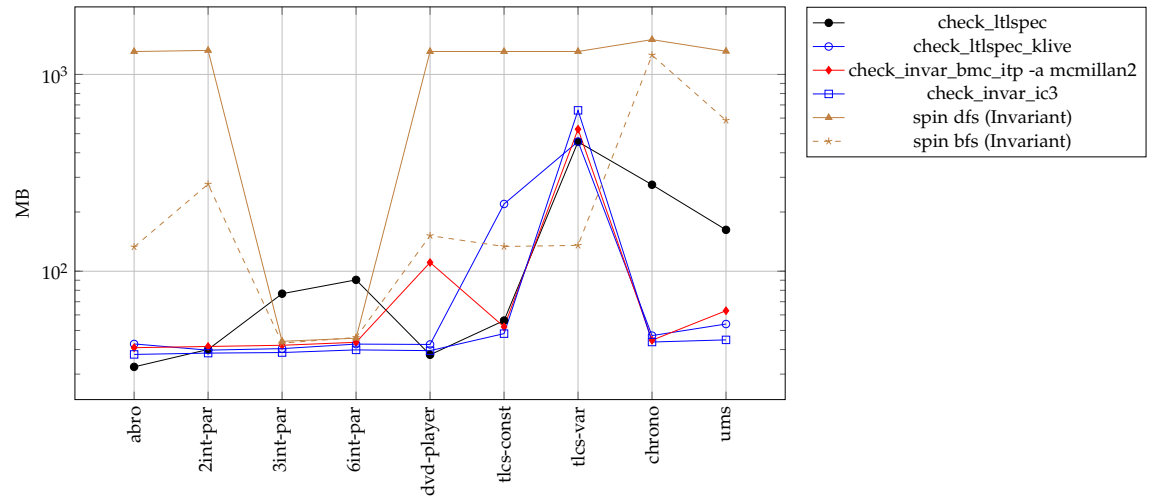


Figure 6.10. Max allocated RAM of model checking algorithms. The memory consumption of processes that timed out is also shown. SPIN timed out in *3int-par* and *6int-par*. All NUXMV algorithms timed out in *tlcs-var*. Further, K-liveness timed out in *tlcs-const* and `check_ltlspec` timed out in *chrono*.

Model checking performance of SPIN drops when the state-space increases. This has been observed with the models that are counting three and six 8-bit integers in parallel. NUXMV was able to handle these. SPIN was only able to model check the version that is counting with two 8-bit integers in parallel.

The memory consumption is also interesting when doing model checking. Figure 6.10 shows the memory consumption for the algorithms that performed well in the evaluation. For better readability, only the memory consumption of SPIN for the invariant properties is plotted. The values for LTL properties are similar. SPIN has been used with depth-first search and breadth-first search options. A

DFS is the default in SPIN. However, in the tested models, the BFS option often required less memory with similar run-time.

The evaluation showed that the symbolic approach of nuXmv often requires less memory than SPIN in a successful verification when using the presented translation to SMV and PROMELA. The algorithms that performed best by nuXmv required less than 100 MB of memory in most tests. They timed out in one version of the TLCS model and still required less than a GB of RAM. In contrast to this, SPIN required more than one GB in a successful verification run using a DFS. The BFS algorithm still required at least 100 MB in successful verifications. The low memory consumption of SPIN in *3int-par* and *6int-par* is because the process timed out.

6.2.4 Comparison with Lesar

The UMS was re-created from a Lustre model, which had been checked with Lesar by Halbwachs et al. [HLR92]. Thereby, Lesar required approximately one second for the verification using both, the symbolic (i. e., BDD) and explicit approach. This information is useful to put the results of the presented evaluation in a context. Some nuXmv commands verified the model in under a second (e. g., IC3). However, the BDD based model checking commands (e. g., `check_invar`), timed out in the UMS model. Furthermore, SPIN required approximately 3.5 seconds for its verification in a DFS. As a result, Lesar performed better on the Lustre UMS model than nuXmv and SPIN with similar algorithms on the tested SCCharts version. This is true even though the hardware used to verify the SCCharts version is orders of magnitude faster than the hardware that has been used in the paper by Halbwachs et al.

There are multiple possible explanations for this. First, the translation from the Lustre model to SCCharts has been done by hand and might not be optimal. Second, the presented translations to SMV and PROMELA might not be optimal.

The bad performance of SPIN compared to Lesar for the UMS model seems to have further reasons than the number of global variables. In contrast to nuXmv and Lesar, PROMELA has not been designed for a synchronous domain. Instead, it allows for modeling of concurrent processes with interleaving semantics that communicate via message passing and shared memory. As a result, SPIN seems to consider and store more states than necessary when verifying PROMELA models that have been created from SCCharts using the presented translation. For instance, SPIN names 52503 stored states in the UMS example, whereas the Lesar paper names only 54. A deviation from the original model in this order of magnitude is unlikely. Further, the run-time of the explicit model checking approach should not be affected as much by superfluous pre-variables because most of their configurations are unreachable. In conclusion, SPIN seems to consider unneeded states using the presented translation, which makes it not optimal for verification of SCCharts.

Anyway, the IC3 algorithms and BMC with interpolants managed to verify the UMS and most other models in under one second and with less than 100 MB of RAM. Although the examples are simple, it demonstrates the effectiveness of these algorithms for model checking SCCharts and its potential for practical use.

Chapter 7

Conclusion

The following summarizes the contributions of this thesis. This includes the design and implementation of model checking features for SCCharts and their evaluation. Afterwards, Section 7.2 names possible applications for the implementation in its current state. Finally, Section 7.3 discusses possible future work to extend the presented implementation and to evaluate model checking features for SCCharts in a practical setting.

7.1 Summary

The goal of this thesis is to enable automated model checking of SCCharts that integrates in the KIELER tool. Therefore, the model checkers SPIN and nuXmv have been used as verification back-end. These model checkers performed well in case studies, meet the requirements for reactive systems modeling and enable to evaluate a variety of different approaches for verification of SCCharts.

Translations to the model checker input languages SMV and PROMELA have been presented. Thereby, the sequentialized SCG has been re-used, which is a low-level representation of SCCharts within the KIELER compiler. This data-structure was translated to PROMELA by mapping its language constructs to equivalent code patterns in PROMELA.

SMV is a data-flow language in contrast to the SCG, which is control-flow based. Two approaches have been discussed to translate the sequentialized SCG to SMV. The first is using a program counter to encode which statement of the SCG is executed. This approach results in multiple reactions of the SMV model to simulate one reaction of the SCG. The second translation approach is using the semantic proximity of the sequentialized SCG to a netlist that expresses the reaction in a single synchronous reaction using data-flow. Therefore, the SCG is first translated to an SSA from. This thesis focused and implemented the SSA approach for the translation to SMV because it conserves temporal properties of the original model and requires fewer global variables.

The work-flow for model checking SCCharts that has been implemented in this thesis is the following: The model is augmented with properties to be checked. Further, invariants that are assumed to hold in the model can be specified. Adding this information has been implemented using annotations, such that all model-checking relevant information is held within the model. Next, the model checking task is started, which will translate the model to SMV or PROMELA and start the corresponding model checker.

Interfacing with the model checking process has been implemented using stdin and stdout. The concrete commands that are sent to nuXmv and SPIN can be configured by the user from within KIELER. The output of the model checker is parsed to present the results. In case a counterexample is found, it is translated to the KTrace format from the perspective of the original model. Thereby, only values of inputs and outputs are kept as these describe the behavior of the high-level model. A KTrace counterexample can be re-played on the original SCChart in KIELER.

The resulting SMV and PROMELA code from the translation has been evaluated using small SCCharts and different configurations of the model checkers. For invariants, the IC3 algorithm and interpolation-

based BMC implemented in nuXmv have been found to perform best in most cases. However, there are models that SPIN did handle more efficiently. For LTL properties, K-liveness and BDD-based model checking performed well. These algorithms timed out in different models, such that for a practical usage, a combination of both can be beneficial. Model checking finished within fractions of a second in the best cases using nuXmv and within a few seconds in case SPIN performed best. This shows the potential of the new model checking features for practical use.

In the execution of this thesis, the integration of nuXmv into KIELER has been perceived as easier and more stable than SPIN because nuXmv can detect division-by-zero errors and clearly tells whether a property passed, failed or could not be verified. Further, the ranges of integers are checked by nuXmv in certain cases, e. g., during the BDD construction. Some characteristics of models have been found that affect the performance of model checking SCCharts when using the proposed translation to SMV.

- Comparing integer variables with constants performed better than comparing them with other variables. Certain algorithms of nuXmv seem to be able to deduce the bounds of variables from comparisons with constants.

- The state-space introduced by integer variables is not trivial to explore for most tested model checking algorithms. This is true even when they are assigned only a single value in SCCharts. As a result, variables should be declared constant when possible in SCCharts. This will replace them with their associated value such that unnecessary variables are avoided in the SMV and PROMELA code.

7.2 Possible Applications

The new model checking features allow SCCharts developers to verify properties of created models using nuXmv and SPIN. A motivation and application area for this is the verification of safety-critical applications in an industrial context. However, there are further possible scenarios to utilize the implemented features in their current form, which are named in the following.

7.2.1 Teaching Temporal Logics

In KIELER, SCCharts are written in a textual syntax and the graphical model is synthesized automatically to combine the benefits of both approaches. Certain characteristics of the model, for example loops, are visually laid out, which can increase the understanding.

Thus, the new model checking features enables to teach students temporal logics using a combination of graphical and textual models. Using KIELER, students can learn the basics of different temporal logics and model checking in a hands-on approach. The work-flow for model checking requires little interaction. Performing model checking and getting the result can be achieved with two buttons. Similarly, re-playing a counterexample requires few button clicks. This allows for a fast code-test work-flow and requires little effort to experiment with properties and models. The translation of CTL properties to SMV has not been the focus of this thesis but has been implemented nonetheless. Thus, LTL and CTL can be used for properties of a model to explore the semantics of a linear time and branching time logic.

There are plans to use the new model checking features for KIELER SCCharts to teach temporal logics as part of an embedded systems lecture at Kiel University.

7.2.2 Testing of SCCharts Compiler

Model checking can be useful not only for end-users who want to verify their models. Developers can use the new model checking features to test the KIELER compiler implementation.

In fact, the new model checking features have already been useful in finding a bug in the KIELER compiler. A strong-abort was not behaving correctly in compile chains that added information about the taken transitions of a reaction. Model checking the low-level SCG representation resulted in a counterexample that was used to find and fix the bug. This example reconfirms the choice to do model checking on a low-level representation, which has been discussed in Section 4.2.

The KIELER testing infrastructure can be used to write test-cases that perform model checking on SCCharts and compare the expected result with the actual result from new compiler implementations. This can complement simulation tests, in which expected input-output traces for a model are compared against its actual behavior. Using model checking can increase test-coverage because it is not limited to existing traces. However, such tests are not comparable to a formally verified compiler. It does not prove that the translation is correct for all models, only for the test-cases and specified properties.

7.3 Future Work

The following names possible future work to improve the current implementation of model checking SCCharts and other sequentially constructive languages. Further, a case study is suggested to evaluate the tools in a practical setting.

7.3.1 Extending the Translation to SMV and PROMELA

This thesis was executed within a fixed time frame, which is why a subset of SCCharts has been selected for the implementation of the translation to SMV and PROMELA. Moreover, the focus in this thesis was on LTL properties and invariants, although nuXmv supports further temporal logics. Thus the implementation can be extended as described in the following.

- More data types can be translated to SMV and PROMELA, e. g., floats, arrays, and strings.
- There are some non-standard assignment operators in SCCharts, e. g., min=, and max=. These syntactical elements are used to aid the compiler in the analysis of sequential constructiveness and scheduling of statements. Semantically, however, these operators can be translated to PROMELA and SMV using conditionals.
- nuXmv supports the standardized Property Specification Language (PSL) [IEEE1850][EF07]. Thus, the model checking implementation for SCCharts can be extended to support such specifications as well.
- Adding information relevant for model checking has been implemented using the general annotation mechanism of SCCharts. Another possibility is to extend the textual SCCharts grammar with model checking constructs such as `assert` and `assume`. This would allow the parser to detect syntax errors in properties. Moreover, integrating grammars for specification languages (e. g., LTL) would enable to translate and manipulate properties on the level of an AST instead of doing string manipulations.
- The model-to-model transformations used to compile SCCharts in KIELER can be extended to add information relevant for model checking. For instance, range assumptions could be provided by many transformations that introducing new integers.

- ▻ This thesis focused on the netlist-based compilation of SCCharts due to the equivalence to hardware circuits and the structural simplicity of the sequentialized SCG. However, the priority-based compilation approach can handle a greater set of programs. Implementing priority-based scheduling of an SCG in general requires multiple iterations over the tick logic and jumps between concurrently executed statements. In PROMELA this could be implemented using the goto-statement and loops. In SMV it could be implemented using the program counter approach explained in Section 4.3.2. Afterwards, the results could be compared to the netlist-based approach for model checking SCCharts.

7.3.2 Further Model Checkers

The model checkers nuXmv and SPIN have been used as verification back-end in this thesis. However, there are plenty of model checkers available for various domains. SPIN seems to consider more states than necessary when verifying PROMELA models created from SCCharts as discussed in Section 6.2.4. Thus, another model checker that implements an explicit model checking approach could be beneficial for model checking SCCharts.

Besides the option of re-using existing model checkers, it is also possible to implement model checking algorithms dedicated to SCCharts.

7.3.3 Further Synchronous Languages

This thesis focused on the SCCharts language. However, the implemented solution uses the sequentialized SCG for the translation to PROMELA and SMV. As a result, the model checking features can be re-used for other synchronous languages that can be compiled to this representation. For instance, KIELER implements a sequentially constructive version of Esterel (SCEst) [SMR+17] that could be model checked with the presented translation. Therefore, properties and assumptions of the model have to be specified, which could be implemented similar to the annotation mechanism in SCCharts.

7.3.4 Visualization of State-Space

nuXmv can output the explicit finite state machine of an SMV model in XMI format. This could be visualized in KIELER to enable graphical exploration of the state-space with automatic layout and filtering options. In comparison to this, the Xeve model checker for Esterel can generate minimized finite state machines in the textual FC2 format, which can be loaded in the tool ATG for graphical exploration [Bou97a].

7.3.5 Evaluation in Case Study

An evaluation of the implementation has been executed on small SCCharts models and efficient model checking algorithms have been identified. Further, certain model characteristics that affect performance have been found. This enables the evaluation of the new model checking features on real-world tasks in future work. One possibility is an evaluation as part of lecture exercises, which has been done before for SCCharts.

Another option is to implement and verify the solution for an industrial size problem in SCCharts. As discussed in Chapter 3, the Steam-boiler Control Specification Problem has been used to demonstrate capabilities of various modeling languages and verification tools [ABL96]. This problem in particular is suited to evaluate the new model checking features in a case-study because there is data for comparison with other tools from the synchronous domain. Models of the Steam-boiler Specification have been written in various languages including Lustre [CD96] and Esterel [Bou97b].

Bibliography

[IEEE1850] "IEEE Standard for Property Specification Language (PSL)". In: *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)* (Apr. 2010), pp. 1–182. DOI: 10.1109/IEEESTD.2010.5446004.

[ABL96] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack. "The steam boiler case study: competition of formal program specification and development methods". In: *Formal Methods for Industrial Applications*. Springer, 1996, pp. 1–12.

[And95] Charles André. *Syncharts: a visual representation of reactive behaviors*. Tech. rep. 1995.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. "The Synchronous Languages Twelve Years Later". In: *Proc. IEEE, Special Issue on Embedded Systems*. Vol. 91. Piscataway, NJ, USA: IEEE, 2003, pp. 64–83.

[Ber89] Gérard Berry. "Real time programming: special purpose or general purpose languages". PhD thesis. INRIA, 1989.

[BK08] C Baier and Joost P. Katoen. *Principles of model checking*. MIT Press, May 2008. ISBN: 978-0-262-02649-9.

[Bou97a] Amar Bouali. *XEVE : an ESTEREL Verification Environment : (Version v1_3)*. Technical Report RT-0214. INRIA, 1997, p. 23. URL: https://hal.inria.fr/inria-00069957.

[Bou97b] Michel Bourdellès. *The Steam Boiler Controller Problem in ESTEREL and its Verification by Means of Symbolic Analysis*. Tech. rep. RR-3285. INRIA, 1997. URL: https://hal.inria.fr/inria-00073403.

[BW96] Robert Büssow and Matthias Weber. "A steam-boiler control specification with statecharts and z". In: *Formal Methods for Industrial Applications*. Springer, 1996, pp. 109–128.

[BWL06] Frederic Boniol, Virginie Wiels, and Emmanuel Ledinot. "Experiences in using model checking to verify real time properties of a landing gear control system". In: *ERTS 2006: 3rd European Congress Embedded Real Time Software*. 2006, pp. 25–27.

[CCD+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. "The nuxmv symbolic model checker". In: *International Conference on Computer Aided Verification*. Springer. 2014, pp. 334–342.

[CD96] Thierry Cattel and Gregory Duval. "The steam boiler problem in lustre". In: *Formal Methods for Industrial Applications*. Springer, 1996, pp. 149–164.

[Cho07] Yunja Choi. "From nusmv to spin: experiences with model checking flight guidance systems". In: *Formal Methods in System Design* 30.3 (2007), pp. 199–216.

[CKS+05] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. "Satabs: sat-based predicate abstraction for ansi-c". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2005, pp. 570–574.

[EF07] Cindy Eisner and Dana Fisman. *A practical introduction to psl*. Springer Science & Business Media, 2007.

[FFC+10] Marc Frappier, Benoît Fraikin, Romain Chossart, Raphaël Chane-Yack-Fa, and Mohammed Ouenzar. "Comparison of model checking tools for information systems". In: *International Conference on Formal Engineering Methods*. Springer. 2010, pp. 581–596.

[FH10] Hauke Fuhrmann and Reinhard von Hanxleden. *Taming graphical modeling*. Technical Report 1003. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2010.

[Fuh11] Hauke Fuhrmann. "On the pragmatics of graphical modeling". Dissertation. Kiel: Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, 2011.

[Hal93] Nicolas Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Publishers, 1993.

[HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. "SCCharts: Sequentially Constructive Statecharts for safety-critical applications". In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Edinburgh, UK: ACM, June 2014, pp. 372–383.

[HLM+12] Reinhard von Hanxleden, Edward A. Lee, Christian Motika, and Hauke Fuhrmann. "Multi-view modeling and pragmatics in 2020 — position paper on designing complex cyber-physical systems". In: *Pre-Proceedings of the 17th International Monterey Workshop on Development, Operation and Management of Large-Scale Complex IT Systems*. Oxford, UK, 19–21 3 2012, pp. 209–223.

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. "Programming and verifying real-time systems by means of the synchronous data-flow language lustre". In: *IEEE Trans. Softw. Eng.* 18.9 (1992), pp. 785–793. ISSN: 0098-5589.

[HMA+14] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop. "Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation". In: *ACM Transactions on Embedded Computing Systems, Special Issue on Applications of Concurrency to System Design* 13.4s (July 2014), 144:1–144:26.

[Hol97] Gerard J. Holzmann. "The model checker spin". In: *IEEE Transactions on software engineering* 23.5 (1997), pp. 279–295.

[MF18] Franco Mazzanti and Alessio Ferrari. "Ten diverse formal models for a cbtc automatic train supervision system". In: *arXiv preprint arXiv:1803.10324* (2018).

[MVR14] Zaur Molotnikov, Markus Völter, and Daniel Ratiu. "Automated domain-specific c verification with mbeddr". In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM. 2014, pp. 539–550.

[Pei17] Lars Peiler. "Priority-based compilation of SCCharts". `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lpe-mt.pdf`. Master thesis. Kiel University, Department of Computer Science, 2017.

[PG09] Julio C Peralta and Thierry Gautier. "Towards smv model checking of signal (multi-clocked) specifications". In: *Electronic Communications of the EASST* 23 (2009).

[Ray08] Pascal Raymond. "Synchronous program verification with lustre/lesar". In: *Modeling and Verification of Real-Time Systems*. ISTE/Wiley, 2008. Chap. 6.

[Rus14] John Rushby. “The versatile synchronous observer”. In: *Specification, Algebra, and Software: Essays Dedicated to Kokichi Futatsugi*. Ed. by Shusaku Iida, José Meseguer, and Kazuhiro Ogata. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 110–128. ISBN: 978-3-642-54624-2. DOI: 10.1007/978-3-642-54624-2_6. URL: https://doi.org/10.1007/978-3-642-54624-2_6.

[Sch16] Alexander Schulz-Rosengarten. “Strict sequential constructiveness”. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/als-mt.pdf. Master thesis. Kiel University, Department of Computer Science, 2016.

[SMH18] Steven Smyth, Christian Motika, and Reinhard von Hanxleden. “Synthesizing manually verifiable code for statecharts”. In: *Proc. Reactive and Event-based Languages & Systems (REBLS '18), Workshop at the ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*. Boston, MA, USA, Nov. 2018.

[SMR+17] Steven Smyth, Christian Motika, Karsten Rathlev, Reinhard von Hanxleden, and Michael Mendler. “SCEst: Sequentially Constructive Esterel”. In: *ACM Transactions on Embedded Computing Systems (TECS)—Special Issue on MEMOCODE 2015* 17.2 (Dec. 2017), 33:1–33:26. ISSN: 1539-9087.

[SSH18] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. “Towards interactive compilation models”. In: *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018)*. Vol. 11244. LNCS. Limassol, Cyprus: Springer, Nov. 2018, pp. 246–260.

[VRK+13] Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schaetz. “Mbeddr: instantiating a language workbench in the embedded software domain”. In: *Automated Software Engineering* 20.3 (2013), pp. 339–390.

[YDT14] Bin Yu, Zhenhua Duan, and Cong Tian. “Bounded model checking of traffic light control system”. In: *Electronic Notes in Theoretical Computer Science* 309 (2014), pp. 63–74.

List of Acronyms

AST	Abstract Syntax Tree
BDD	Binary Decision Diagram
BFS	Breadth-first search
BLIF	Berkeley Logic Interchange Format
BMC	Bounded Model Checking
CBMC	C Bounded Model Checker
CTL	Computation Tree Logic
DFS	Depth-first search
DSL	Domain Specific Language
EMF	Eclipse Modeling Framework
IDE	Integrated Development Environment
IUR	Initialize-Update-Read
KiCo	KIELER Compiler
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
LTL	Linear Temporal Logic
MoC	Model of Computation
MPS	Meta Programming System
PROMELA	Process Meta Language
PSL	Property Specification Language
SAT	Satisfiability
SC MoC	Sequentially Constructive Model of Computation
SCADE	Safety-Critical Application Development Environment
SCChart	Sequentially Constructive StateChart
SCG	Sequentially Constructive Graph
SCL	Sequentially Constructive Language
SMT	Satisfiability Modulo Theories

7. List of Acronyms

SPIN Simple PROMELA Interpreter

SSA Static Single Assignment

TLCS Traffic Light Control System

UI User Interface

UMS U-turn Management System

GUI Graphical UI

List of Listings

2.1 A small KTrace example . . . 9
4.1 Textual SCChart with a failing property . . . 23
4.2 Textual SCChart with a failing property and the resulting logic in SCL . . . 23
4.3 SMV counterexample when using VAR for inputs . . . 23
4.4 SMV counterexample when using IVAR for inputs . . . 23
4.5 A program with control-flow. . . . 24
4.6 The control-flow program in data-flow using a program counter (pc). . . . 24
4.7 SMV code for the SSA SCG . . . 25
4.8 PROMELA code for the sequentialized SCG . . . 28
5.1 Textual SCChart with model checking annotations . . . 35
5.2 Main method for the translation of the SCG tick logic to PROMELA. . . . 36
5.3 Adapting LTL formulas to SPIN syntax with an initial setup reaction. . . . 36
5.4 Method that adapts expressions to SMV syntax . . . 38
5.5 KTrace counterexample for Listing 4.3 . . . 40
6.1 Textual SCChart that performs a division-by-zero. . . . 43
6.2 PROMELA model exploring the full range of an int. . . . 43
6.3 The UMS model implemented in SCCharts using a data-flow region. . . . 51

List of Figures

1.1 Syntax overview of SCCharts . . . 2
1.2 Schematic of model checking . . . 3

2.1 ABRO example in SCChart . . . 6
2.2 Timeline of micro-ticks and macro-ticks. . . . 7
2.3 Transformation matrix of SCCharts, SCG, and data-flow . . . 8
2.4 Illustration of the intuitive semantics of LTL . . . 11
2.5 Overview of nuXmv commands . . . 13

4.1 SCG in SSA form for the ao-int model . . . 25
4.2 The ao-int model and its sequentialized SCG. . . . 28
4.3 A simple traffic light modeled in SCCharts . . . 32
4.4 Synchronous observer pattern applied to the traffic light model . . . 32

5.1 Overview of relevant plug-ins . . . 33
5.2 Common data structures that hold information for model checking tasks. . . . 34
5.3 Compile chain to PROMELA. . . . 35
5.4 Compile chain to SMV. . . . 36
5.5 Schematic of a counterexample from nuXmv . . . 40
5.6 Screenshot of the model checking view in KIELER . . . 42

6.1 SCChart that sequentially counts with three integers (version 1) . . . 45
6.2 SCChart that sequentially counts with three integers (version 2) . . . 45
6.3 An SCChart that counts concurrently from 0 to 255 with 6 integers. . . . 47
6.4 DVD-Player example in SCCharts . . . 47
6.5 Example for a chronometer that counts seconds and minutes . . . 48
6.6 SCChart for a Traffic Light Control System . . . 50
6.7 Schematic of a U-turn area for trains . . . 51
6.8 Run-time of model checking algorithms for LTL properties . . . 53
6.9 Run-time of model checking algorithms for invariant properties . . . 54
6.10 Max allocated RAM of model checking algorithms . . . 54

List of Tables

4.1 Comparison of NUXMV and SPIN 19
4.2 Translation patterns from SCL to PROMELA. 27
4.3 Translation patterns for reactive systems in PROMELA. 30
4.4 Translation pattern for assumptions in PROMELA. 31

6.1 Statistics about the tested models 46