Institut für
Theoretische Informatik


VerSyKo

Verifikation von Systemen synchroner Software-Komponenten

Verification of Systems With Synchronous Software Components

Contract Specification and
Domain Specific Modeling Language for GALS Systems
An Approach to System Validation

Version of 24. Juni 2013

Verified Systems International GmbH

Informatik Consulting Systems AG

Institut für Theoretische Information, Technische Universität Braunschweig

Das diesem Bericht zugrunde liegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung und Forschung unter dem Förderkennzeichen 01IS10050A, 01IS10050B, 01IS10050C gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt bei den Autoren.

Contents

Chapter 1

Introduction

1.1 Problem description and main goals

Computer systems are pervasive in everyday life and often carry responsibility for human lifes, e. g. in airplane autopilots or train control systems for railways. Such safety-critical systems must thus adhere to very high quality requirements concerning safety, reliability and availability. Hence, a very high effort has to be put into verification, validation and certification. The necessary measures to ensure the required high quality of the software of safety critical system are regulated through standards (e.g. CENELEC EN 50128 for the railway domain, DO178B for the avionics domain, as well as IEC 26262 for the automotive domain) that must be adhered to during the development process. The use of **formal methods** in the software development process is highly recommended by the standards. They allow to model and verify important and safety relevant logical functionality of a software at an early point during the development process and independently from hardware platforms used later on for deployment of systems. It is important to notice here that state-of-the-art systems are almost always composed from other distributed (component) systems (**system-of-systems (SoS)**). This is, however, still a challenge for formal methods when scalability to real industrial applications is regarded.

The goal of VerSyKo is the development of a general and universal approach to modeling and verification of software of distributed safety critical embedded systems; an approach that is innovative for the industrial practice and adresses the scalibility problem. The project focuses on the model based development and analysis of asynchronously communicating embedded control systems that are composed from components that operate synchronously. This system paradigm is known as GALS (**globally asynchronous-locally synchronous**) architecture. For complex safety relevant control tasks this is the most preferred solution: in the nodes of the distributed system one has controllers performing specialized tasks in hard real time by operating cyclically and in a synchronous way. For such a controller the model based development approach of SCADE is an attractive solution, which provides code generation, (formal) verification and test automation. Today, the relevance of the synchronous paradigm has been widely acknowledged in the scientific communities. Moreover, attractive alternatives to SCADE – such as synchronous interpretations of UML2 or Matlab/Simulink models – are also available, reducing the risk for industry to invest into this paradigm, which represents a new technology according to the state of the art in many companies.

However, for a whole distributed system, a synchronous implementation is neither realistic from a technical point of view nor desirable from the point of view of applications. Instead, locally synchronous nodes are integrated through a surrounding layer of software (often called *glue code*) that provides the necessary infrastructure for asynchronous communication. As a result, the SCADE (or

any other purely synchronous) formalism cannot be applied to a system of systems. Within VerSyKo we will close this methodological gap between synchronous component systems and asynchronous system of systems through domain specific modeling formalisms. The main emphasis is upon modeling and specification (with and without explicit consideration of real time and stochastic aspects pertaining random component failures), verification, validation and test. Domain specific formalisms are developed as UML 2.x profiles. They incorporate graphical representations and user friendly syntax for specification of GALS systems tailored towards system engineers with background in the respective application domain.

The main idea to address the complexity issues of GALS system is to provide for each synchronous component an abstract model in the form of a **contract** that can be locally verified for the component (e. g., for components modeled in the SCADE formalism by the formal verification engine provided by SCADE). The network of all component contracts then forms an **abstract GALS model** against which a requirement can be formally verified.

Another goal of the project is to provide tools for effective system testing: in an SoS scenario, local components of the GALS system network will have been thoroughly tested and even formally verified before their integration into the SoS network. These local verification results are denoted by **guaranteed behavior** of the associated local synchronous components. In the current approach to SoS testing applied in practice today it has to be criticised that the resulting tests frequently just "redo" variants of the HW/SW integration tests already performed by component suppliers, so that the system-level tests fail to increase the confidence into overall system correctness and effectiveness. It is therefore an explicit goal of this project to elaborate a systematic strategy for exploiting guaranteed behavior in the design of system verification goals and test strategies: documented proven behavior helps to avoid duplication of HW/SW integration tests on system level, and at the same time serves as counter examples or as means to strengthen contracts during the abstract GALS model verification process.

To evaluate all methods, languages and tools developed by VerSyKo we will use two case studies provided by the industrial partners; one case study from the railway domain and one from the avionics domain.

1.2 Overview of main concepts/tools developed by VerSyKo

The goal is VerSyKo is to develop a framework for specification and verification of GALS systems. This specification framework is described in Sections 4 and 5. It allows to specify the synchronous components of a system and how they are composed to form the whole system. For each synchronous component an interface with its input and output data is specified. Components are composed by connecting their outputs with inputs of other components. To each component its implementation in a synchronous language (e. g. SCADE) is associated. In order to make verification possible the behavior of each component is described by one or more *contracts*. Each contract is either specified as a formula in linear temporal logic or by a synchronous state machine. Contracts are thought as an abstraction of the complete behavior of a concrete component, the latter of which is given by its associated SCADE model. So a GALS systems is specified as a "network of contracts" in our framework called *abstract GALS model*.

There is a special form of contract called a *guaranteed behavior*. Such a contract specifies behavior of a component that the corresponding SCADE model is known to fulfil. This could be behavior verified by testing or formal verification by the supplier of a component before integration into the GALS system.

Our specification formalism supports multiple instances of components. This is necessary because GALS systems often contain many instances of certain—possibly very simple—components (see for example our avionics case study in Section 12.1).

In addition to the components (their instances) and the connections between them one also specifies *verification goals*. They express system requirements (in the form of safety properties) that the integrated GALS system has to satisfy. While for a single synchronous component linear temporal logic and synchronous automata are sufficient, on the global level one needs to be able to formulate properties that refer to real time, and so on the global level we have a dense time semantics. Whenever verification goals do not refer to real time formal verification can be performed with an analysis tool based on a interleaving semantics (PROMELA/SPIN). Otherwise a real time formalism liked UPPAAL timed automata is necessary.

The overall architecture of the framework consists of three layers as shown in Fig. 1.1.

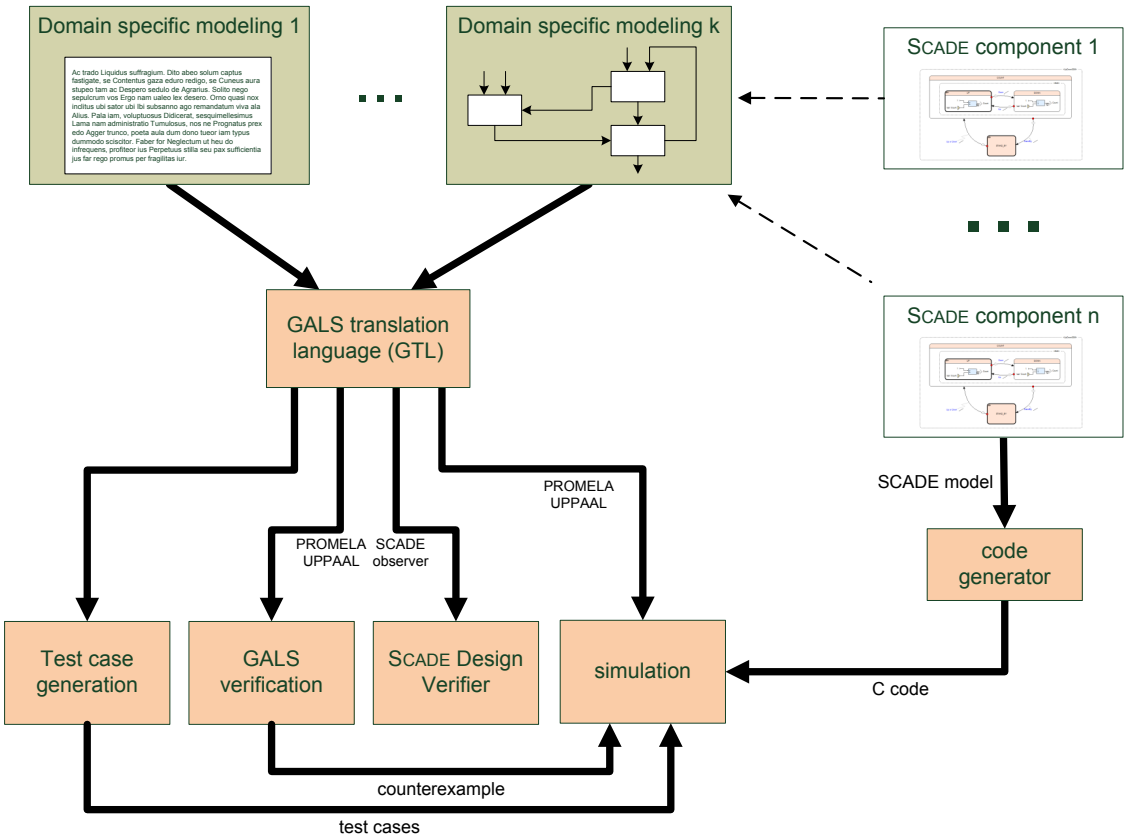

Figure 1.1: Framework for verification, validation and test of GALS systems

The top layer is a domain specific modeling language to specify GALS systems. This layer uses a mixture of graphics and text for specification. It is based on a UML profile and prototypical tool support is developed in Enterprise Architect[1]. This is described in detail in Section 4.

On the middle layer we have a textual form of the specification of a GALS system. For this VerSyKo develops a specification language for GALS systems called *GALS Translation Language* (GTL). Its purpose is to have a stable language core so that (1) on top of it one can build extensions

[1]Enterprise Architect is an UML case tool developed by Sparx Systems `http://www.sparxsystems.com`.

and user friendly graphical respresentations like the one we develop on the first level and (2) various model transformations to test automation and formal verification formalisms and tools can be defined. We describe the syntax and semantics of GTL in Section 5.

From the GTL level there are four kinds of model transformations considered in VerSyKo. We briefly describe them here; the corresponding details and algorithms are described in Section 7.

The purpose of the first transformation is local verification of components, i. e., the formal verification that components (given by their SCADE models) satisfy their contracts. So this transformation translates the contracts of each component into a synchronous observer for the component, and uses the SCADE Design Verifier (DV) (the formal verification component of SCADE) to prove that components satisfy their contracts. This transformation ignores any guaranteed behaviors of a component as those are already known to hold for the components. SCADE DV has limitations because its proof engine is based on a SAT solver and because it uses the SCADE syntax to specify properties for formal verification. So there is only a fragment of the GTL language that can be translated. Within VerSyKo we also plan to use *bounded model checking* for this kind of local verification. Here a model transformation that takes a SCADE model of a component and the contracts of the component and produces input for a bounded model checker will be implemented.

The second kind of transformation is for the verification of GALS systems. So the purpose is to verify the verification goals for a GALS system. This transformation translates the abstract GALS model ("network of contracts") to an appropriate analysis tool to perform this formal verification. Within VerSyKo we will develop two such model transformations. One uses PROMELA/SPIN as backend for formal verification, and this allows to verify verification goals that do not refer to real time. The second one uses UPPAAL timed automata as backend and this allows the verification of real time properties. Once these transformations are implemented in prototypical form we will also investigate how well the two different verification backends perform.

It is interesting to note that the guaranteed behaviors also play a special rôle for this GALS verification. In a first verification attempt they may not be translated to the verification backend. Instead they can be used for automatic abstraction refinement in the case of "false negatives" as explained next in connection with the third model transformation.

The third model transformation makes use of the C integration in SPIN. Its main purpose is to eleminate "false negatives" within the verification results. False negatives are *spurious* counterexamples produced by a verification backend to show that a verification goal does not hold for the abstract GALS model while it is actually true for the corresponding concrete GALS model ("network of implementations"). For the SPIN verification backend we can make use of the C integration in PROMELA to recognize false negatives. This works as follows: suppose we are given an error trace from the formal verification of a verification goal. Then our third model transformation integrates the C code generated from all SCADE models for the components of a GALS system into one PROMELA model. Now one simulates this model with the given error trace to see whether the simulation indeed exhibits erroneous behavior. If not then one has found a false negative, which is an indication that the contracts of the components were not strong enough to force the verification goal. In this case one can use guaranteed behaviors to strenthen the contracts. This means that the formal verfication of the whole GALS system is rerun but this time taking into account one or several guaranteed behaviors as additional contracts. In addition one might also use the error trace produced by the previous verification for strengthening contracts, as it was verified by model simulation that this trace does in fact not occur with the SCADE implementations.

Finally, the fourth kind of transformation in Fig. 1.1 above is for test case generation. Here the abstract GALS model is understood as a test model for the integration test of the components. So a test case generator extracts test cases from this model which can then drive a test environment (e. g. the

above simulation model in PROMELA or RT-Tester[2]) in which the concrete GALS model is tested.

1.3 Related Work

Numerous works are devoted to combining synchrony with asynchrony, or to extend synchronous modeling towards less synchronous applications (see e. g. [18] for a summary on work on synchronous languages).

The concept of a GALS system was first investigated by Chapiro [30]. The GALS paradigm has mainly been studied in connection with hardware systems (see e. g. [13, 19]). More recently, GALS is also investigated in connection with software systems. For example, the formal verification of GALS systems by a combination of synchronuous and asynchronouous formalisms appears in several different works:

(1) Thivolle and Garavel [41] explain the basic idea and show the advantages of the GALS approach by applying it to the verification of a communication protocol. The paper explains how synchronous components can be understood as functions and how these components can be integrated into an asynchronous verification tool . In contrast to our work in VerSyKo this work does not develop a language for the specification of GALS systems, and components are not abstracted by contracts as in our work. The paper only treats one example where synchronous components are integrated into a GALS system.

(2) Doucet et al. [38] describes a translation from the synchronous language SIGNAL to PROMELA. The verification of translated GALS models is then performed using SPIN. The idea to use contracts as abstractions of synchronous components and formally verify those contracts within the synchronous formalism is new in our work in VerSyKo.

(3) "Communicating Reactive State Machines" [86] are another formalism for modelling and verification of GALS systems. The code is translated to PROMELA and verified using SPIN. The described system is comprised of a graphical editor, a simulator and a verification engine. Apart from the well researched method of slicing, no further optimization techniques are being discussed.

(4) The "IF toolset" [25] introduces a formalism for the specification of asynchronous systems. This formalism is based on components and their composition by connections. That work does not focus on GALS systems but rather on a formalism that can integrate many different design formalisms.

We now mention three approaches to deal with asynchrony which are less related to the work in VerSyKo.

The first approach to dealing with asynchrony is to extend synchronous formalisms. For example, "Multiclock Esterel" [85] extend the Esterel synchronous language by a possibility to supply several clocks for different components. However, this formalism remains fully synchronous and every model can be translated into an equivalent Esterel model.

Secondly, a different line of research deals with compiling synchronous programs to produce distributed, not strictly synchronous code which is correct-by-construction. One example of this is [83].

[2]RT-Tester is the real time test tool developed by Verified Systems

Thirdly, an approach we do not follow in VerSyKo is to use synchronous formalisms to model asynchrony. This idea goes back to R. Milner [73, 74]. The modeling tool Model-Build [15, 16] and the Polychrony workbench [68] are based on this idea and also the work in [54, 63, 76].

It is well-known that formal verification of synchronous (component) programs, in particular SCADE models, can be performed by using synchronous observers to specify properties (see [53, 55]). An application of this approach in an industrial context is for example [36]. Further work on formal verification of SCADE models concerns formal safety analysis (see [2] and [50]). We are not aware of work that uses abstraction of components in connection with formal verification in SCADE.

The concept of systems of systems (i. e., systems being composed of components) also appears in numerous works. For example, one formal model of a system component are interface automata [6]. While this formalism deals well with composition and refinement the components have asynchronous behavior in contrast to what we study in VerSyKo.

Using contracts as specifications for parts of a program or system is also not a new idea; see for example work on rely/guarantee logic [64]. The idea to abstract systems components by contracts appears recently for example in [45, 44]. The specification language used there is based on UML and the work does not deal with synchronous components and GALS systems.

So the various ingredients (contracts for abstraction, synchronous verification, GALS systems) of our work are well-established in the literature. However, to the best of our knowledge these ingredients have not been brought together in one single practical framework. It is this gap that we intend to fill with VerSyKo.

Chapter 2

Prerequisites

In this section we recall those concepts, languages and tools that form the background for the work in VerSyKo that we present in the subsequent sections.

2.1 SCADE

The SCADE tool suite (Safety-Critical Application Development Environment) is a model-based development framework for safety-critical software. It supports certification of systems according to common standards from the aviation and rail transportation industries.

2.1.1 SCADE modelling language

The SCADE modeling language is a synchronous and dataflow-oriented language based on LUSTRE [52], and was extended by safe state machines [9]. The SCADE semantics is based on the "synchronous hypothesis" which states that the calculation of a model's clock cycle does not consume any time. Of course, this assumption is a simplification and cannot be fulfilled by real systems. Nevertheless, all logical operations and causal interrelations may be modelled. For the practical use, the simplifying assumption only implies the following restriction: The calculation time used by the model within one cycle must be smaller than the cycle time allocated to one clock cycle. The SCADE language is deterministic and there are no run-time effects. Each language construct has a graphical representation in the editor of the tool suite. An brief overview of the language constructs follows below (for further details see the SCADE tutorial and language reference):

Data: SCADE permits the handling of structured data. All data structures are static. It is possible to build structures and arrays from the basic types (real, int, bool). The modeling language is strongly typed.

Data-flow modeling: The SCADE modelling language includes the usual Boolean logic (and, or, not etc.) and arithmetic ($+$, $-$, $*$, etc.) operators to form expressions on dataflows. A conditional operator (if-then-else) allows to switch between two dataflows according to the value of a Boolean dataflow. In addition, the temporal operator fby ("followed by") allows to access data flow values of past clock cycles. There are operators for accessing structures and arrays including a dynamically indexed array access. However, there is no way of explicitly describing loops. Instead there are the "algebraic loops" Map and Fold, which are well-known in functional programming. Map is used to apply an operator with a certain input data type to an array of that data type element-wise. With Fold, calculations may be accumulated over an array. Fig. 2.1 shows an example of a simple SCADE model.

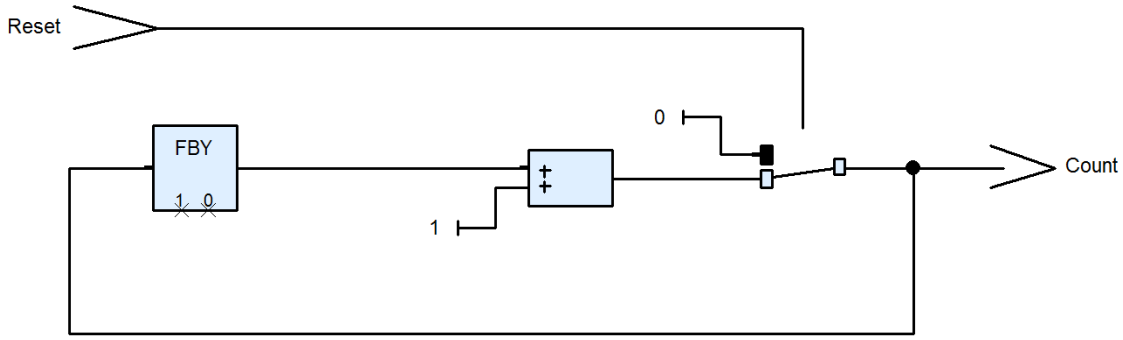

Figure 2.1: SCADE model of a simple counter

State machines: The state machines in SCADE are synchronously clocked, hierarchic state machines, so called safe state machines [9].

They look similar to UML state charts. However, an important difference is the strictly synchronous semantics: In each clock-cycle and in each (parallel) state machine precisely one state and precisely one transition is active and being carried out. Cyclical dependencies between model elements are not permitted and are, at code generation time, treated like syntax errors. Fig. 2.2 illustrates an example of a state machine. The two means of description, data flow and state machines, are fully integrated and thus may be combined in any way.

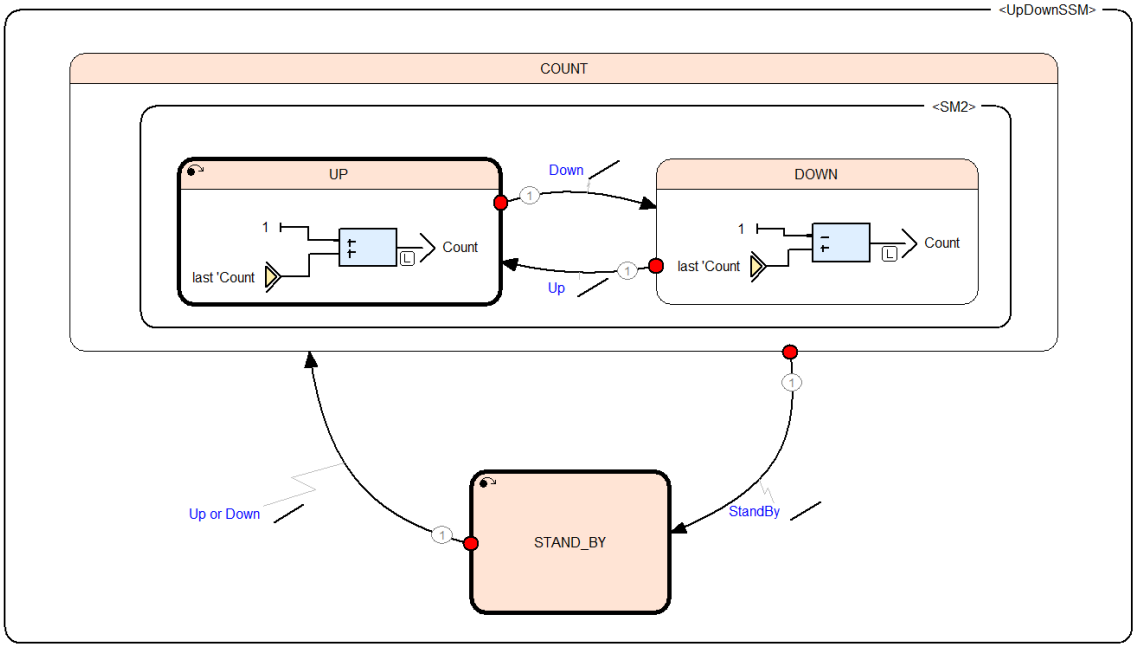

Figure 2.2: A state machine in SCADE

2.1.2 SCADE Tool Suite

The SCADE suite provides extensive tool support for the SCADE modelling language. We list the main features:

Graphical Editor: As already mentioned, each syntactic model element has a graphical representation. In the SCADE editor, these graphical elements can be used for modelling. Also, a textual modelling language may be used.

Code Generator: The code generator creates a C code from SCADE models. This code generator is qualified for the development of safety-related software in accordance with the standards DO-178B [37], up to Level A, and with CENELEC EN 50128 [27], up to SIL 3/4.

Simulator/Debugger: This feature allows running the code generated from a SCADE model, to test it on the model level and to debug it. The simulator may be controlled via TCL scripts and provides an automation interface.

Model Test Coverage: This tool component allows measuring the structural coverage of a given model by a test suite. SCADE models may automatically be instrumented for measurements according to two different coverage criteria: Decision Coverage (DC) and Modified Condition/Decision Coverage (MC/DC). Other criteria may be easily supplied by the user for (self-modelled) libraries.

Design Verifier (SCADE DV): This tool component allows the formal verification of safety properties of SCADE models. As this feature plays an important role for VerSyKo we will describe it a bit more detailed in the next subsection.

Gateways to other tools: The SCADE suite provides a number of gateways to exchange data with other tools. The requirements gateway permits the linking of SCADE model elements with their requirements which are, for instance, recorded in a tool such as DOORS. Also, classically coded parts of the software and test cases may be linked.

Other gateways link the SCADE suite with Rhapsody, Simulink and ARTiSAN.

2.1.3 Formal Verification using SCADE DV

The behavior of a SCADE design model can be given as a transition system $\mathcal{T} = \langle S, I, \rightarrow \rangle$, where S is a set of states, $I \subseteq S$ is a set of initial states and $\rightarrow \subseteq S \times S$ is the transition relation, see [2]. For efficiency reasons, the SCADE DV transforms a design model into a set of Boolean and linear arithmetic formulas that symbolically represent $\mathcal{T}$. If P denotes a *state predicate*, i.e. $s \models P \Leftrightarrow s \in P$, SAT-based model checking enables to verify *safety properties* [1]:

$$\forall s_0, s_1, \ldots s_n \in S : I \ni s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_n \Rightarrow s_n \in P.$$

In contrast to other approaches, the SCADE DV does not offer a temporal logic but properties have to be modeled as *synchronous observers* using the same language operators as for the design. This is illustrated in Fig. 2.3.

The synchronous observer is simply a SCADE operator having as input the inputs and outputs of the SCADE design to be verified and as output a Boolean data-flow. SCADE DV then verifies that this output is always `true` (i.e., the constant Boolean flow with value `true`). To perform formal verification SCADE DV uses the SAT based proof engine of Prover Technologies (`http://www.prover.com`). Notice, however, that more general temporal properties, notably unbounded liveness properties, cannot be automatically verified using this SAT-based approach.

[1] Safety properties express that the system always stays in a good or "safe" state.

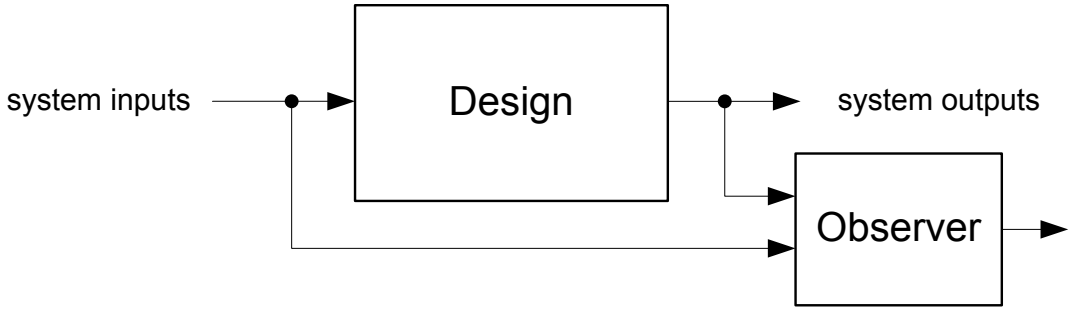

Figure 2.3: A state machine in SCADE

2.2 PROMELA/ SPIN

SPIN is a general tool for verifying the correctness of distributed software models in a rigorous and mostly automated fashion. It was written by Gerard J. Holzmann and others in the original Unix group of the Computing Sciences Research Center at Bell Labs, beginning in 1980. SPIN stands for *Simple* PROMELA *Interpreter*.

Systems to be verified are described in PROMELA (**Pro**cess / **Pro**tocol **Me**ta **La**nguage), which supports modeling of asynchronous distributed algorithms as non-deterministic automata [59]. Properties to be verified are expressed as Linear Temporal Logic (LTL) formulas, which are negated and then converted into Büchi automata as part of the model-checking algorithm. SPIN supports the verification of *safety* properties ("Something bad will never happen") as well as *liveness* properties ("Something good will eventually happen").

The SPIN tool uses an "explicit-state model checking" algorithm, i.e. it explores every reachable state of a system to verify whether the properties to be verified hold or not. If a property does not hold, then a counter-example in form of an execution trace leading to the illegal state is generated. In addition to model-checking, SPIN can also operate as a simulator, following one possible execution path through the system and presenting the resulting execution trace to the user.

SPIN also offers a large number of options to further optimize the model-checking process for speed and memory, such as:

- partial order reduction [47];
- state compression [58];
- bistate hashing (instead of storing whole states, only their hash code is remembered in a bitfield; this saves a lot of memory but voids completeness);
- weak fairness enforcement;
- use of slicing-techniques to allow distributed model-checking on multi-processor systems [60].

Unlike many model-checkers, SPIN does not actually perform model-checking itself, but instead generates C sources for a problem-specific model checker. This technique saves memory and improves performance, while also allowing the direct insertion of chunks of C code into the model.

2.3 Timed Automata and UPPAAL

UPPAAL is a model checking tool for dense real-time, developed jointly by UPPsala and AALborg university groups. First conceived in 1995, it has undergone a large number of optimizations and benchmark analyzes to make it applicable to a larger set of problems. Today it is widely used for educational purposes and has a sound reputation in industrial application.

The UPPAAL tool features a graphical editor, a simulator, and a verifier component. The verifier allows several user configurations on state space exploration, where the optimal configuration usually depends heavily on the system under investigation. Verification results can be used as input to the simulator, which allows to trace and explore unexpected model behavior.

The modeling language is an extension of the *timed automaton* formalism by Alur et al. [8]. UPPAAL supports parallel operation of several state machines, that can synchronize on global variables, synchronization channels (hand-shake), or time. States can be equipped with (downward closed) invariants on clock variables. Additional annotations – like urgent or committed states or channel urgency – are powerful concepts to steer the timing behavior of a network of state machines and effectively reduce the size of the state-space to be explored. In the recent years more and more C-style constructs were added to the model syntax, including arrays, structured data types, and user functions.

The logical language consists of a small subset of timed CTL (TCTL) [8]. Untimed *local properties* are based on the automaton syntax, i.e. are Boolean expressions over variable values, automaton locations, and clock equations. Temporal properties are constructed from local properties by application of temporal operators (in a restricted way). The strongest construct is the *unbounded response* ϕ --> ψ, which states that a state where local property ϕ holds is always (eventually) followed by a state where local property ψ holds. The possibility of adding observer automatons allows to verify a wide range of interesting system properties.

While a large number of optimization techniques have found application in the UPPAAL tool, the most prominent ones might be the following:

- avoiding of expansion to the full symbolic state space (called *region graph* in [8]); rather use a compact representation, which includes efficient data types for clock relation representation
- restriction of supported verification formulas to a subset of verifiable properties, that still allow for efficient model checking algorithms

The success of the tool is at least partially owed to the careful selection of modeling language extension, which do not break these.

2.4 Satisfiability Modulo Theories

The *Satisfiability Modulo Theories* (or *SMT* for short) problem is a decision problem for formulas which extend first-order logic with certain theories. Its instances are a generalization of the boolean satisfiability problem (called *SAT*), because every SAT problem is also a SMT problem. On a fundamental level, the SMT problem asks: "Given a formula φ, is there a valuation of the variables in φ such that φ becomes true?".

In recent years, computer programs (so called *solvers*) have become increasingly efficient in solving SMT problems. A proposed standard, to which many SMT solvers adhere, is called *SMTLib*[14]. It specifies a communication protocol based on LISP data-structures to formulate problems and receive answers from the solver. Not only can modern SMT solvers check the satisfiability of a given formula, they can also

- Provide a valuation for the variables of the formula which make the formula true.
- Extend or reduce the formula to facilitate incremental solving of problems.
- Declare new data types and functions which can then be used in formulas.

By using the SMTLib standard, application developers can make the SMT solver exchangeable and thus profit from the multitude of different SMT implementations currently existing.

2.5 LLVM

LLVM, which once was an acronym for "Low Level Virtual Machine" (now it's the full name of the project), is a collection of modular and reusable compiler and toolchain technologies. In the context of VerSyKo, two components of the LLVM framework are relevant:

1. The LLVM *intermediate representation* ("IR") language is high-level assembler language. Unlike most assembler languages, it is platform-independent and uses a static single assignment ("SSA") form instead of a register-based notation.
2. *Clang* is a compiler which can be used to translate C- and C++-programs into the LLVM-IR.

Chapter 3

GALS System-Level Validation

3.1 System-Level Verification Approach

As indicated in the introduction the VerSyKo approach to system-level verification is as follows (further details are described in Section 8):

- The **system-level verification goal** Φ is specified as a (timed) LTL formula expressing the desired behavior of the GALS system.
- The behavior of each synchronous component C in the GALS network is abstracted by its **contract** Φ_C which is expressed by an LTL formula (or other modeling techniques which are equivalent to expressing such a formula).
- From the network of contracts an **abstract GALS model** M_G is derived. This consists of a network of concurrent components C', such that each C' shows the most non-deterministic behavior still satisfying Φ_C.
- It is verified by means of property checking whether M_G satisfies Φ. We say that a **verification succeeds** if property checkig shows $M_G \models \Phi$ and the **verification fails**, otherwise.
- As additional verification artifacts, local test and verification results obtained during component verification and validation (V&V, for short) are provided as **guaranteed behavior** β_C by component suppliers. Again, β_C may be expressed as an LTL formula.

3.2 Verification Threats

Recall that the objective of **validation** is to check whether the system is adequate for its intended purpose, while **verification** checks the consistency of development artifacts with a reference specification (which may also be a model). The validation verdict is based on the collection of verification results achieved, in combination with specific validation activities, such as tests or formal verifications investigating the effectiveness of the system's operation in its intended environment[1].

From the point of view of GALS system validation the verification on system-level as investigated in this project poses three threats, which we call **verification threats (VTH)**. Potential root causes leading to these threats will be analyzed in the sections below.

[1] These tests and verifications complement the ones already performed during verification.

1. VTH 1: The verification fails because the implemented GALS system network is inadequate for the system-level specification. This is the "normal" verification, failure analysis and correction cycle that should finally lead to an improved system implementation.

2. VTH 2: The verification fails though the implemented GALS system network is *adequate* for its purpose. This situation is called a **false negative**: a correct system is rejected due to a verification failure which should not have occurred.

3. VTH 3: The verification succeeds though the implemented GALS system network is *inadequate* for its purpose. This situation is called a **false positive**: if not detected by system validation, an inadequate implementation will be accepted and become operative.

3.3 Root Causes for Verification Threats

In Fig. 3.1 a root cause analysis for the verification threats introduced above is shown in the form of a cause-consequence graph. Each arrow $A \rightarrow B$ in this figure has the meaning *"A may cause B"*. The root causes are represented in Fig. 3.1 by the grey-shaded boxes possessing outgoing arrows only. Their meaning is defined as follows:

- **Erroneous verification goal.** The GALS system-level verification goal Φ is inadequate for the ultimate goals of system validation, that is, the properties expressed by Φ are not the ones required for the intended purpose of the system.

- **Component modeling failure.** A synchronous component has been modeled in a way that the implied behavior is not adequate for the GALS system.

- **Manual implementation failure.** The component (or some part of it) has been programmed in a manual way, and during this process a failure was injected into the implementation.

- **Contract inconsistency.** The contract describes a behavior which is inconsistent with the true behavior of the component.

- **Contract weakness.** The contract is consistent with the true component behavior, but asserts weaker properties than the ones actually fulfilled by the component.

- **HW/SW integration failure.** During HW/SW integration a failure is injected, such as faulty linkage of relocatable code, insufficient register word length for the occurring mathematical operations or firmware and microcode errors.

If VTH 1 applies, the GALS system-level verification fails due to a "real" failure of the system. This must have been caused by at least one erroneous component. The component's failure may either be reflected already in its model, or it may have been injected in a manual implementation step not captured by the model. Observe that in any case the component's contract must have captured the erroneous behavior, because otherwise the GALS system-level verification would not have failed.

A false negative (VTH 2) may have been caused by an erroneous GALS system-level verification goal, or by inadequate contracts. Both inconsistent or weak contracts may cause false negatives: in the former case the contract implies erroneous component behavior though the component will really perform in an adequate way; in the latter case the contract is consistent with the true component

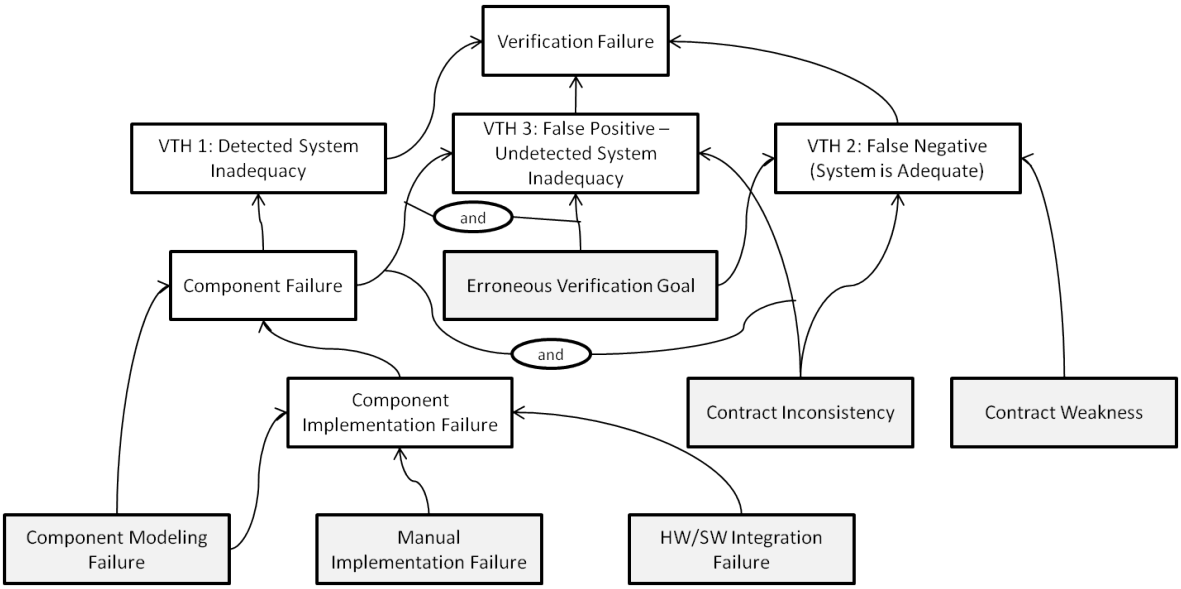

Figure 3.1: Root causes for verification threats.

behavior but too weak to prove the system-level verification goal. Typical variants of inadequate contracts will be discussed below.

Verification threat VTH 3 (false positive) applies in situations where at least one component is faulty but the system-level verification fails to detect this. This can be caused by an erroneous system-level verification goal or by a contract inconsistency which does not reflect a component's true faulty behavior. Observe that a contract weakness may never cause a false positive because it is always consistent with the true component behavior. Moreover, VTH 3 may be caused by a HW/SW integration failure which leads to component behavior which is even inconsistent with the software code.

3.4 Validation of System-Level Verification Results

From the perspective of system validation it is crucial to detect the presence of verification threats VTH 1, 2, 3 and abolish their causes. To this end, it has to be investigated first how to find out whether VTH 1, 2 or 3 are present in a verification result.

3.4.1 Detection of False Negatives

Since both VTH 1 and VTH 2 lead to a system-level verification failure, it has to be decided after occurrence of this failure which of the two threats applies. The following techniques can be applied to identify false negatives. The negative model checking result on system level is associated with an **error trace** $\pi = \langle s_0, s_1, s_2, \ldots, s_n \rangle$, where each s_i is a valuation function mapping each interface symbol addressed in the abstract GALS model to its current value. Since π is a legal finite computation trace of the model M_G and M_G only consists of components C satisfying their contracts Φ_C, computation π is consistent with each of these contracts,

$$\forall C : \pi \models \Phi_C.$$

Note that a contract Φ_C may not necessarily address all interface variables of C in its LTL formula. We require, however, that the contract interface specification contains C's complete interface. As a consequence, each interface symbol x of C is in the domain of the valuation functions s_i, but the

model checker may have assigned a random value to $s_i(x)$ if x is not further restricted by the LTL formula Φ_C.

False negatives caused by contract inconsistencies. Now suppose that the false negative has been caused by a contract inconsistency. This can be uncovered in the following ways:

- If some component C is associated with a complete formal model M_C which is consistent with the true behavior of C, but C's contract Φ_C is faulty, the inconsistency between Φ_C and its model M_C can be simply uncovered by "local" model checking of contract Φ_C against the model M_C, which will result in $M_C \not\models \Phi_C$.
- If no complete model M_C exists the guaranteed behavior β_C – if provided by the supplier – may be exploited: if $\pi \not\models \beta_C$ then the error trace π is inconsistent with the assertions β_C already established, so π is not a trace of C. There can be two root causes for this inconsistency.
  1. The contract Φ_C is faulty.
  2. A contract of another component C_0 is faulty and C_0 produces data consumed by C. The output data of C_0 in π is consistent with the faulty contract Φ_{C_0}, but violates the (correct) assumptions about admissible inputs in contract Φ_C.

False negatives caused by contract weaknesses. Next assume that the false negative has been caused by a contract weakness of component C. This can be uncovered as follows:

- If a complete formal model M_C of C exists it can be shown that π is not a trace of this model, $\pi \not\models M_C$.
- If guaranteed behavior β_C is defined for C, the contract can be refined by $\Phi_C \wedge \beta_C$ since the guaranteed behavior must be consistent with the contract. The refined contract may be harder to evaluate during model checking, but it may show that error trace π does not exist.

False negatives caused by erroneous system-level verification goals. If the false negative has been caused by an erroneous system-level verification goal Φ, this can be identified by validating the error trace π and finding out that this trace should really be considered as legal, i.e. π is a correct trace of the system composed of components C.

3.4.2 Detection of False Positives

Analyzing again the cause-consequence graph of Fig. 3.1 leading to false positives (VTH 3) yields the following case distinctions:

- If the false positive has been caused by contract inconsistencies while the system-level verification goal Φ is adequate, it is again necessary to uncover this inconsistency. As explained above in the case of false negatives, this can be detected by checking Φ_C against its model M_C and uncovering $M_C \not\models \Phi_C$ or by proving that $\Phi_C \wedge \beta_C$ has no solution.
- If the false positive has been caused by an inadequate system-level verification goal Φ only guaranteed behavior in combination with simulations may help to uncover the failure situation: a system-level computation π fulfilling Φ may contradict a guaranteed behavior β_C of some system component. Observe that C is not necessarily a component associated with a faulty

contract; the contradiction may have been caused by another component contributing to π in a way that violates the requirements of C.

3.4.3 Summary of Validation Obligations With Respect to Verification Results

Summarizing, the following activities should be performed to validate system-level verification results:

- All contracts Φ_C of components C associated with formal models M_C have to be checked with respect to validity of $M_C \models \Phi_C$.

  This task can be performed by model checking.

- All components C associated with guaranteed behaviors β_C have to be checked with respect to consistency between contract and guaranteed behavior. This means to prove that solutions of $\Phi_C \wedge \beta_C$ exist.

  This task can be performed by SAT or SMT-solving, respectively.

- All error traces π rejected by the system-level verification goal Φ have to be checked with respect to consistency with guaranteed behaviors, $\pi \models \beta_C$.

- All simulation traces π consistent with the system-level verification goal Φ have to be checked with respect to consistency with guaranteed behaviors, $\pi \models \beta_C$.

3.5 Contract Validation

3.5.1 Contract Insufficiencies

As elaborated above, contracts of local components in the GALS network may be insufficient in two ways.

- Contracts may be **inconsistent** to their underlying detailed models.
- Contracts may be **too weak** to allow for the verification of the global GALS verification goal.

In any case an insufficient contract may lead to verification failures (**false negatives** or **false positives**) when trying to prove the global GALS verification goal.

In the sections below we classify typical patterns of inconsistencies and weakness that we know from practical experience to occur in contract specifications. We will then explain how each of these insufficiencies can be uncovered, exploiting local and global model checking techniques as well as the guaranteed behavior assertions available in contract specifications.

A frequently occurring pattern for contracts represented by LTL formulas is

$$\mathbf{G}(\phi \Rightarrow \psi)$$

(*Whenever ϕ holds, ψ is ensured*). In particular, this pattern occurs in specifications for safety-critical systems. We will therefore analyze inconsistencies and weaknesses using this specification pattern.

3.5.2 Contract Inconsistencies

Table 3.1 below describes typical inconsistencies between contracts and their underlying detailed models.

No.	Correct Formula	Inconsistent Formula	Description
I1	$\mathbf{G}(\phi \Rightarrow \psi)$	$\mathbf{G}(\phi \Rightarrow \psi')$	It is erroneously assumed that condition ϕ has effect ψ', whereas the real effect expressed in the model is ψ. Formulas ψ and ψ' only differ in Boolean and/or arithmetic operators and/or brackets.
I2	$\mathbf{G}(\phi_1 \wedge \phi_2 \Rightarrow \psi)$	$\mathbf{G}(\phi_1 \Rightarrow \psi)$	A desired effect ψ is assumed to occur already under condition ϕ_1, while the detailed model only guarantees ψ if the stronger condition $\phi_1 \wedge \phi_2$ is fulfilled.
I3	$\mathbf{G}(\phi_1 \Rightarrow \psi_1)$	$\mathbf{G}(\phi_1 \Rightarrow \psi_1 \wedge \psi_2)$	A pre-condition ϕ_1 is assumed to achieve a stronger effect $\psi_1 \wedge \psi_2$ than what is really guaranteed by the model (ψ_1).
I4	$\mathbf{G}(\phi_1 \Rightarrow \psi_1 \vee \psi_2)$	$\mathbf{G}(\phi_1 \Rightarrow \psi_1)$	It is erroneously assumed that condition ϕ_1 always implies effect ψ_1, whereas the model will only guarantee ψ_1 to be among the *possible* effects, so ψ_2 may be observed instead of ψ_1.

Table 3.1: Typical inconsistencies in LTL contract specifications.

3.5.3 Contract Weaknesses

Table 3.2 describes typical weaknesses of contracts. This means that the contract specification is consistent with the underlying model, but it does not express all the properties which are necessary to prove the global GALS verification goal.

No.	Correct Formula	Weaker Formula	Description
W1	$\mathbf{G}(\phi \Rightarrow \psi_1 \wedge \psi_2)$	$\mathbf{G}(\phi \Rightarrow \psi_1)$	The model guarantees that ϕ will have effects $\psi_1 \wedge \psi_2$, while the contract only lists the effect ψ_1.
W2	$\mathbf{G}(\phi_1 \vee \phi_2 \Rightarrow \psi_1)$	$\mathbf{G}(\phi_1 \Rightarrow \psi_1)$	The model guarantees that effect ψ_1 will occur if either ϕ_1 or ϕ_2 are ensured. The contract, however, only captures occurrence of effect ψ_1 in case of precondition ϕ_1.
W3	$\mathbf{G}(\phi_1 \Rightarrow \psi_1)$	$\mathbf{G}(\phi_1 \wedge \phi_2 \Rightarrow \psi_1)$	The contract assumes that ψ_1 will only occur if both ϕ_1 and ϕ_2 are fulfilled, while the model already guarantees ψ_1 to occur if ϕ_1 holds.
W4	$\mathbf{G}(\phi_1 \Rightarrow \psi_1 \vee \psi_2)$ $\mathbf{G}(\phi_1 \wedge \phi_2 \Rightarrow \psi_1)$ $\mathbf{G}(\phi_1 \wedge \phi_3 \Rightarrow \psi_2)$	$\mathbf{G}(\phi_1 \Rightarrow \psi_1 \vee \psi_2)$	The contract only expresses (correctly) that condition ϕ_1 yields effect ψ_1 or effect ψ_2. For the global GALS verification, however, the stronger properties are required, which can guarantee *either* ψ_1 *or* ψ_2 under stricter conditions $\phi_1 \wedge \phi_2$ and $\phi_1 \wedge \phi_3$, respectively.

Table 3.2: Typical weaknesses in LTL contract specifications.

Chapter 4

Domain Specific Contract Modeling Language for Users

4.1 Objectives

As indicated in Fig. 1.1, the objective of the GALS translation language GTL (Section 5) is to provide a formalism for specification of contracts and system requirements which is close to the level of abstraction required by simulators, verification and test tools. In contrast to this, the **Contract Domain Specific Language (CDSL)** has the goal to facilitate the elaboration of contract networks and system requirements for the end user. The "ingredients" of the CDSL have been identified by analysis of the case studies presented in Sections 12.1 and 12.2 and other embedded system models used by Verified Systems International GmbH for the purpose of testing distributed embedded systems [80, 81]. While the expressive power of the CDSL is equivalent to that of the GTL, the former is syntactically richer, in order to provide adequate representation "instruments" for the different modeling objectives required from the end users' perspective. From the description of the CDSL $\rightarrow$ GTL transformation in Section 4.5 it will become apparent that these syntactic elements can all be mapped to the structural capabilities, formula and state machine expressions available in the GTL.

The syntactic elements of the CDSL have been motivated by the following insights.

- Some aspects of contracts depend on hidden states. These types of assertions are often more easily expressed by **state machines** than by LTL formulas.
- Though always representable by state machines or LTL formulas, it is often more convenient to specify stateless input-output relationships by means of **decision tables**.
- The efficiency of the verification process on GALS system level is facilitated and made more effective if contracts can optionally express **guaranteed behavior** about local components of the GALS network (see Section 4.3).
- A mixed graphical and textual representation of contract and system specifications is preferred by end users who are often familiar with formalisms like SCADE, UML or Matlab/Simulink.
- Graphical representations should not force users to represent multiple similar nodes and their interfaces in the GALS system network in an explicit way[1]. It is more suitable to depict "representatives" (or, formally, classes) of concrete object and interface collections in the graphical

[1] Just as users are reluctant to explicitly represent many instances if the same class in a UML object model.

contract network description and use tabular instantiation rules indicating how the network is unfolded by multiple instantiation of the representatives.

4.2 Abstract Language Description

In the following we sketch the CDSL language in more detail. We use EBNF-style[2] notation to explain the syntactic elements. The "$\oplus$" operator denotes a simple composition of elements.

<CONTRACTNETWORK> ::= <CONTRACTNET>
$\oplus$ <INTERFACE>$^+$
$\oplus$ <CONSTANTDEFINITIONS>
$\oplus$ <INSTANTIATION>

<CONTRACTNET> ::= $\underbrace{\text{<SYNCHRONOUSSTRUCT>}^+}_{\texttt{asynchronously connected}}$ $\oplus$ <CONNECTOR>*

<SYNCHRONOUSSTRUCT> ::= (<CONTRACTCLASS> | <COMPOSITESTRUCT>)
$\oplus$ <GUARANTEEDBEHAVIOUR>*

<COMPOSITESTRUCT> ::= $\underbrace{\text{<CONTRACTCLASS>}^+}_{\texttt{synchronously connected}}$ $\oplus$ <CONNECTOR>*

<CONTRACTCLASS> ::= <INPUTVARS> $\oplus$ <OUTPUTVARS> $\oplus$ <INTERNALVARS>
$\oplus$ <CLASSCONSTRAINT> $\oplus$ <CONTRACT>

<GUARANTEEDBEHAVIOUR> ::= <LTLFORMULA>

<CLASSCONSTRAINT> ::= <LTLFORMULA>*

<CONTRACT> ::= (<LTLFORMULA> | <STATEMACHINE>)

where

<CONSTANTDEFINITIONS>	is a set of symbolic constants (parameters),
<INTERFACE>	is the abstract definition of data flow (structured data),
<CONNECTOR>	is a directed arrow associated with an <INTERFACE>,
<*VARS>	is a set of variables (respectively input, output, or internal),
<STATEMACHINE>	is a state machine description without parallelism, and
<LTLFORMULA>	is one Linear-Time-Logic Formula interpreted over state machine states and variables

Figure 4.1: EBNF-style description of the CDSL, root element is <CONTRACTNETWORK>.

The crucial part of the language definition in Figure 4.1 is that all nodes in a contract net, i.e. the <SYNCHRONOUSSTRUCT> elements, are *asynchronously* connected with each other. One node of the net may be described as a *synchronous* composition of components. The guaranteed behavior is

[2]EBNF is a shorthand for Extended Backus-Naur Form, see e.g. [84].

given in form of LTL formulas (e.g. invariants), and can be accumulated (corresponding to logical "and").

A *contract class* follows the class stereotype. The different categories of variables are listed explicitly, since input and output variables correspond to the connectors, i.e., the *interfaces*. The behavioral restriction on a class—i.e., the contract— is made explicit either via an *LTL formula* or a *state machine* definition.

Similar to the contract class, the *interface* is seen as a template, that has to be instantiated appropriately. The *instantiation* commonly defines the concrete instances of classes and interfaces.

State Machines. State machines may decompose hierarchically, but without parallelism (architectural hierarchy). States (or super-states) can be annotated with entry- or do-actions, which can be defined either via assignment expressions or via *decision tables* for reasons of brevity.

Transitions can be guarded with boolean expressions that have to evaluate to true in order to enable a transition. The states can be equipped with constraints. A states cannot be entered if this would violate any constraint; a state has to be left if one of the contraints would be violated otherwise.

For our purpose, the state machines are deterministic in the sense that there in any state there is at most one transition that is enabled. If two or more transitions would be enabled according to guards and contraints, then *priorities* associated with transitions resolve this situation deterministically.

A complete example for the CDSL is given in connection with the case study in Section 12.1, see Figure 12.2 for the <CONTRACTNETWORK>.

4.3 Specifying and Exploiting Guaranteed Behavior

4.3.1 Guaranteed Behavior in Contract Specifications

Guaranteed behavior is a part of a contract specification expressing a property P of the local system which has been verified in a trustworthy way so that counter examples occurring during system verification and contradicting P imply that either

- C's contract has not been adequately specified, or,
- another component has previously violated the assumption made by C, so that C is no longer obliged to fulfill its contract.

The concept of guaranteed behavior naturally arises in a scenario where a large GALS system is integrated by a main contractor or system integrator, while the components are provided by different suppliers who have performed thorough verification, validation and test activities on their component, but without consideration of causal dependencies of the global system.

- The integrator specifies the contract network for the GALS system with the objective of GALS system verification. For this purpose the local contracts specified by the integrator for each node may be highly non-deterministic, that is, under-specified. Only those behavioral aspects of each component are captured that seem relevant for the system-level verification.
- The verification results obtained by the sub-contractor may be optionally added to each component's contract as guaranteed behavior to support validation of the contract network.

- Analyzing the guaranteed behavior, integrators may investigate which facts have already been established on local components, so that they do not have to be re-verified on system level. As a consequence integrators will direct their system-level verification activities to those which are not immediate consequences of guaranteed behaviors.
- If system-level verification implies the existence of system computations (arising, for example, from counter examples produced by a model checking tool) contradicting one or more guaranteed behavior assertions, integrators can conclude that their network of contracts has been inadequately specified (cf. the discussion in Section 3.4).

These aspects of guaranteed behavior utilization during system-level verification are explained in more detail in the sections to follow.

4.3.2 Patterns of Guaranteed Behavior

Guaranteed behavior may be based on

- formal verification of component models or/and component code, and,
- tests that have been performed with the software or with the integrated HW/SW system and that have passed their evaluation criteria.

When focusing on safety properties, the former case may be typically expressed by LTL properties of the form

$$\mathbf{G}(\phi \Rightarrow \psi)$$

(*"Whenever condition ϕ is fulfilled in a state of component C, the component will ensure ψ"*). The latter case is typically represented by formulas of the form

$$\mathbf{F}(\phi \wedge \psi)$$

(*"Finally the pre-condition ϕ for the given test purpose has been reached, and from there on the expected reaction ψ of C could be observed"*).

In many situations the application of the equivalence class principle can be applied, so that *all* system states fulfilling some condition ϕ are adequately represented by the set of system states fulfilling $\phi_1, \ldots, \phi_k$ which have been visited during the tests for the objectives $\mathbf{F}(\phi_i \wedge \psi_i), i = 1, \ldots, k$. If this is the case,

$$\bigwedge_{i=1}^{k} (\phi_i \Rightarrow \phi)$$

holds[3]. If the results ψ_i obtained in these tests are consistent with ψ, that is,

$$\bigwedge_{i=1}^{k} (\psi_i \Rightarrow \psi)$$

it is admissible, according to existing standards for safety-critical systems verification, to assume that also $\mathbf{G}(\phi \Rightarrow \psi)$ is guaranteed behavior, established by these tests.

[3] In rare cases the tests are even *exhaustive* in the sense that only k states exist satisfying ϕ, and these states are characterized by the conditions $\phi_i, i = 1, \ldots, k$.

4.4 Realization with Enterprise Architect

We employ the computer-aided software engineering tool *Enterprise Architect* (EA) [69] for the creation of CDSL models. While this is generally a tool used for round-trip engineering of software systems, we focus on its modeling facilities only. As such, *Enterprise Architect* is well suited for modeling tasks using the Unified Modeling Language UML 2.0.

We utilize *UML Composite Structure Diagrams* firstly to decompose a system of systems into individual systems running asynchronously, and secondly to decompose a subsystem into separate functional tasks running synchronously and in parallel. Hence the top-level composite structure diagram represents the globally asynchronous view of the entire system, each subsequent composite structure diagram will represent the functional decomposition of a locally synchronous system. The possibility to further decompose synchronous system components for the purpose of contract specification is motivated by the fact that some contracts are very hard to express as LTL formulas over the component viewed as a black box, while they can be elegantly expressed by a synchronous parallel composition of "smaller black boxes" and internal interfaces, each sub-component associated with an LTL formula or a state machine.

In this sense, *UML Classes* are used to represent the elements of the decomposition on each level. Classes may be annotated with attributes and methods to aid modeling of functionality. Since modeled classes representing subsystems may have multiple instances within the system of systems, attributes are also used to uniquely identify system instances as well as their respective inputs and outputs. Section 12.1.3 elaborates this.

UML Interfaces are used to describe data exchange between components on each decomposition level. Again, multiple instances of interfaces are possible on the global asynchronous level. Hence they may contain unique identifiers as well as their payload for data exchange.

In order to specify interface ownership and usage, classes must expose *UML Provided Interfaces* for ownership and *UML Required Interfaces* for usage. Typically, these exposed interfaces are parametrized over a class' unique identifier in order to specify data flow topology.

For readability and maintenance purposes, composite structure diagrams may declare *UML Enumerations* to define model constants.

The behavior of leaf components (i.e. classes without further decomposition into composite structures) is specified using *UML State Charts*. These are assumed to follow the semantics put forward by David Harel [56].

When applicable, the user may annotate the model with guaranteed behavior on all levels of decomposition. This is done by inserting the appropriate GTL-specifications into the model. Guaranteed behavior will then be represented as a (partial) textual description of a system or system component.

Furthermore, the overall verification goals need to be specified in order for a model to be of use. As with proven behavior, verification goals are currently given in textual GTL form. They are specified as an annotation to the entire model.

Example of a Contract Network. We illustrate the Enterprise Architect usage with parts of the the Level Crossing case study (explained in detail in Section 12.2).

In Fig. 4.2 part of the contract network is shown in UML modeling style. All visible Contract-Classes belong to the SynchronousStruct that describes the system under test.

The Connectors are represented by the interface stereotype. The association of class instance variables with the interface is resolved by the labelling of the corresponding ports. For example, the interface "`s_DATA0`" connects four instances of "`strassensignal`" with one instance of the "`ueberwachungssignal`". Consequently, it contains four integer values ("`values:int[4]`")

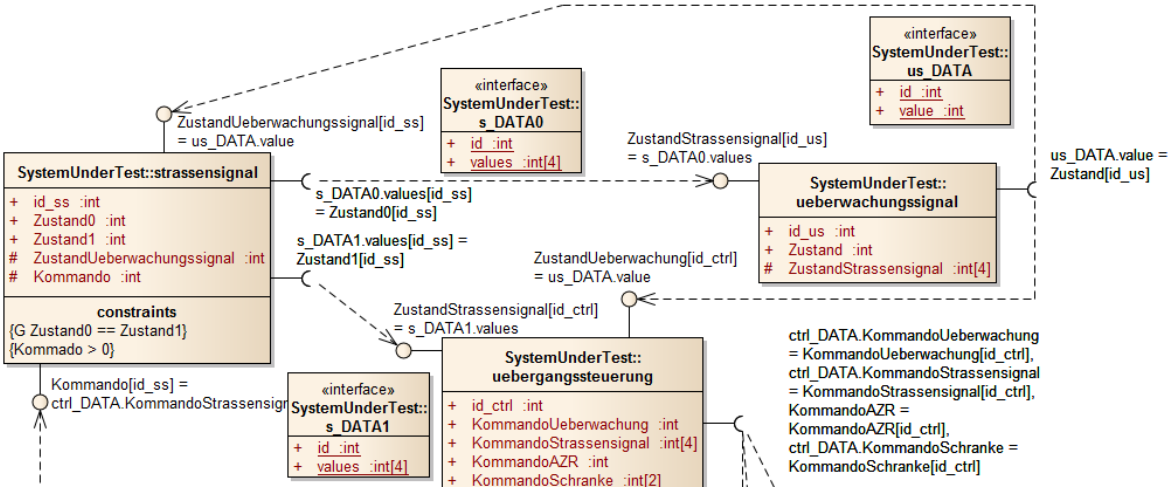

Figure 4.2: Part of the contract network corresponding to the level crossing case study.

where index `i` corresponds to instance `i` (0..3) of "`strassensignal`". This is made explicit by the association "`s_DATA0.value[id_ss] = Zustand0[id_ss]`".

On the side of the "`ueberwachungssignal`", the complete array of values is then associated with "`ZustandStrassensignal`" (of type "`int[4]`"). For the association "`ZustandStrassensignal[id_us]`", the index "`[id_us]`" is equivalent to "`[0]`", since there is only one instance of "`ueberwachungssignal`".

Note that the ContractClass "`strassensignal`" is associated with two ClassConstraints (in the diagram visible in the "`constraints`" section of the class). These LTL formulas restrict the valid behaviour in addition to the contract formulation which is provided as a state machine. Technically, this syntax is redundant, since every state machine can be expressed equivalently by an LTL formula which then can be combined with the ClassConstraint to a single LTL formula. However, ClassConstraints allow to organise the contract in a more human readable way.

Example of a State Machine. Fig. 4.3 shows the state machine that describes the contract of class "`strassensignal`" in the level crossing case study. The transitions are organised according to the inputs "`ZustandUeberwachungssignal[id_ss]`", "`Kommando`", "`UmgebungKfz`", "`Gestoert`", and "`GestoertTest`". The first two inputs originate from the system under test, the last three originate from the environment model. The outputs "`Zustand0`", and "`Zustand1`" are generated. Owing to class constraints, the value of both outputs is identical at any point in time.

The numbers associated with the transitions correspond to the priorities: always the enabled transition with the lowest number is taken.

Note that the inputs do not only occur in guard conditions but also in constraints associated with states. For example, state "`UnInit`" has to be left if input "`UmgebungKfz`" has a value different from "`StrasseTestOB`".

For a closer look at a complete example of a CDSL modeling formalism application in its current state, the reader is referred to section 12.1.2, which will focus on a concrete CDSL model developed within the context of our smoke detection protocol case study.

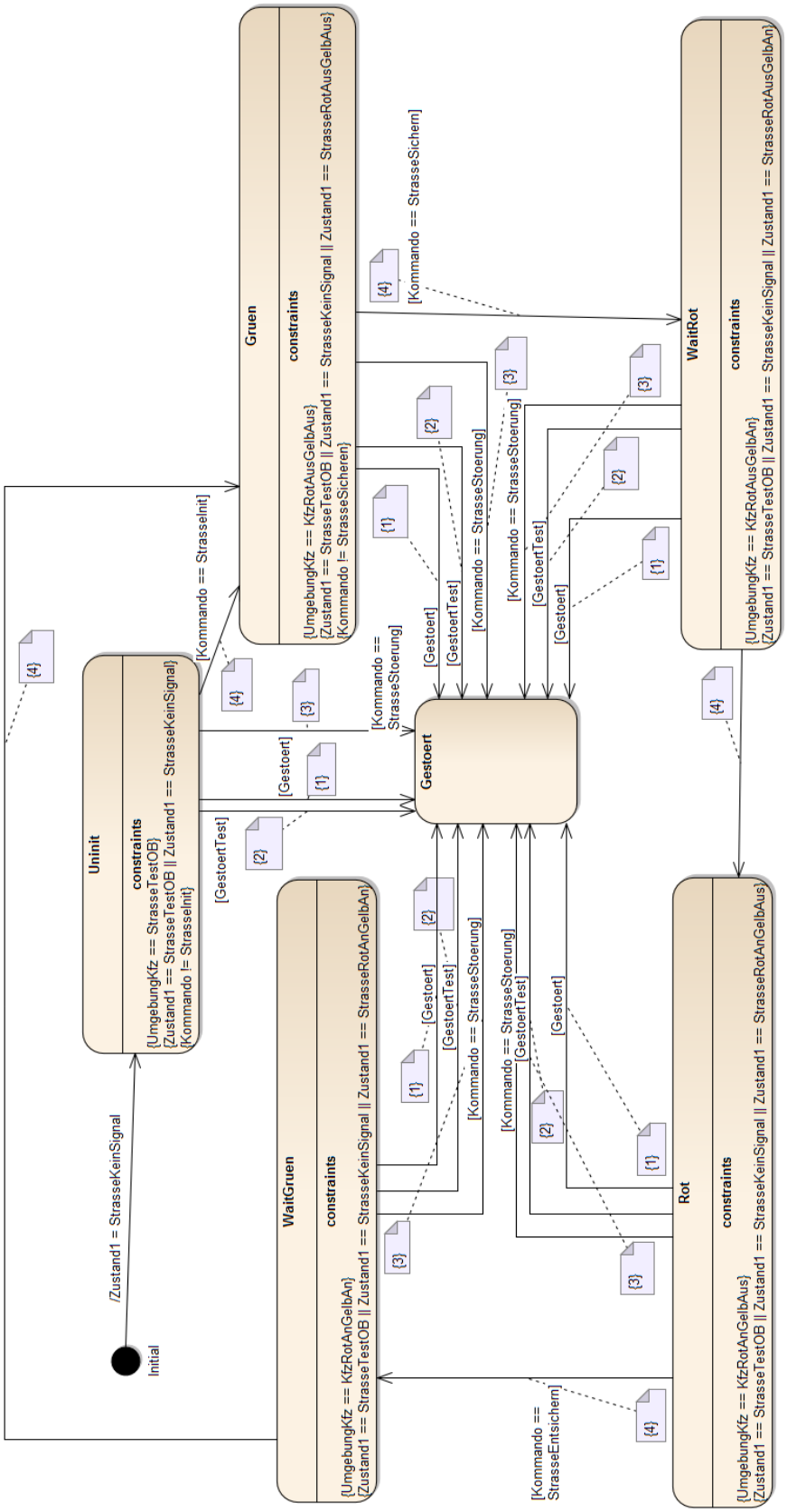

Figure 4.3: The contract of ContractClass "`strassensignal`" is described by a deterministic state machine. The numbers associated with transitions are priorities that resolve the non-determinism in case of two or more enabled transitions.

4.5 Rules for the CDSL → GTL Transformation

Part of the VerSyKo tool chain is the transformation of model descriptions to the (textual) GTL syntax, described in detail in chapter 5.

While transformations between formalisms in general can be complicated or even impossible, here this operation is theoretically sound: the CDSL is constructed with care, such that the expressive power does not exceed the expressiveness of GTL. The presence of an *instance concept* and behavioral descriptions via *automata* in both formalisms make the transformation rather straightforward.

In the following we outline the transformation, grouped into structural elements, transition elements, and annotations.

Translation Principles CDSL to GTL (structure).

- Each state machine is translated to one GTL model, that is specified via an automaton.
- Hierarchical state machines are flattened; this is straightforward, since parallelism is not present on lower-level state charts. Basic states of a state machine are translated to states of the automaton, including all invariants (if present) inherited from the source state or (hierarchical) parent states thereof.
- The instance concept is preserved, i.e., instances of state machines are translated to instances of models.

Translation Principles CDSL to GTL (transitions).

- Transitions between state chart states are translated to transitions between the corresponding (flat) automaton states; guard annotations are preserved.
- Transition *actions* are translated to *invariants* that are added to the corresponding automaton target state.
- For states with more than one incoming transition, it may be necessary to split up the corresponding automaton state into several states that share the same outgoing transitions.

Translation Principles CDSL to GTL (annotations).

- Entry/do actions of states are translated to invariants.
- Non-behavioral annotations, like verification goals, are translated literally.

In the smoke detection case study — which serves us as a guiding example — this transformation has been performed manually (see section 12.1.3).

Ultimately, the transformation will be performed by a *model parser*, which operates on the XMI-export of the UML model (see section 4.4).

Chapter 5

Textual Contract Specification Language GTL

In this section we describe the syntax and semantics of the specification language GTL (GALS translation language) we develop within VerSyKo to specify GALS systems. In GTL a GALS system is a network of synchronous components. Recall that we speak about *abstract GALS models* ("network of contracts") and *concrete GALS models* ("network of implementations"). The GTL is used to specify abstract GALS models.

We will first describe the syntax of the GTL in Sec. 5.1 and then present the syntax grammar in Sec. 5.2. Subsequently, we explain the data types that can be used in GTL specifications in Sec. 5.3. In Sec. 5.4 we give an informal description of the GTL semantics. Finally, we discuss restrictions of the GTL language in Sec. 5.5.

5.1 Syntax

To specify an abstract GALS model in the GTL, four elements are required:

model declarations Each synchronous component used in the specification has to be declared using a model declaration. A model declaration declares a class of components which can be instantiated. The model declaration specifies what formalism is used to describe the component (e.g. SCADE) as well as how to load the component specification (location of source files etc.). For example, the following snippet declares a component "car" which is written in SCADE in the file "car.scade" and its SCADE-node is named "ExampleCar":

```
1 model[scade] car("car.scade","ExampleCar");
```

The parameter list given to the model declaration depends on the synchronous formalism used. It is possible to have multiple models written in different formalisms.

Model declarations can also contain a body, in which contracts and other attributes can be specified, for example

```
1 model[scade] car("car.scade","ExampleCar") {
2   cycle time 40ms;
3   input int gear;
4   output float speed;
5 }
```

A body can contain:

- The cycle-time in which the synchronous model is able to perform one calculation step, e.g.:

```
1 cycle-time 40ms;
```

- Declaration of variables. A variable can either be input, output or local. A component reads from its input variables and writes to the output variables. Local variables are used to store internal state information. As GTL is a typed language, each variable declaration has to specify the type of the variable. For example, the following code declares x to be an input variable of type "int".

```
1 input int x;
```

  If the synchronous formalism supports extracting type information, neither input nor output variables have to be declared.

- Initialization values for variables, which can be used to give variables a value before the component performs a calculation step. The following assigns the variable x the initialization value 4:

```
1 init x 4;
```

  The assigned value must be of the same type as the variable itself (see Section 5.3).

- Contracts which describe an abstraction of the component's behavior. For example, the following contract guarantees that the value of *speed* will always be less that 100:

```
1 contract always (speed < 100);
```

  Contracts can use any construct described in Section 5.1.1. Contracts can be declared to be "guaranteed" which means that the correctness of the contract has already been established by other means (e.g. testing):

```
1 guaranteed contract always (speed < 100);
```

instances Declared components need to be instantiated in order to be used. This is similar to the concept of classes and instances known from object-oriented programming. It allows the user to reuse the same component without having to re-declare its contract every time. The following code declares c_1 and c_2 to be instances of the component "car":

```
1 instance car c1;
2 instance car c2;
```

connections A connection links an output variable of one component to an input variable of another one. This means that if the component writes onto this variable, the other component will read it in its next cycle. The following statement connects the variable *speed* from the component c_1 with the variable $current\_speed$ of the component cc:

```
1 connect c1.speed cc.current_speed;
```

It is required that the two connected variables are of the same type.

verification goals To specify a verification goal which has to be proved or disproved by the GTL-tool, the "verify"-construct is used. As with contracts, every construct from Section 5.1.1 may be used. The following snippet encodes the statement that a push on the brake always results in a speed reduction:

```
1 verify {
2   always c1.brake => exists x = c1.speed:
3     finally[50ms] (c1.speed < x);
4 }
```

5.1.1 Expressions

Expressions are used to specify contracts as well as verification goals. Expressions are composed from so called *atomic expressions*. An atomic expression is either a variable, a relation ("less-than", "greater-or-equal" etc.) between two arithmetic expressions or a constant.

An expression can have two possible forms:

Temporal formulas This provides the user with a LTL-like language to specify behavior. Within contracts in a component, the usual step-based interpretation of LTL is used. So for example a statement like

```
1 next x;
```

means that x will be true in the next clock cycle of the synchronous component. On the global level a GALS model performs a step whenever one of its components performs a step. This makes the *next* connective somewhat hard to use. So on the global level a time-based interpretation of temporal operators is useful. For example, the LTL operator "next" can be annotated with a time constraint like

```
1 next[150ms] x;
```

which means that for the next 150ms, x will be true. Notice that within contracts, time annotations may optionally be used, in which case they refer to the cycle-time of components. For instance, within the contract of a component with a cycle-time of $40ms$, the above time constraint specifies that x holds for the next 4 cycles (i.e. actually for $160ms$). Other temporal constructs are:

- The "until" operator states that a property holds at least until another property holds. If it is annotated with a time constraint it means that the second property has to hold in that time frame.
- "always" specifies that a property is always true. This corresponds to the LTL operator "globally"($\Box$).
- "finally" can be used to describe that a property will eventually hold. If it is annotated with a time, the property has to hold at least once within that time frame.

It is possible to refer to values of variables from a certain point in the past by binding them to a new variable:

```
1 exists x = p: next (x and q);
```

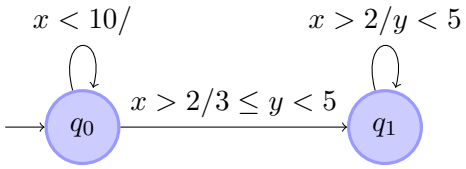

Figure 5.1: State-machine example

This specifies that both p and the next value of q are true.

State machines Some behaviors are easier formulated as a state machine than as a formula. In those cases one can declare a state machine as follows:

```
automaton {
  init state q0 {
    x < 10;
    transition q0;
    transition[y >= 3] q1;
  }
  state q1 {
    x > 2;
    y < 5;
    transition q1;
  }
}
```

If x is an input variable and y is an output variable, the state machine would declare a transition system as in Figure 5.1.

State machines can also use local variables.

The body of an automaton contains states which can be declared as follows:

```
state recv {
  ...
}
```

This declares a new state "recv". States can be made initial with the keyword "initial". States can contain atomic expressions or transitions. An atomic expression containing only input variables formulates a guard condition for the state, i. e., the state can only be entered if this guard condition is true (e. g., $x > 2$ in q_1 above, cf. Fig. 5.1 above). Atomic expressions containing output variables are constraints that determine possible values of these variables. Local variables can occur in guard conditions, output constraints, and they can (deterministically) be assigned new values.

A transition declares another state to be a possible successor to this one. Transitions may have guard conditions which must be fulfilled before the transition can occur. The following code declares a transition into the state "send" which can only occur if the variable x is less than 10.

```
transition[x<10] send;
```

Transitions may contain both atomic expressions on output variables and input variables. Again, an atomic expression containing only input and local variables (and which is not an assignment

of a local variable) is a transition guard. Atomic expressions containing outputs or which are assignments of local variables determine the values of these variables.

5.2 Grammar

A GTL-specification file consists of a list of declarations:

⟨*gtl_spec*⟩ ::= ⟨*declaration*⟩*

Each declaration can be either a component declaration, an instance declaration, a connection or a verification goal:

⟨*declaration*⟩ ::= ⟨*model_decl*⟩
| ⟨*instance_decl*⟩
| ⟨*connect_decl*⟩
| ⟨*verify_decl*⟩

A component declaration consists of the name of the synchronous back end formalism (for now, only SCADE), its name, a list of arguments and its contract. The arguments specify how to load the synchronous model specification. For example in SCADE, it needs the filename in which the model is specified and the name of the SCADE-node which implements the model.

⟨*model_decl*⟩ ::= '`model`' '`[`' ⟨*id*⟩ '`]`' ⟨*id*⟩ '`(`' (⟨*string*⟩ ('`,`' ⟨*string*⟩)*)? '`)`' ⟨*model_contract*⟩

The model contract can either be empty or a list of contract bodies:

⟨*model_contract*⟩ ::= '`;`'
| '`{`' ⟨*contract_body*⟩* '`}`'

A contract body is either a variable declaration, a contract formula, an initialization value for a variable or an automaton declaration:

⟨*contract_body*⟩ ::= ⟨*direction*⟩ ⟨*type*⟩ ⟨*id*⟩ '`;`'
| '`guaranteed`'? '`contract`'? ⟨*formula*⟩ '`;`'
| '`init`' ⟨*id*⟩ ⟨*value*⟩ '`;`'
| '`cycle-time`' ⟨*time_spec*⟩ '`;`'
| '`automaton`' '`{`' (⟨*state*⟩)* '`}`'

A variable can be an input, output or be local to the model:

⟨*direction*⟩ ::= '`input`'
| '`output`'
| '`local`'

GTL supports many common types:

⟨*type*⟩ ::= '`int`'
| '`byte`'
| '`bool`'
| '`float`'
| '`enum`' '`{`' ⟨*id*⟩ ('`,`' ⟨*id*⟩)* '`}`'
| ⟨*type*⟩ '`^`' ⟨*int*⟩
| '`(`' ⟨*type*⟩ ('`,`' ⟨*type*⟩)* '`)`'

Variable values match the available types:

⟨*value*⟩ ::= '−'? ['0'-'9']+ ('.' ['0'-'9']+)?
| 'true'
| 'false'
| ''' ⟨*id*⟩
| '[' (⟨*value*⟩ (',' ⟨*value*⟩)*)? ']'
| '(' (⟨*value*⟩ (',' ⟨*value*⟩)*)? ')'

Variables can be qualified or unqualified names:

⟨*var*⟩ ::= ⟨*id*⟩ ⟨*index*⟩
| ⟨*id*⟩ '.' ⟨*id*⟩ ⟨*index*⟩

⟨*index*⟩ ::= ('[' ⟨*int*⟩ ']')*

Formulas are either atomic propositions (see below) or linear temporal logic formulas including existential quantifications:

⟨*formula*⟩ ::= ⟨*atom*⟩
| 'not' ⟨*formula*⟩
| ⟨*formula*⟩ 'and' ⟨*formula*⟩
| ⟨*formula*⟩ 'or' ⟨*formula*⟩
| ⟨*formula*⟩ 'implies' ⟨*formula*⟩
| 'always' ⟨*formula*⟩
| 'next' ('[' ⟨*time_spec*⟩ ']')? ⟨*formula*⟩
| 'finally' ('[' ⟨*time_spec*⟩ ']')? ⟨*formula*⟩
| ⟨*formula*⟩ 'until' ('[' ⟨*time_spec*⟩ ']')? ⟨*formula*⟩
| 'exists' ⟨*id*⟩ '=' ⟨*var*⟩ ':' ⟨*formula*⟩
| '(' ⟨*formula*⟩ ')'

Atomic propositions can be variables, values or relations:

⟨*atom*⟩ ::= ⟨*var*⟩
| ⟨*value*⟩
| ⟨*expr*⟩ '<' ⟨*expr*⟩
| ⟨*expr*⟩ '>' ⟨*expr*⟩
| ⟨*expr*⟩ '<=' ⟨*expr*⟩
| ⟨*expr*⟩ '>=' ⟨*expr*⟩
| ⟨*expr*⟩ '=' ⟨*expr*⟩
| ⟨*var*⟩ 'in' '{' (⟨*value*⟩ (',' ⟨*value*⟩)*)? '}'

Expressions can also be variables, values or arithmetic terms:

⟨*expr*⟩ ::= ⟨*var*⟩
| ⟨*value*⟩
| ⟨*expr*⟩ '+' ⟨*expr*⟩
| ⟨*expr*⟩ '−' ⟨*expr*⟩
| ⟨*expr*⟩ '*' ⟨*expr*⟩
| ⟨*expr*⟩ '/' ⟨*expr*⟩
| '(' ⟨*expr*⟩ ')'

States consist of state content. They can optionally be declared as initial states:

⟨*state*⟩ ::= ‘`initial`’? ‘`state`’ ⟨*id*⟩ ‘`{`’ (⟨*state_content*⟩ ‘`;`’)* ‘`}`’

A state content is either a formula that has to hold in this state or a transition declaration into another state, possibly annotated with a guard condition:

⟨*state_content*⟩ ::= ⟨*formula*⟩
| ‘`transition`’ (‘`[`’ ⟨*formula*⟩ ‘`]`’)? ⟨*id*⟩

Instances are declared by specifying the model and the name of the instance. Optionally, they can have additional contracts:

⟨*instance_decl*⟩ ::= ‘`instance`’ ⟨*id*⟩ ⟨*id*⟩ ⟨*instance_body*⟩

⟨*instance_body*⟩ ::= ‘`;`’
| ‘`{`’ (⟨*formula*⟩ ‘`;`’)* ‘`}`’

Connections are specified by giving the source component and variable and target component and variable:

⟨*connection_decl*⟩ ::= ‘`connection`’ ⟨*id*⟩ ‘`.`’ ⟨*id*⟩ ⟨*index*⟩ ⟨*id*⟩ ‘`.`’ ⟨*id*⟩ ⟨*index*⟩ ‘`;`’

Verification goals are simply formulas:

⟨*verify_decl*⟩ ::= ‘`verify`’ ‘`{`’ (⟨*formula*⟩ ‘`;`’)* ‘`}`’

Time specifications are a number followed by a time-unit, which can also be given in cycle counts:

⟨*time_spec*⟩ ::= ⟨*int*⟩ ⟨*time_unit*⟩

⟨*time_unit*⟩ ::= ‘`s`’
| ‘`ms`’
| ‘`us`’
| ‘`cy`’

5.3 Data types

The following data types are supported:

int 64bit unsigned integer. Range from -2^{63} to $(2^{63}-1)$.

byte 8bit unsigned integer. Range from -128 to 127.

float IEEE 754 double precision (64bit) floating point numbers.

enum Simple enumeration types which consist of a list of possible values. Each value is distinct from every other value. Enumeration types are equal if they have exactly the same values. For example, to declare an input variable “color” which can have the three values “red”, “green” or “blue”, one writes:

```
1 input enum { red, green, blue } color;
```

To specify that the variable has the value "blue" one can use an enumeration constant by prefixing an allowed value with a quote:

```
color = 'blue;
```

array Arrays are composite types which replicate a given type a given number of times. To declare an array, one uses the power operator "^" for types:

```
output int^3 xs;
```

This would declare xs to be an array of three integer values. Array values are constructed as follows:

```
xs = [3, 5, 6];
```

Arrays can be indexed using the familiar C-syntax:

```
xs[1] > 4;
```

This would specify that the second element in the array xs must be greater than 4. Note that this syntax is also available in connections.

tuple Tuples are similar to arrays, but while arrays are restricted to replicate a given data type, tuples can be composed from any number of data types. A tuple type is declared as such:

```
local (int,bool) x;
```

This would declare x to be a tuple consisting of an integer component and a Boolean component.

5.4 Semantics

In this document we will not give a formal semantics of GTL but rather describe the semantics informally. For a fragment of GTL (without automata, local variables and timed constraints) a formal semantics is described in [51]. At the moment we leave the formal semantics of the complete language as described here for future work.

Definition 1. *We denote by Id the set of available identifiers.*

In a practical implementation Id will usually be the set of all strings, minus reserved grammar keywords.

5.4.1 Type system

Definition 2. *Let Λ be the set of all types as defined in the grammar (see Section 5.3). We define the function $[\![]\!]_T$, mapping types to their domain[1] as follows:*

$$[\![]\!]_T : \Lambda \to Set$$

[1] The domain for floats is not really reals but IEEE 754 64bit doubles.

$$
\begin{aligned}
[\![\mathit{int}]\!]_T &= \{-2^{63}, \ldots, 2^{63}-1\} \\
[\![\mathit{bool}]\!]_T &= \mathbb{B} \\
[\![\mathit{byte}]\!]_T &= \{-128, \ldots, 127\} \\
[\![\mathit{float}]\!]_T &= \mathbb{R} \\
[\![\mathit{enum}\{id_1, \ldots, id_n\}]\!]_T &= \{id_1, \ldots, id_n\} \\
[\![t^n]\!]_T &= ([\![t]\!]_T)^n \\
[\![(t_1, \ldots, t_n)]\!] &= [\![t_1]\!]_T \times \cdots \times [\![t_n]\!]_T
\end{aligned}
$$

To define the type system we will specify which type bindings are correct for a given program.

Definition 3. *Let* Γ *be a* type binding*, i. e., a mapping from variables of a* GTL*-specification to their corresponding type and their "function" which can be input, output or local variable. The variables are encoded by strings of the form* $c.v$ *where* v *is the name of the variable and* c *the name of its component. For local variables, the component name is omitted. For undefined variables,* Γ *returns* $\bot$*. Thus* Γ *has the signature*

$$\Gamma : Id \times Id \cup Id \to (\Lambda \cup \{\bot\}) \times \{input, output, local\}$$

We are now ready to define the property of being *well-typed* for a given GTL specification. Below we define the typing relation $\vdash$ by structural induction; it has the signature

$$\vdash\!: (Id \times Id \cup Id \to (\Lambda \cup \{\bot\}) \times \{input, output, local\}) \times (\Lambda \cup \{\top\}) \times \mathcal{L}_{GTL},$$

where $\mathcal{L}_{GTL}$ is the set of GTL specifications generated by the grammar from Section 5.2.

Notation 4. *Instead of* $(\Gamma, t, \alpha) \in \vdash$*, we write*

$$\Gamma \vdash \alpha : t.$$

If a language construct is un-typed, i. e., it has the type $\top$*, we omit this type and simply write*

$$\Gamma \vdash \alpha.$$

Definition 5. *A textual model* α *is* well-typed *if there exists a valid type-mapping* Γ *for it and its type is* $\top$*.*

A GTL-model is well-typed if each declaration in it is well-typed and the type mapping does only contain qualified variables:

$$\text{all}\frac{\Gamma \vdash d_1 \quad \ldots \quad \Gamma \vdash d_n \quad \neg\exists v \in Id : \Gamma(v) \neq \bot}{\Gamma \vdash d_1 \ldots d_n}$$

A component declaration is well-typed if all its body elements are valid for the binding Γ' which consists of all variables which are local to the declared component:

$$\text{model}\frac{\Gamma' \vdash c_1 \quad \ldots \quad \Gamma' \vdash c_n}{\Gamma \vdash \textbf{model}[\beta]\ c(args)\{c_1\textbf{;}\ldots\textbf{;}c_n\}}$$

Here, Γ' is the function which makes all variables from Γ local to the component c:

$$\Gamma'(x) = \begin{cases} \Gamma(c, x) & x \in Id \\ \bot & x \in Id \times Id \end{cases}$$

A type declaration must conform to the type given in Γ:

$$\text{decl}\frac{v \in Id \qquad \Gamma(v) = (t, d)}{\Gamma \vdash d\ t\ v}$$

An initialization value must be of the corresponding type:

$$\text{init}\frac{v \in Id \qquad \Gamma(v) = (t, \_) \qquad i \in [\![t]\!]_T}{\Gamma \vdash \textbf{init}\ v\ i}$$

A contract formula must be of type *bool* and be valid according to the type binding Γ:

$$\text{contract}\frac{\Gamma \vdash \pi : \text{bool}}{\Gamma \vdash \textbf{contract}\ \pi}$$

An instance of a component must be compatible to the type-bindings of the component:

$$\text{instance}\frac{c \in Id \qquad i \in Id \qquad \forall v \in Id : \Gamma(c, v) = r \Leftrightarrow \Gamma(i, v) = r}{\Gamma \vdash \textbf{instance}\ c\ i}$$

A connection may only be established between variables of the same type:

$$\text{connection}\frac{\begin{array}{cc}\Gamma(i_1, v_1) = (\_, \text{output}) & \Gamma \vdash i_1.v_1\ c_1 : t \\ \Gamma(i_2, v_2) = (\_, \text{input}) & \Gamma \vdash i_2.v_2\ c_2 : t\end{array}}{\Gamma \vdash \textbf{connect}\ i_1.v_1\ c_1\ i_2.v_2\ c_2}$$

Verification goals must contain well-typed Boolean expressions:

$$\text{verify}\frac{\Gamma \vdash v_1 : \text{bool} \qquad \ldots \qquad \Gamma \vdash v_n : \text{bool}}{\Gamma \vdash \textbf{verify}\{v_1; \ldots; v_n\}}$$

A variable simply has its associated type and can be either qualified (in a verification goal) or unqualified (in a contract):

$$\text{var1}\frac{c \in Id \qquad v \in Id \qquad \Gamma(c, v) = (t, \_)}{\Gamma \vdash c.v : t} \qquad \text{var2}\frac{v \in Id \qquad \Gamma(v) = (t, \_)}{\Gamma \vdash v : t}$$

Existential quantifier create a new variable of the same type as the bound variable:

$$\text{exists}\frac{x \in Id \qquad v \in Id \times Id \cup Id \qquad \Gamma(v) = (t, \_) \qquad \Gamma \cup \{(x, t)\} \vdash e : t'}{\Gamma \vdash \textbf{exists}\ x = v\text{: } e : t'}$$

Constants follow:

$$\text{const}\frac{t \in \Lambda \qquad l \in [\![t]\!]_T}{\Gamma \vdash l : t}$$

Expressions may be indexed:

$$\text{index}\frac{i \in \mathbb{N} \qquad \Gamma \vdash e : (t_1, \ldots, t_i, \ldots, t_n)}{\Gamma \vdash e\textbf{[}i\textbf{]} : t_i}$$

Logic operators have the usual type semantics:

$$\text{not}\frac{\Gamma \vdash l : \text{bool}}{\Gamma \vdash \textbf{not}\ l : \text{bool}} \qquad \text{and}\frac{\Gamma \vdash l : \text{bool} \qquad \Gamma \vdash r : \text{bool}}{\Gamma \vdash l\ \textbf{and}\ r : \text{bool}}$$

$$\text{or}\frac{\Gamma \vdash l : \text{bool} \qquad \Gamma \vdash r : \text{bool}}{\Gamma \vdash l\ \textbf{or}\ r : \text{bool}} \qquad \text{implies}\frac{\Gamma \vdash l : \text{bool} \qquad \Gamma \vdash r : \text{bool}}{\Gamma \vdash l\ \textbf{implies}\ r : \text{bool}}$$

Temporal operators work in a similar fashion (note that time constraints are not handled here because they do not change the types):

$$\text{next}\frac{\Gamma \vdash l : \text{bool}}{\Gamma \vdash \textbf{next}\ l : \text{bool}} \qquad \text{always}\frac{\Gamma \vdash l : \text{bool}}{\Gamma \vdash \textbf{always}\ l : \text{bool}}$$

$$\text{finally}\frac{\Gamma \vdash l : \text{bool}}{\Gamma \vdash \textbf{finally}\ l : \text{bool}} \qquad \text{until}\frac{\Gamma \vdash l : \text{bool} \quad \Gamma \vdash r : \text{bool}}{\Gamma \vdash l\ \textbf{until}\ r : \text{bool}}$$

Relations work only on equal types:

$$\text{equal}\frac{\Gamma \vdash l : t \quad \Gamma \vdash r : t}{\Gamma \vdash l = r : \text{bool}} \qquad \text{nequal}\frac{\Gamma \vdash l : t \quad \Gamma \vdash r : t}{\Gamma \vdash l\ \textbf{!=}\ r : \text{bool}}$$

$$\text{lesser}\frac{\Gamma \vdash l : \text{int} \quad \Gamma \vdash r : \text{int}}{\Gamma \vdash l\ <\ r : \text{bool}} \qquad \text{greater}\frac{\Gamma \vdash l : \text{int} \quad \Gamma \vdash r : \text{int}}{\Gamma \vdash l\ >\ r : \text{bool}}$$

Arithmetic operators only work on integers:

$$\text{plus}\frac{\Gamma \vdash l : \text{int} \quad \Gamma \vdash r : \text{int}}{\Gamma \vdash l + r : \text{int}} \qquad \text{minus}\frac{\Gamma \vdash l : \text{int} \quad \Gamma \vdash r : \text{int}}{\Gamma \vdash l - r : \text{int}}$$

$$\text{mult}\frac{\Gamma \vdash l : \text{int} \quad \Gamma \vdash r : \text{int}}{\Gamma \vdash l * r : \text{int}} \qquad \text{div}\frac{\Gamma \vdash l : \text{int} \quad \Gamma \vdash r : \text{int}}{\Gamma \vdash l / r : \text{int}}$$

State machines are also valid Boolean expressions, if all states are valid:

$$\text{automaton}\frac{\Gamma \vdash s_1 \quad \ldots \quad \Gamma \vdash s_n}{\Gamma \vdash \textbf{automaton}\ \{s_1\textbf{;}\ldots\textbf{;}s_n\} : \text{bool}}$$

A state is valid if all guards are valid and each transition label is well-typed:

$$\text{state}\frac{\Gamma \vdash c_1 : \text{bool} \quad \ldots \quad \Gamma \vdash c_n : \text{bool}}{\Gamma \vdash \textbf{state}\ n\ \{c_1\textbf{;}\ldots\textbf{;}c_n\}}$$

Transitions must have well typed guards (or no guard at all):

$$\text{transition}\frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash \textbf{transition}[e]\ n : \text{bool}}$$

5.4.2 Semantics of SCADE-components

Being described in a synchronous language, every SCADE-component in a GTL-specification can be written as a deterministic mealy machine. The set of states is the assignment of values to all flows in the SCADE model. The type of the input for the mealy machine is a vector of all input variable types. For example if the component has two *int*-inputs and one *bool*-input then the input type for the mealy machine is $[\![int]\!]_T \times [\![int]\!]_T \times [\![bool]\!]_T$. The same construction applies for the vector of output types. The components have a finite internal state space that is defined by the synchronous formalism.

In every step that a component performs, values are read from the input variables and depending on them and the internal state, output variables are assigned to new values and a new internal state is entered.

5.4.3 Semantics of GTL-components

If contracts are used to abstract the concrete component implementation, the components become non-deterministic mealy machines, because the contracts specify more behavior of the components. Every contract is transformed into a Mealy automaton: for a contract given by a LTL-formula, we adapt the translation algorithm from [42], and for a contract that is already given by a state machine the transformation into an automaton is straightforward; see Section 7.1 for some details. If a component has more than one contract its semantics is the automaton obtained by forming the product automaton of the automata obtained from the contracts.

The synchronous behavior of this automaton is obtained as follows: in every synchronous cycle the automaton performs one step. The length of the cycles of a component is specified by the cycle time statement and is used when giving the semantics of the composition of the components of a GTL-specification as an abstract GALS model in the next section.

Because the verification targets do not (yet) restrict the execution of contracts to the runs where some Büchi acceptance condition is met, the final states of the resulting automata can be ignored and the automaton can be treated as a non-deterministic Mealy machine.

5.4.4 Semantics of GALS models

A GTL-model is interpreted as a network of synchronous mealy machines (either deterministic or non-deterministic, depending on the usage of contracts to abstract components). The connections between components can be interpreted in different ways:

- The simplest one is a shared-memory abstraction that views connections as being loss-less and instantaneous. This can be realized by creating a variable for each connection that is being written by one component and read by another. Even though this is easy to implement, it has the drawback that it might not be very realistic for some applications and it can result in values being overwritten before they can be consumed by the reading process.

- More complex interpretations could consider factors like message loss or message delay. With message loss, the user could specify that a connection is lossy which would result in simulations where messages are non-deterministically dropped from the connection. To prevent the loss of all messages, certain fairness conditions might be interesting to consider. Message delay would either result in a constant delay that is added to every message transmission (deterministic delay) or a user specified "delay-range" in which a message might be delivered.

There are also several options available to interpret the execution strategy for the components:

- By allowing full asynchronicity, each process is allowed to do a step at every time. This interpretation is very easy to implement, but it is also very unrealistic, as it allows one component to "starve" every other component by not allowing them to make a step.

- Fairness constraints can be used to prevent a component from starving. While this prevents the worst execution-cases from being considered for the verification, it still might not be good enough to achieve realistic verification results, as many unrealistic cases are still being considered (e.g. one process doing 1000 cycles while another one only performs one).

- A *scheduling* of the components' execution may keep track of the execution time for each component and only allow a component to make a step when certain conditions are fulfilled. For example only the component with the least amount of execution time can be allowed to execute

in each step effectively disabling asynchronicity. Alternatively, components significantly ahead of all other components may be slowed down by denying them execution steps.

For the prototypical model transformations described in the next section we take a transformational semantics that (a) uses shared memory abstraction and (b) implements scheduling that lets synchronous components progress synchronously with a global clock. In this scheduling we assume that there is a global dense time with which all components are synchronized. This is the reasonable realistic assumption that cycles of all real components refer to the same global time (i. e., the current time is the same for all components).[2] The scheduling then lets components make steps according to their cycle time. But globally the components are running asynchronously. In this scheduling we also assume (as a simplification) that all components start at the same time. A more complicated variant lets components start non-deterministically within a certain specified time period.

5.5 Restrictions of GTL specifications

In this section we list a number of restrictions of GTL specifications. Some of them are dictated by the semantics that we have described and some of them are restrictions imposed by the model transformations described in the next section. These latter restrictions might be lifted in the future as implementations of the model transformations are improved.

Only one output variable per relation in contracts. For contracts of a component that are specified by an LTL formula, every atomic expression containing a relation (e. g. $<, =$ etc.) has to contain at most one output variable.[3] The reason for this is that when two or more output variables are involved it is not obvious how to define their range of values. For example, given two output variables x and y with possible set of values V_x and V_y, what is their set of variables after we specify that $x < y$.

This restriction applies only to the PROMELA and UPPAAL targets. The SMT target can handle any kind of contract expression.

Verification goals must not refer to cycle counts. Indeed, on the global level synchronous cycles have no meaning. Verification goals should use temporal connectives with a time constraint referring to the global dense time. They can also use ordinary temporal connectives which refer to the steps of the global GALS model, which makes a step whenever one of its components is making a step.

Local variables can only be assigned deterministically. The reason for this is that local variables can be read and written. So an expression like $x < y$, where x is a local variable and y an input variable is ambiguous: this could be checking whether y is greater then the current value of x or it could be an (non-deterministic) assignment setting x to any value less than y. This restriction can be lifted if one syntactically distinguishes read and write access to local variables.

LTL contracts may only use bounded temporal connectives. This restriction is imposed by the SCADE DV. Since synchronous observers are defined using the SCADE language constructs. So since SCADE does not have any unbounded temporal connectives LTL formulas containing "*finally*" or "*until*" cannot be translated to SCADE. This restriction may be lifted when a different verification tools than SCADE DV is used for local verification. This will be investigated in WP 4 of VerSyKo.

[2] An instance of this is are the clocks in German railway stations that always show the same time.

[3] Currently, this one output variable even has to be alone on one side of the relation, e. g. $o > i + 42$ but not $o - i > 42$.

Chapter 6

This page has been left intentionally blank

Chapter 7

Model Transformations and GALS-Verification

GTL models can be transformed and verified using a number of target languages and mechanisms. The simplest way to verify a GTL model is by including generated code from the synchronous components into a PROMELA model. This approach is described in Section 7.2.1. Abstraction by contract is a technique which can be used by translating the GTL model either to PROMELA (Section 7.2.2), to UPPAAL (Section 7.3) or an SMT instance (Section 7.6). The correctness of the specified contracts can be verified using the SCADE synchronous observer, as being described in Section 7.4. Strategies for implementing schedulers for the components are being discussed in Section 7.5.

7.1 Transformation of component contracts to automata

This transformation step is shared by every global verification technique. Because the contracts are mixtures of LTL-formulas and state machines, a unified representation is needed for the contracts before translating them into the target languages. Büchi automata [26] are a good way to achieve this, because there are standard algorithms for translating LTL formulas into Büchi automata [43, 42]. In addition, UPPAAL directly supports an automata representation as an input, and the translation of automata into PROMELA or SMT is straightforward.

Even though the expression language used for GTL contracts is more complex than pure LTL, note that all temporal operators can be translated into LTL-formulas since every time specification can be translated into a step-number by dividing with the cycle time. So if the cycle time of a component is

denoted by c, it follows that:

$$\text{next}[t]\ \varphi = \underbrace{\varphi \wedge \bigcirc(\varphi \wedge \dots)}_{\lfloor \frac{t}{c} \rfloor}$$

$$\text{after}[t]\ \varphi = \underbrace{\bigcirc(\bigcirc \dots (\bigcirc}_{\lfloor \frac{t}{c} \rfloor} \varphi))$$

$$\text{finally}[t]\ \varphi = \underbrace{\varphi \vee (\bigcirc(\varphi \vee \dots))}_{\lfloor \frac{t}{c} \rfloor}$$

$$\varphi\ \text{until}[t]\ \psi = \underbrace{\psi \vee (\varphi \wedge \bigcirc(\psi \vee (\varphi \wedge \dots)))}_{\lfloor \frac{t}{c} \rfloor}$$

Translation of LTL contracts. We adopt the algorithm from [42] that translates a given LTL formula into an equivalent Büchi automaton. Recall that a Büchi automaton is a tuple $(Q, \Sigma, \delta, q_0, F)$ with the following components:

- Q is the set of states of the automaton. It is usually represented by integer numbers in a practical implementation.
- Σ is the set of so called "atomic" expressions, that is expressions which do not contain logical connectors, but are only comprised of relations and arithmetic expressions. Using the grammar defined in Section 5.2, elements of Σ are represented by the non-terminal symbol `<atom>`.
- $\delta : Q \times 2^{\Sigma} \to 2^{Q}$ is the transition relation for the automaton. It takes a state and the set of currently valid atoms and produces a set of successor states.
- $q_0 \in Q$ is the initial state, in which the execution of a Büchi automaton starts.
- $F \subseteq Q$ is the set of final states, meaning states that have to be entered infinitely often for an accepting run.

To actually simulate a contract-abstraction in a target language, it is necessary to generate the set of variable mappings which make a given transition condition true. This is done by splitting the set of atoms Σ that must hold into two categories:

(1) Σ_I are atoms that perform checks on the input variables or on the local variables. These atoms must not contain any output variables or be an assignment to a local variable (cf. Sec. 5.5). Example: $i < 10$ for the input variable i.

(2) Σ_O are atoms which determine the values of output variables or are assignments of local variables. These atoms will either be some relation containing at least one output variable or an assignment to a local variable. Notice that the current restrictions of GTL (see Section 5.5) enforce that there is exactly one output variable. Example

$$o > i + 10,$$

where i is an input variable and o the output variable.

This means that Σ can be written as $\Sigma = \Sigma_I \cup \Sigma_O$. Our translated automaton will be a non-deterministic Mealy automaton with a Büchi acceptance condition, i. e., a tuple $(Q, \Sigma_I, \Sigma_O, \delta', q_0, F)$, where the transition function has the following type:

$$\delta' : Q \times 2^{\Sigma_I} \to 2^{Q \times 2^{\Sigma_O}}.$$

We call this type of automaton a *component automaton*. A transition $(q, X) \to (q, Y)$ in a component automaton means that whenever in the state q the atoms in the set $X \subseteq \Sigma_I$ are true the automaton proceeds to state q' and the atoms in $Y \subseteq \Sigma_O$ become true. The subset Y defines a system of (in)equations, which can be solved for the output variables. Each solution provides possible values of the output variables. In the target language one has to generate statements that non-deterministically assign each of these possible solutions to an output variable. In PROMELA, this can be realized by constructing a loop. For example, the following code assigns all values from 0 to 10 to the variable o:

```
1 o = 0;
2 do :: o < 10;
3       o = o + 1
4    :: break;
5 od
```

Translation of state machine contracts. Contracts given by a state machine in GTL are easily translated into component automata. The translation essentially only needs to transform Moore states, i. e., states containing atoms, to Mealy states. This transformation is straightforward.

Dealing with several contracts. Whenever the behavior of a component is specified by several contracts, one translates each of the contracts into an automaton and then forms a product automaton.

7.2 Transformation of Components to PROMELA processes

The transformation to PROMELA can be done in two ways:

(1) By translating the (concrete) SCADE models of the synchronous components into C and integrating them into PROMELA. This method is described in Section 7.2.1.

(2) By translating the component contracts into native PROMELA-code, described in Section 7.2.2.

7.2.1 Native Integration

A GTL-specification can be translated into PROMELA by translating every synchronous component into C. Note that this process is dependent on the synchronous formalism used to describe the component.

For example, SCADE-components are translated using the SCADE code-generator which is provided with the SCADE-suite. The translation process has to make sure that no naming-conflicts arise and is described in detail in [57]. The code-generator generates two data-structures for each component in the model. The first one is a structure of all input variables in the component and can be used to provide the component with the input data for each step. It has the name of the component, prefixed by "inC_". The second data-structure contains the internal state of the component as well as its output (because the output can potentially be used as state in SCADE). This data-structure is prefixed by "outC_". The code-generator also provides two generated functions for each component:

- A function to set the internal data-structure to its initialization-values, which has the name of the component suffixed by "_reset".
- A function to perform a calculation step of the component. This function takes both the input data-structure and the state data-structure and manipulates the internal state of the component. The name of the function is the name of the component.

The integration of the synchronous components in PROMELA is now done as follows: For each component, an instance of the state data-structure is added to the global state vector. This can be done by using the "c_state" construct in PROMELA. For a component named n, this can be done by using

```
1 c_state "outC_n n_state" "Global"
```

The state vector now has a variable named "n_state". The keyword "Global" declares the variable to be accessible from every process in the system (this is necessary so that other components can read the output).

The input data structure has no influence on the state of the model and can thus be declared to be outside of it. This is done by using the "c_decl" construct as follows:

```
1 c_decl {
2   inC_n n_input;
3 }
```

This creates a variable "n_input" which can be used to transfer inputs to the component.

At the start of the verification, the reset-function for each component has to be called, which can be done in the PROMELA "init" process. This process then proceeds to instantiate every component process. For a system of two components, named n_1 and n_2 this would generate the following code:

```
1  init {
2    c_code {
3      n1_reset(&now.n1_state);
4      n2_reset(&now.n2_state);
5    }
6    atomic {
7      run n1();
8      run n2();
9    }
10 }
```

In each step of a component, the inputs of the component are filled with the current outputs of the corresponding connected components. After this is done, the C step function has to be called.

Even though this translation process is straightforward, it has the problem that the outputs of each component are automatically contained in the state vector, even if they are not used by the component to determine its next step. This can result in a heavier memory consumption than actually required.

7.2.2 Contract abstraction

Component contracts are translated to PROMELA by translating the contract into a component automaton as described in Section 7.1. The resulting automaton is then transformed into a PROMELA process as follows:

Each state is translated into a labeled statement with the label corresponding to an element of Q. The statement is an "if"-construct that checks the input variables according to the function δ'. In each

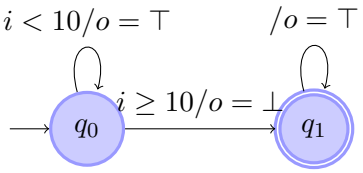

Figure 7.1: Example Büchi automaton

branch of the construct, the output variables are non-deterministically assigned and the new state is entered with a "goto"-statement. Care must be taken to insure that the initialization state q_0 is the first state generated so that the execution of the process starts in it.

As an example, consider the component automaton

$$(\{q_0, q_1\}, \delta, \{i < 10, i \geq 10\}, \{o = \top, o = \bot\}, q_0, \{q_1\})$$

with δ as displayed in Figure 7.1.

This automaton would be transformed into the following PROMELA process:

```
1  proctype component() {
2    q0: atomic {
3      if :: i < 10;
4            o = 1;
5            goto q0;
6         :: i >= 10;
7            o = 0;
8            goto q1;
9      fi
10   }
11   accept0: q1: atomic {
12     if :: o = 1;
13           goto q1;
14     fi
15   }
16 }
```

Note the use of the "atomic" construct to prevent SPIN from introducing unnecessary mid-steps. The Büchi-acceptance condition is enforced by using acceptance labels in the automaton (i.e. label "`accept0`").

7.2.3 Verification goals

Verification goals that are simply LTL formulas are automatically translated into PROMELA and can the be tested using the latest version of SPIN. Recall again, that this feature of SPIN cannot be used to generate the contract automata, because it generates an automaton which checks if the system conforms to the formula, not a process which generates the specified behavior.

Timed LTL formulas in verification goals semantically only make sense when components synchronize their execution with a global dense time as described in Sec. 7.5.4. In this case a timed LTL formula is translated by introducing clocks into the system. For each timed LTL subformula, a fresh clock is introduced which is used to check the time constraint of the formula. Clocks are numerical values c which decrease as time progresses. Each time a synchronous component (or rather its translated PROMELA process) makes a computation step, the timer c is decremented by the time that has

passed since the last step was performed by some (possibly different) component. A clock c can be set to a specific time t with the expression $c := t$. A clock's expiration status can be checked by using the clock name like a boolean expression. The expression is true if the clock is not yet expired. A timed until construct x **until**$[t]$ y can now be translated, using the fresh clock c, into

$$(c := t) \wedge ((x \wedge c) \textbf{ until } y)$$

A timed next construct **next**$[t]$ x is similarly translated to

$$(c := t) \wedge (x \textbf{ until} \neg c)$$

Derived constructs like finally are easily derived by the following equality:

$$\textbf{finally}[t]\ x \equiv \top \textbf{ until}[t]\ x$$

7.3 Transformation of Components to UPPAAL timed automata

Translating the resulting Büchi automaton from Section 7.1 to UPPAAL is easy, because UPPAAL already uses a state-based modeling language with transition guards. Notice that in UPPAAL the query language, in which verification goals are written, is somewhat restrictive as it provides only a fragment of timed CTL [8]. So some LTL formulas (e. g. $aU(bUc)$) express properties that cannot be expressed in the UPPAAL query language.

7.4 Translation of Component Contracts to Synchronous Observers

Synchronous observers are used in SCADE to verify a component (called "node" in SCADE). The synchronous observer watches the output (or part of it) of a component and produces a Boolean output which indicates whether or not the output is correct. The design verifier can then be used to verify that the produced variable is always true, no matter what inputs are supplied, which can then be used to deduce that the node's behavior is indeed correct.

The generated component automaton is translated into a SCADE state machine [10]. This state machine receives both the input and the output of the component and decides whether or not they conform to the behavior specified by the contract.

For example, the following observer node is generated for the expression "$\Box outp < 10$":

```
1  node Testnode (outp : int) returns (test_result : bool)
2  let
3    automaton
4      initial state init
5        unless
6          if outp < 10 restart init;
7          if true restart fail;
8        let
9          test_result = true;
10       tel
11     state fail
12       let
13         test_result = false;
14       tel
15   returns ..;
16 tel
```

Formally, a contract $\phi = \phi_1 \wedge \cdots \wedge \phi_n$ composed of n contract formulas is translated by translating each sub-formula ϕ_i into a SCADE automaton a_i. The generated node has both the vector i of input variables as well as the vector o of output variables of the component as its input. Local variables of the contract are translated into local variables l of the generated node.

Each automaton has a boolean variable "test_result_i" which indicates whether the current prefix path of input/output pairs still matches the behaviour described by the component. The result of the generated test-node is the conjugation of these variables.

```
1  node Testnode (i,o) returns (test_result : bool)
2  var
3    test_result_0 : bool;
4    ...
5    test_result_n : bool;
6    l
7  let
8    test_result = test_result_0 and ... and test_result_n;
9    a_0
10   ...
11   a_n
12 tel
```

The translation process of the sub-formula ϕ_i into the SCADE automaton a_i is done by first generating a buchi automaton a_i' from the sub-formula. This step has two different cases:

1. ϕ_i is an automaton. Because automatons in the GTL are already buchi automaton, we can simply use the formula as the buchi automaton ($a_i' = \phi_i$).

2. ϕ_i is a LTL formula. In this case the algorithm of Gastin and Oddoux [42] is used to create the buchi automaton a_i'.

Now the resulting buchi automaton a_i' is transformed by first checking if every state of it is final. This has to be done, because SCADE has no means to formulate liveness properties [1]. If all states are indeed final, the generated buchi automaton is determinized using well known algorithms [87] [2]. Now the variable "test_result_i", which indicates whether the current prefix path is valid, is assigned to be true in every state. A single new state is added to the automaton in which the variable "test_result_i" is false. A transition from every state is added to this new state with the lowest possible priority. This means that a transition into this failure state is only taken if there is no other transition available, which corresponds to not accepting the prefix path in the original buchi automaton.

7.5 Asynchronous Execution of processes

For the asynchronous execution of the abstract GALS model given by a GTL specification a *scheduler* has to be generated. In order to be able to generate different scheduling strategies our transformation generates a cycle counter for each translated component. The scheduling strategy should not depend on the absolute number of steps of each component but rather the relative difference between cycle numbers. This has the nice property that the cycle count can be normalized such that at least one

[1] It would be theoretically possible to use the "Liveness Checking as Safety Checking"-approach[22], but this would require an extensive analyzation and rewriting of the SCADE component. Also the paper suggests a 3-4 time increase in both verification time and required memory.

[2] Because every state in the generated automaton is final, we can also use a simpler powerset-based determinization algorithm.

cycle counter is zero after normalization. This reduces the state space as it makes cycles possible. A scheduling strategy can now be described as a function which takes a cycle counter for each process and decides if a given component may execute a cycle or not.

In the following subsections, a few possible scheduling strategies are being discussed.

7.5.1 Full asynchronicity

This scheduling strategy ignores the cycle counter and allows every component to execute a step every time. This is easy to implement (it doesn't even need the cycle counters to begin with) and if any property is proven to hold using this scheduling, it holds for any other. However, verification using this scheduling may produce errors that cannot occur in the real world, as it takes into account "unfair" runs in which one or more components make no step at all, while the others do.

7.5.2 Synchronicity with non-determinism

In this scheduling only the process with the lowest cycle count is allowed to proceed. This means that the cycle count for a process differs by at most one cycle from every other cycle counter. It is equivalent to a round robin scheduler.

7.5.3 Bounded asynchronicity

In this scheduling, components are allowed to have a certain execution time difference. If a process is farther away from the least executed process than a certain limit, it is no longer allowed to execute.

7.5.4 Synchronicity w.r.t. global dense time

In this, scheduling, components synchronize with a global dense time (see Section 5.4.4). The scheduling lets components make steps according to their cycle time. Here, in lieu of a cycle counter, for each component i, $1 \leq i \leq n$, an integer variable t_i is created, which holds the number of time units, the component has been executing. After a step of process j, the following updates are performed:

$$t'_j = t_j + c_j$$

where c_j is the cycle time of component j. For all i the following update is performed:

$$t'_i = t_i - min\{t_k \mid k \in \{0, \ldots, n\}\}$$

This ensures that always at least one counter is zero, and process i can make a step whenever t_i has value 0.

7.6 Transformation to an SMT instance

In every unrollment step i of the translation, each variable (input, output, local and automaton) v of component c has a representation as an SMT variable. This SMT variable is given by $\varphi(c, v, i)$. The *state vector* $s(c, i)$ of a component c at step i is the collection of all variables $\varphi(c, v, i)$ of the component. The contract of every component is translated into two predicates:

1. A predicate which determines the initial state of the component called I_c. Given a state $s(c, i)$ of the component, $I_c(s(c, i))$ states that it is an initial state of the component.

2. Another predicate which encodes the transition relation of the component, i.e. given a state of the component how the state in the next step looks like. This predicate is called T_c and $T_c(s(c,i), s(c,i+1))$ states that $s(c,i+1)$ is a successor state of $s(c,i)$.

For each connection statement "**connect** $c_1.v_1$ $c_2.v_2$;" in the GTL specification, in every step i, the following assertion is created:

$$\varphi(c_2, v_2, i+1) = \varphi(c_1, v_1, i)$$

This assertion states that the next input of component c_2 is determined by the current output of component c_1.

A *scheduling* provides the following things:

1. A vector $d(i)$ of SMT-variables for each unrollment step i which stores the current scheduling information.
2. A predicate $\alpha(d(i), c)$ which states whether the component c may perform a calculation step at the current scheduling state $d(i)$.
3. Another predicate $\beta(d(i), d(i+1), c)$ which schedules a step of the component c in the transition from state $d(i)$ to $d(i+1)$.
4. Also a predicate $\beta_0(d(0))$ which specifies the initial scheduling information.

We use $h(i,c)$ as the predicate which states that component c performs a step at i while all other components remain the same:

$$h(i,c) = T_c(s(c,i), s(c,i+1)) \wedge \bigwedge \left\{ s(c',i) = s(c',i+1) \mid c' \in \mathcal{C}, c \neq c' \right\} \wedge \beta(d(i), d(i+1), c)$$

The predicate $T(i)$ which specifies a step of the complete system can now be described:

$$T(i) = \bigvee \{ \alpha(d(i), c) \wedge h(i,c) \mid c \in \mathcal{C} \}$$

The initial state of the system is a conjunction of the initialization of each component and the initial scheduling data:

$$I = \bigwedge \{ I_c(s(c,0)) \mid c \in \mathcal{C} \} \wedge \beta_0(d(0))$$

A *path* of length k can thus be encoded as:

$$I \wedge T(0) \wedge T(1) \wedge \cdots \wedge T(k)$$

For notational ease, we introduce s_i as a vector of all variables in step i:

$$s_i = \{ s(c,i) \mid c \in \mathcal{C} \} \cup \{ d(i) \}$$

7.6.1 Verification goal encoding

The encoding of the LTL property to be verified is done by using the bounded LTL encoding [3] presented in [23]. The encoding assumes that the formula f to be verified is in positive normal form (negations only appear in front of atoms). The encoding is *incremental* which means that if the SMT solver supports the redaction of asserted formulas, we can re-use results obtained by checking for error-paths of length n for the checking of paths of length $n+1$. This means that asserted formulas for this encoding are split into three categories:

[3] The encoding presented in the paper uses an extension to classical LTL which allows formulas to speak about past events. This extension is called PLTL ("past LTL").

Base case assertions have to be made once before the first path is checked. They are valid for all paths.

k-invariant assertions are asserted for every step of the path, but they stay valid for every path.

k-variant assertions are only valid for paths of length k and have to be redacted if longer paths are considered.

As LTL can encode liveness properties, the encoding must be able to consider infinite paths. In the context of LTL verification, it is sufficient to assume that infinite paths are paths which form a loop at some point. To be able to distinguish between finite paths and paths with a loop, we introduce a few new variables:

- l_i encodes whether or not a loop starts at step i (meaning that the last state is equal to the i-th state).
- $InLoop_i$ specifies if the state i is part of the loop or not.
- $LoopExists$ is used to determine if a loop exists or if the path is finite.
- s_E is a *proxy* variable which is always equal to the last state in the path.

The assertions for these variables are as follows:

Base	$l_0 \Leftrightarrow \bot$
	$InLoop_0 \Leftrightarrow \bot$
k-invariant	$l_i \Rightarrow (s_{i-1} = s_E)$
$1 \leq i \leq k$	$InLoop_i \Leftrightarrow InLoop_{i-1} \vee l_i$
	$InLoop_{i-1} \Rightarrow \neg l_i$
k-variant	$l_{k+1} \Leftrightarrow \bot$
	$s_E = s_k$
	$LoopExists \Leftrightarrow InLoop_k$

For each sub-formula f' of f, two more proxy variables are created:

- $|[f']|_L$ is the value of f' at the point where the loop starts (or *false* if there is no loop).
- $|[f']|_E$ is the value of f' at the last step of the path.

They are defined by the following assertions:

Base	$\neg LoopExists \Rightarrow (\|[f']\|_L \Leftrightarrow \bot)$
k-invariant, $1 \leq i \leq k$	$l_i \Rightarrow (\|[f']\|_L \Leftrightarrow \|[f']\|_i)$
k-variant	$\|[f']\|_E \Leftrightarrow \|[f']\|_k$
	$\|[f']\|_{k+1} \Leftrightarrow \|[f']\|_L$

For sub-formulas which contain the "finally"-operator $\mathbf{F}$ or the "globally"-operator $\mathbf{G}$, a new encoding is introduced:

- $\langle\langle \mathbf{F} g \rangle\rangle_i$ states that g happened at least once in the loop at state i.
- $\langle\langle \mathbf{G} g \rangle\rangle_i$ states that g happened at all states of the loop up to state i.

The assertions for these variables enforce this:

	f'	
Base	$\mathbf{F}g$	$\langle\langle \mathbf{F}g \rangle\rangle_0 \Leftrightarrow \bot$
	$\mathbf{G}g$	$\langle\langle \mathbf{G}g \rangle\rangle_0 \Leftrightarrow \top$
k-invariant	$\mathbf{F}g$	$\langle\langle \mathbf{F}g \rangle\rangle_i \Leftrightarrow \langle\langle \mathbf{F}g \rangle\rangle_{i-1} \vee (\mathit{InLoop}_i \wedge \lvert[g]\rvert_i)$
$1 \leq i \leq k$	$\mathbf{G}g$	$\langle\langle \mathbf{G}g \rangle\rangle_i \Leftrightarrow \langle\langle \mathbf{G}g \rangle\rangle_{i-1} \wedge (\neg \mathit{InLoop}_i \vee \lvert[g]\rvert_i)$
k-variant	$\mathbf{F}g$	$\langle\langle \mathbf{F}g \rangle\rangle_E \Leftrightarrow \langle\langle \mathbf{F}g \rangle\rangle_k$
	$\mathbf{G}g$	$\langle\langle \mathbf{G}g \rangle\rangle_E \Leftrightarrow \langle\langle \mathbf{G}g \rangle\rangle_k$

With these auxiliary variables in place, we can give the encoding for $\lvert[f']\rvert_i$ which encodes the value of a sub-formula f' of f at position i in the path:

	f'	
Base	$\mathbf{F}g$	$\mathit{LoopExists} \Rightarrow (\lvert[\mathbf{F}g]\rvert_E \Rightarrow \langle\langle \mathbf{F}g \rangle\rangle_E)$
	$\mathbf{G}g$	$\mathit{LoopExists} \Rightarrow (\lvert[\mathbf{G}g]\rvert_E \Leftarrow \langle\langle \mathbf{G}g \rangle\rangle_E)$
	$g_1 \mathbf{U} g_2$	$\mathit{LoopExists} \Rightarrow (\lvert[g_1 \mathbf{U} g_2]\rvert_E \Rightarrow \langle\langle \mathbf{F}g_2 \rangle\rangle_E)$
	$g_1 \mathbf{R} g_2$	$\mathit{LoopExists} \Rightarrow (\lvert[g_1 \mathbf{R} g_2]\rvert_E \Leftarrow \langle\langle \mathbf{G}g_2 \rangle\rangle_E)$
k-invariant	$c.p$	$\lvert[c.p]\rvert_i \Leftrightarrow \varphi(c, p, i)$
$0 \leq i \leq k$	$\neg c.p$	$\lvert[\neg c.p]\rvert_i \Leftrightarrow \neg\varphi(c, p, i)$
	$g_1 \wedge g_2$	$\lvert[g_1 \wedge g_2]\rvert_i \Leftrightarrow \lvert[g_1]\rvert_i \wedge \lvert[g_2]\rvert_i$
	$g_1 \vee g_2$	$\lvert[g_1 \vee g_2]\rvert_i \Leftrightarrow \lvert[g_1]\rvert_i \vee \lvert[g_2]\rvert_i$
	$\mathbf{X}g$	$\lvert[\mathbf{X}g]\rvert_i \Leftrightarrow \lvert[g]\rvert_{i+1}$
	$\mathbf{F}g$	$\lvert[\mathbf{F}g]\rvert_i \Leftrightarrow \lvert[g]\rvert_i \vee \lvert[\mathbf{F}g]\rvert_{i+1}$
	$\mathbf{G}g$	$\lvert[\mathbf{G}g]\rvert_i \Leftrightarrow \lvert[g]\rvert_i \wedge \lvert[\mathbf{G}g]\rvert_{i+1}$
	$g_1 \mathbf{U} g_2$	$\lvert[g_1 \mathbf{U} g_2]\rvert_i \Leftrightarrow \lvert[g_2]\rvert_i \vee (\lvert[g_1]\rvert_i \wedge \lvert[g_1 \mathbf{U} g_2]\rvert_{i+1})$
	$g_1 \mathbf{R} g_2$	$\lvert[g_1 \mathbf{R} g_2]\rvert_i \Leftrightarrow \lvert[g_2]\rvert_i \wedge (\lvert[g_1]\rvert_i \vee \lvert[g_1 \mathbf{R} g_2]\rvert_{i+1})$

7.6.2 Encoding of timed properties

The GTL allows the specification of timed verification properties, such as "x $\mathbf{until}[12s]$ y" which means that x holds until y holds once after at least 12 seconds. To realize such operators, we extend the LTL with two unary operators, $\triangleright_{\sim t}$ and $\triangleleft_{\sim t}$ (with $\sim \in \{<, >, \leq, \geq, =\}$). Given a finite suffix π^i of π starting from the ith state, where the jth step takes $\rho(j)$ time units, the semantics of these operators is defined as follows:

$$\pi^i \models \triangleright_{\sim t}\psi \Leftrightarrow \exists n \geq i \text{ such that } \pi^n \models \psi \text{ and } \textstyle\sum_{j=i}^{n} \rho(j) \sim t$$
$$\pi^i \models \triangleleft_{\sim t}\psi \Leftrightarrow \exists 0 \leq n \leq i \text{ such that } \pi^n \models \psi \text{ and } \textstyle\sum_{j=n}^{i} \rho(j) \sim t$$

With these operators defined, we derive timed variants of the operators $\mathbf{U}$, $\mathbf{R}$, $\mathbf{F}$, and $\mathbf{G}$:

$$\begin{array}{lcl}
\psi_1 \mathbf{U}[t] \psi_2 & \Leftrightarrow & \psi_1 \mathbf{U} \psi_2 \wedge \triangleright_{\leq t} \psi_2 \\
\psi_1 \mathbf{R}[t] \psi_2 & \Leftrightarrow & \psi_1 \mathbf{R} \psi_2 \vee \triangleright_{> t} \neg \psi_2 \\
\mathbf{F}[t] \psi & \Leftrightarrow & \triangleright_{\leq t} \psi \\
\mathbf{G}[t] \psi & \Leftrightarrow & \mathbf{G} \psi \vee \triangleright_{> t} \neg \psi
\end{array}$$

We extend the LTL encoding with two new variable types:

- $\overline{x}(\psi)_i$ states that there is a future state ($\geq i$) which fullfils ψ.

- If $\overline{x}(\psi)_i$ is true then $x(\psi)_i$ encodes the time until the state in which ψ holds is reached.

Since we are using an incremental encoding, the value of $\overline{x}(\psi)_i$ may change from false to true when increasing the k-bound. This can happen because ψ may hold at a state that is outside of the current k-bound. Table 7.1 shows an example of this behaviour: In the $k = 3$ case, $\overline{x}(\varphi)_1$ is false, because no state $\geq i$ within the k-bound satisfies φ. But in the $k = 4$ case, φ is true in the 4th state, which makes $\overline{x}(\varphi)_1$ true. This behaviour is an under-approximation of the LTL formula, since it is guaranteed to be in positive normal form. It is easy to see that $\triangleright_{\sim t}\psi$ is true exactly if ψ happens in the future and the

i	0	1	2	3	4
$\|[\varphi]\|_i$	$\top$	$\bot$	$\bot$	$\bot$	
$x(\varphi)_i$	0	*	*	*	
$\overline{x}(\varphi)_i$	$\top$	$\bot$	$\bot$	$\bot$	
$\rho(i)$	1	2	1	1	
$\|[\triangleright_{<2}\varphi]\|_i$	$\top$	$\bot$	$\bot$	$\bot$	

$\rightsquigarrow$

i	0	1	2	3	4
$\|[\varphi]\|_i$	$\top$	$\bot$	$\bot$	$\bot$	$\top$
$x(\varphi)_i$	0	4	2	1	0
$\overline{x}(\varphi)_i$	$\top$	$\top$	$\top$	$\top$	$\top$
$\rho(i)$	1	2	1	1	2
$\|[\triangleright_{<2}\varphi]\|_i$	$\top$	$\bot$	$\bot$	$\top$	$\top$

Table 7.1: The values of encoding variables for $k = 3$ and $k = 4$

time of its occurance satisfies the constraint $\sim t$. Thus, we can replace it with $\overline{x}(\psi) \wedge x(\psi) \sim t$.

The complete encoding for these variables is as follows:

k-invariant	$\|[\triangleright_{\sim t}\psi]\|_i \Leftrightarrow (x(\psi)_i \sim t) \wedge \overline{x}(\psi)_i$
$0 \leq i \leq k$	$\|[\psi]\|_i \Rightarrow (x(\psi)_i = 0) \wedge \overline{x}(\psi)_i$
	$\neg\|[\psi]\|_i \Rightarrow (x(\psi)_i = x(\psi)_{i+1} + \rho(i)) \wedge (\overline{x}(\psi)_i = \overline{x}(\psi)_{i+1})$
k-variant	$\neg\overline{x}(\psi)_{k+1}$

7.7 LLVM verification of contracts

Instead of just allowing component models to be described in SCADE, we also developed a local verification for component models written in a LLVM-supported language. With it, users of the GTL-tool can either write component models in C or C++, or use the SCADE code generator (called "KCG") to transform a SCADE model into C. A C-model must be in a specific form in order to be verifiable against a GTL-contract [4]:

- A single data structure which holds the state information and the output variables of the component.
- An initialization function which takes a pointer to the state data structure and returns nothing (*void*). It resets both the internal state and the output variables of the component to its initial state. This is required because the values that a freshly allocated data structure can have are often random or zero, while many models require initialization values which aren't zero.
- A function which performs a calculation step in the model. This function takes as argument all the input variables of the model and a pointer to the state data structure. It reads the current state of the model from that pointer and updates the state and output variables according to the input variables and the current state. Like the initialization function, it returns no value.

[4] To allow for an easy verification of translated SCADE models, the format generated by the SCADE KCG conforms to this format.

The following code is an example of a C model which simulates a modulo counter. The model has one input, “tick”, which indicates whether or not the counter value shall be increased. If it is true, the value of the counter is increased and compared to a “modulo value”. In the case that it is equal to this value, the counter is reset to zero and a “tick” is produced.

```
#include <stdbool.h>

typedef struct {
  int cur_val;
  int mod_val;
  bool tick_out;
} counter;

void init(counter* c) {
  c->cur_val = 0;
  c->mod_val = 5;
  c->tick = false;
}

void step(bool tick_in,counter* c) {
  if(tick_in) {
    c->cur_val++;
    if(c->cur_val == c->mod_val) {
      c->tick_out = true;
      c->cur_val = 0;
    } else {
      c->tick_out = false;
    }
  } else {
    c->tick_out = false;
  }
}
```

This example would be included into a GTL-specification as follows:

```
model[c] counter("counter.c","counter","init","step") {
  ...
}
```

The LLVM code checker works by checking user-defined assertions. Since the code checker has yet to be extended with the possibility to define liveness properties, we can only verify (as in the SCADE backend) contracts which represent safety properties of the model. The user can generate non-deterministic values by using the function “nondet_t”, where t is the type of the non-deterministic value.

For example, the contract **always** $cur\_val < 5$ would be verified by checking the following code for assertion violations:

```
int main(int argc,const char* argv[]) {
  counter c;
  bool tick_in;
  init(&c);
  while(1) {
    assert(c.cur_val < 5);
    tick_in = nondet_bool();
    step(tick_in,&c);
  }
```

```
10     return 0;
11 }
```

Chapter 8

Verification of GALS models

In this section we describe how the GTL language and the model transformations described in the previous subsections (also cf. Section 1.2 and Fig. 1.1) are used to verify a GALS system.

8.1 Abstraction by contracts

We begin with a minimalistic example that demonstrates how contracts are used to abstract the real components. Our system consists of two components: a source component and a sink component. The source component is a simple $\mathrm{mod}\ 10$ counter and produces the following sequence of numbers at its only output:

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, \ldots$$

The output outp of the source component is connected with the only input inp of the sink component. The sink simply checks whether its input flow is less than 12 and sets the output to 0 in that case and to 1 otherwise. Fig. 8.1 shows the two corresponding SCADE models.

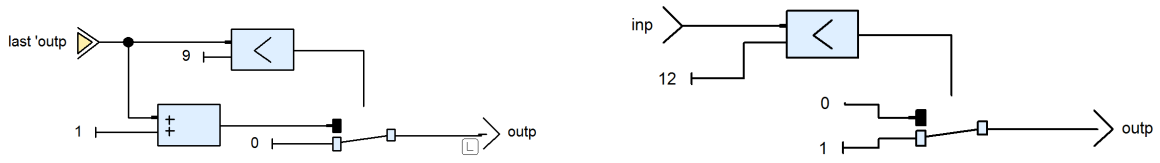

Figure 8.1: SCADE models of two simple components

Let us assume we want to verify that the output value of the sink is always 0. As contracts for the components we specify that the source always produces values less then 10. And the sink outputs 0 if its input value is less then 10. This yields the following GTL specification:

```
1  model[scade] source("source_sink.scade","Source") {
2    contract always outp < 10;
3  }
4
5  model[scade] sink("source_sink.scade","Sink") {
6    init outp 0;
7    init inp 9;
8    contract always (inp < 10 => outp=0);
9  }
10
```

```
11 connect source.outp sink.inp;
12
13 verify {
14   always sink.outp=0;
15 }
```

Listing 8.1: source and sink

Clearly, the above contracts hold for the two SCADE models from Fig. 8.1. Of course, this can easily be checked using the SCADE DV.

In addition, the contracts clearly specify an abstraction of the concrete SCADE components. In fact, there are many different SCADE implementations of the source and sink components that will satisfy those contracts.

The meaning of the contract for the source is that the source component will non-deterministically exhibit every behavior admitted by the contract; in this case its output has any value less then 10 non-deterministically. So contracts abstract components by under-specification.

For our simple example, Fig. 8.2 shows the Büchi automata generated from the GTL specification

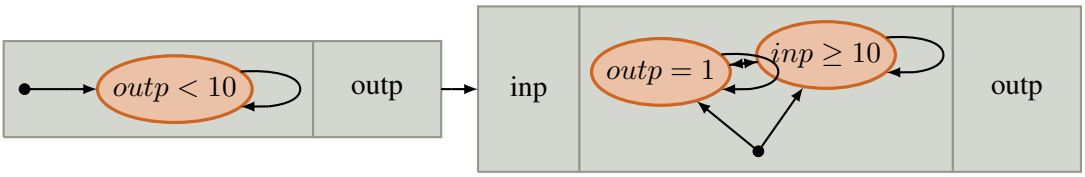

Figure 8.2: Source-Sink contract automata

In general, because of this underspecification it follows that for a verification goal specifying a safety property we have the following:

If the abstract model satisfies the verification goal, then so does the concrete model.

Indeed, a safety property states that certain states of a system are never reached. So if the abstract model (having more possible behavior than the concrete one) satisfies the property so does the concrete model.

It is well-known (see e. g. [18]) that such a statement is false for liveness properties. The reason is that our abstract model exhibits more behavior than the concrete model and so properties stating that something will eventually happen might be true in an abstract model while false in the concrete one.

8.2 Verification steps

The transformations are being implemented prototypically within the GTL tool. Using this tool the verification of a GALS system is performed as shown in Fig. 8.3. Here we explain the workflow using PROMELA as back end language for GALS models and SPIN as analysis tool. But, of course, a very similar workflow is performed when we use UPPAAL timed automata instead.

We now describe the involved steps more detailed:

1. Suppose the SCADE models of components and abstract GALS model as a GTL-specification are given. The contracts for each component are translated with the GTL tool to SCADE synchronous observers (cf. Section 7.4) that are used by the SCADE DV to establish that the components satisfy their contracts.

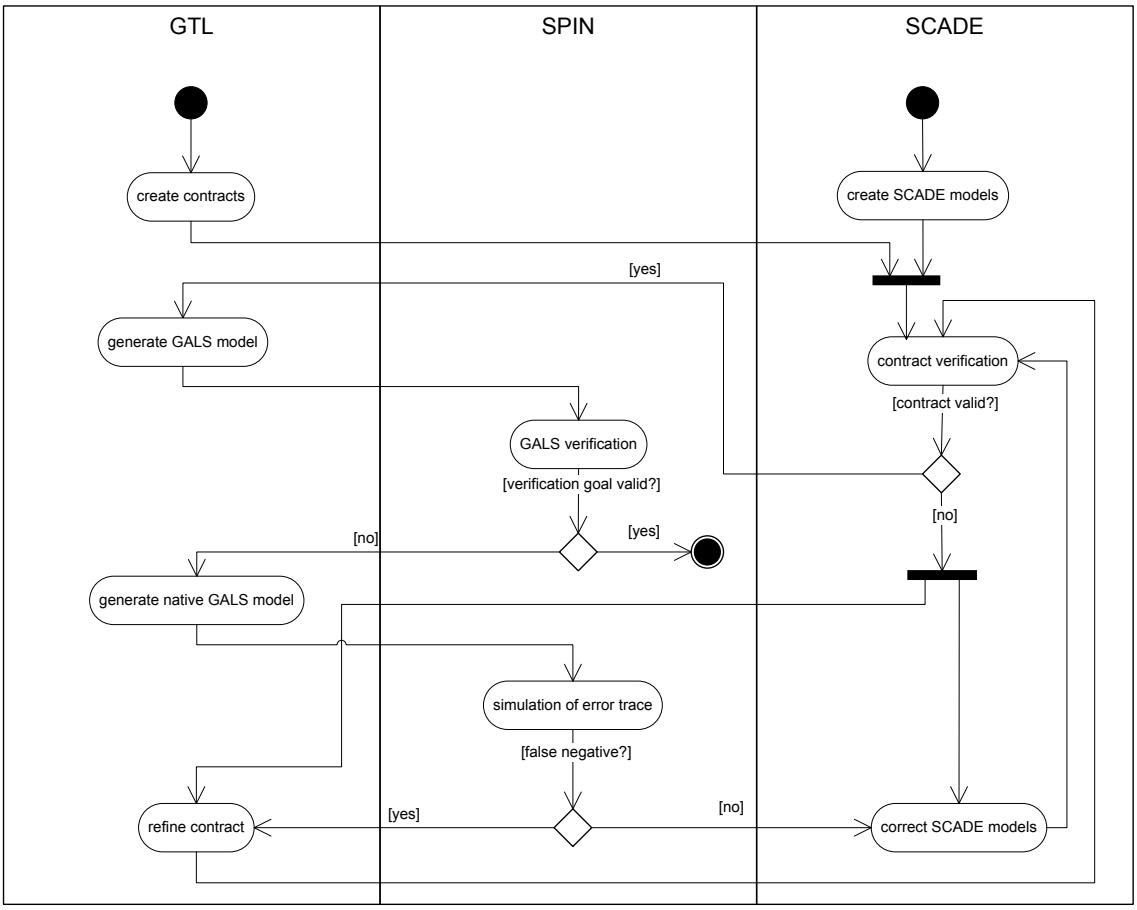

Figure 8.3: Verification of GALS systems

Notice that guaranteed behaviors are not translated to SCADE observers in this step because they represent component behavior that is known to hold for the components and has already been verified previously.

2. If a component does not satisfy its contract this can have two reasons: (a) there is an error in the corresponding SCADE model, and after this error has been corrected one goes back to step 1; (b) the contract formulates a requirement that is too strong, and, hence, the contract must be weakened before step 1. is repeated.

3. Otherwise, all components satisfy their contracts, and one now uses the GTL tool to generate from the given abstract GALS model in GTL a representation in PROMELA.

4. The analysis of the verification goals on the PROMELAGALS model is performed with SPIN. If the verification is successful, we are done. Otherwise, some verification goal does not hold for the GALS model of contracts, and SPIN produces a corresponding error trace.

5. With the help of the GTL tool one now generates the concrete GALS models in PROMELA. This

means that the GTL tool automatically integrates the C code generated from the SCADE models of the components is integrated into a GALS system model in PROMELA.

6. The resulting concrete GALS model is simulated with the error trace from step 4., and it is checked whether this system does indeed exhibit the faulty behavior from the error trace.

7. If this is the case, we have found a real error in the GALS model, and the SCADE models of the components do not correctly compose to a system. The error has to be analyzed further, which should eventually result in one or more SCADE models of system components to be adjusted or corrected. This debugging process can be supported by the GTL tool in two ways: from the given error trace one can generate

   (a) a global *schedule* which contains the inputs and outputs for the system and its components plus the information at what time each component makes steps until the state violating the verification goal is found, and

   (b) for each component a SCADE simulator script containing the inputs for as many cycles of the component as occurring in the global schedule from (a).

   When all necessary corrections of SCADE models are completed one repeats the verification from step 1.

8. Otherwise, the concrete GALS model does not exhibit the faulty behavior from the given error trace, and we have thus a "false negative" verification result. This means that one or several component contracts are too weak to force the desired verification goal to hold. These contracts need to be refined, and then one repeats the verification from step 1.

8.3 Refinement of contracts

As we saw previously, a failed verification can imply a "false negative", which means that certain contracts in an abstract GALS model are not strong enough and need to be refined. We envision two methods how contract refinement can be achieved:

(1) One uses guaranteed behavior for refinement. Indeed, recall that guaranteed behaviors are certain contracts that are known to hold for their corresponding component. For example, these could be LTL formulations of requirements that have been verified for a component during some previous module or component test. Typically the guaranteed behaviors will *not* be used when a GTL specification is translated to its corresponding abstract GALS model as they might blow up the complexity of this model. But in the case of a "false negative" one can consider guaranteed behaviors as real contracts and take them into account when generating the abstract GALS model from the GTL specification. In this way contracts are refined, and in a subsequent verification step (see step 4. in Section 8.2) one hopes to verify that the verification goal in question holds for the GALS model.

   Notice that this kind of refinement can be performed automatically.

(2) The second possibility for refinement of component contracts is to use the error trace produced together with a negative outcome of a verification in step 4. from Section 8.2. This idea is inspired by counter example guided abstraction refinement [34].

In the case of a "false negative" this error trace describes a run of the system where the concrete model behaves correctly. Hence, from the error trace yields a system test case with concrete input values that the concrete GALS model passes. Now, the idea is to use this passed system test to refine the abstract GALS model by new contracts with the goal to be able to verify the original verification goal.

However, it is *not* sufficient to just observe a component's in- and outputs during the run of the passed system test and translate this to an equivalent LTL formulation to refine the component's contract. This will refine the contract but only in such a way that re-verification of the failed verification goal will produce a longer counterexample.

Instead, one must use the information from the passed system test to formulate a new contract that further constrains the behavior of the specified component. This can be done, for example, by extracting an invariant $\mathsf{G}\varphi$, where φ is a propositional formula, a (bounded) response property $\mathsf{G}(\varphi \implies \mathsf{F}\psi)$, where φ, ψ are propositional formulas or an equivalent formulation using automata in GTL, respectively.

Notice that this technique requires creative human intervention to formulate a new contract for refinement of components. This can be supported by the automatic generation of SCADE simulator scripts from the given passed system test (represented by the given error trace from the previous model checking step) as mentioned in step 7(b) above.

Chapter 9

Bounded Model Checking and Model-Based Testing

Note. The results presented in this chapter have been elaborated in cooperation with the EU FB7 research project COMPASS which is funded under grant agreement no.287829.

9.1 Model-Based Testing

In the VerSyKo project and in the COMPASS project, model-based testing support is provided by the RT-Tester tool with its MBT component RTT-MBT. RT-Tester is an industrial strength tool for model-based testing of reactive concurrent real-time systems [82, 97].

Following the definition currently given in Wikipedia[1]

> *Model-based testing (MBT) is the application of Model based design for designing and optimally executing the necessary artefacts to perform software testing. Models can be used to represent the desired behaviour of the System Under Test (SUT), or to represent the desired testing strategies and testing environment.*

In this definition only software testing is referenced, but it applies to hardware/software integration and system testing just as well. Observe that this definition does not require that certain aspects of testing – such as test case identification or test procedure creation – should be performed in an automated way: the MBT approach can also be applied manually, just as design support for testing environments, test cases and so on. This rather unrestricted view on MBT is consistent with the one expressed in [12].

Automated MBT has received much attention in recent years, both in academia and in industry. This interest has been stimulated by the success of model-driven development in general, by the improved understanding of testing and formal verification as complementary activities, and by the availability of efficient tool support. Indeed, when compared to conventional testing approaches, MBT has proven to increase both quality and efficiency of test campaigns; we name [**?**] as one example where quantitative evaluation results have been given. In this report the term model-based testing is used in the following, most comprehensive, sense: the behaviour of the *system under test (SUT)* is specified by a model elaborated in the same style as a model serving for development purposes. Optionally, the SUT model can be paired with an environment model restricting the possible

[1] `http://en.wikipedia.org/wiki/Model-based_testing`, (date: 2012-06-14).

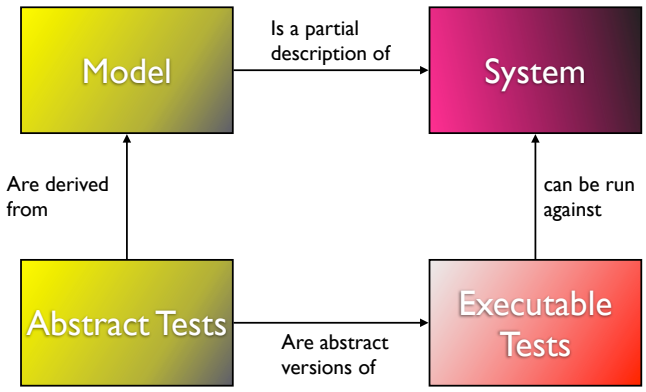

Figure 9.1: The model-based testing paradigm

interactions of the environment with the SUT. A *symbolic test case generator* analyses the model and specifies *symbolic test cases* as logical formulas identifying model computations suitable for a certain test purpose. Constrained by the transition relations of SUT and environment model, a *solver* computes concrete model computations which are *witnesses* of the symbolic test cases. The inputs to the SUT obtained from these computations are used in the test execution to stimulate the SUT. The SUT behaviour observed during the test execution is compared against the *expected* SUT behaviour specified in the original model. Both stimulation sequences and *test oracles*, i. e., checkers of SUT behaviour, are automatically transformed into *test procedures* executing the concrete test cases in a model-in-the-loop, software-in-the-loop, or hardware-in-the-loop configuration.

Observe that this notion of MBT differs from "weaker" ones where MBT is just associated with some technique of graphical test case descriptions. According to the MBT paradigm described here, the focus of test engineers is shifted from test data elaboration and test procedure programming to modelling. The effort invested into specifying the SUT model results in a return of investment, because test procedures are generated automatically and debugging deviations of observed against expected behaviour is considerably facilitated because the observed test executions can be "replayed" against the model. Moreover, V&V processes and certification are facilitated because test cases can be automatically traced against the model which in turn reflects the complete set of system requirements.

In Fig. 9.1 the MBT paradigm is sketched as described in Wikipedia as referenced above. There the term *abstract tests* is used for symbolic test cases, and *executable tests* for test procedures.

9.2 Development Models Versus Test Models

The reference model used for a MBT campaign may

- coincide with the model used to generate the SUT code according to the model-driven development approach, or
- consist of a separate model developed by the V&V team,

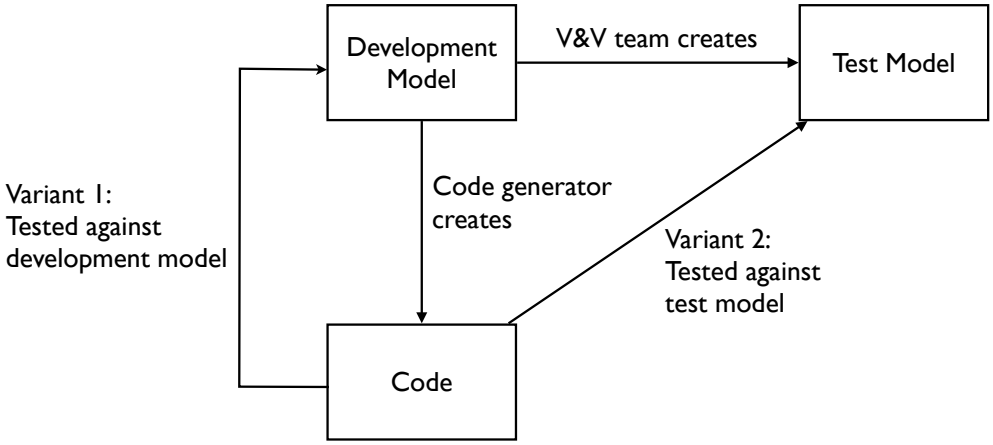

Figure 9.2: Development model versus test model as source for MBT.

as depicted in Fig. 9.2. In the former case tests are derived from the development model, so there are no possibilities to uncover *logical, functional* errors during testing, because the tests just ensure that the SUT behaves in a way which is consistent with the development model. This MBT variant is useful if the development model has been exhaustively verified with respect to correctness and validated with respect to completeness. The objective of the test suite is then only to verify whether the generated code is a correct refinement of the development model.

The latter case is to be applied in situations where

- the development model cannot be trusted with respect to correctness and completeness, or
- the level of abstraction of the test model is unsuitable for the test objectives.

In automated model-driven development models often show a greater level of detail than needed for testing: development models need to capture internal task structures, communication channels and event handlers which are not relevant, for example, when designing a black-box testing campaign only monitoring or stimulating the hardware interfaces of the SUT. In these cases the V&V team creates separate test models from the development models and from their own interpretation of the requirements. As a consequence, the test model may be at a differentlevel of abstraction than the development model and describe a SUT behaviour that deviates from the one captured in the development model. The test suite may detect both deviations between code and development model and logical, functional errors in the development model.

9.3 Basic MBT Automation Techniques

The RT-Tester architecture supporting automated MBT is depicted in Fig. 9.3. Test models are parsed in textual format (typically XMI) by the parser front end, and the model is internally represented by an abstract syntax tree (AST), called the RT-Tester *internal model representation (IMR)*. The *test case*

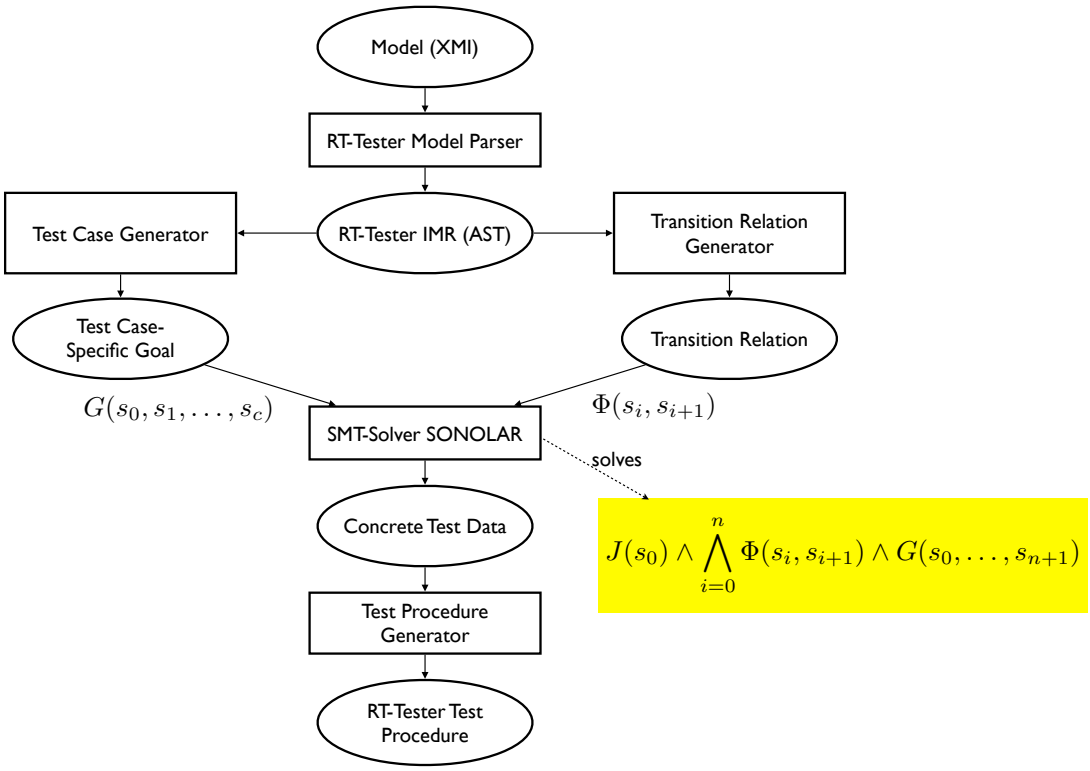

Figure 9.3: Test procedure generation using test case generator and SMT solver.

generator traverses the AST and identifies relevant test cases based on the syntactic model representation. Test cases are represented in *symbolic* form, that is, as logical formulas $G(s_0, s_1, \ldots, s_c)$ over consecutive state valuations $s_0, s_1, \ldots, s_c$: any test objective can be encoded as a formula specifying the characteristics of a finite sequence of states (also called a *trace*) to be traversed for meeting this objective.

The *transition relation generator* traverses the AST with the objective to encode the model's operational semantics as a transition relation $\Phi(s_i, s_{i+1})$ relating states s_i to their post-states s_{i+1}. Mixed traces of event occurrences and state changes can also be internally encoded as formulas over states s_i, s_{i+1}.

Concrete test data is created by solving constraints of the type

$$J(s_0) \wedge \bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \wedge G(s_0, \ldots, s_{n+1})$$

using the integrated SMT solver SONOLAR [79]. In such a formula, conjunct $J(s_0)$ characterises the current model state from where the next test objective $G(s_0, \ldots, s_{n+1})$ should be covered. Conjunct $\bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1})$ ensures that the solution of $G(s_0, \ldots, s_{n+1})$ results in a valid trace of the model, starting from s_0.

Finally the *test procedure generator* takes the solutions calculated by the SMT solver and turns

them into stimulation sequences, that is, timed input traces to the SUT. Moreover, the test procedure generator creates test oracles from the model components describing the SUT behaviour.

Formulas of the type displayed above are called *bounded model checking instances (BMC instances)*[2]: in BMC models are verified in the vicinity of states s_0, whether an undesirable property $G(s_0, \ldots, s_{n+1})$ – for example, a safety violation – can be realised within n transition steps from s_0. The difference between test generation and model checking lies in the expectation whether a solution for $G(s_0, \ldots, s_{n+1})$ can be found: for a reasonably defined symbolic test case many solutions of the formula should exist, while solutions of the formula during model checking always uncover a model error. Since the SMT solver is able to find solutions for BMC instances, we can exploit this both for test generation and for local verification of test models.

9.4 Model-Based Test Case Generation

Test cases can be derived from the model in an automated way. The strategies needed for this purpose are described in this chapter. Moreover, test experts may define their own "special-purpose test cases"; this is explained in Section 9.7.

9.5 Computations, Traces and Model Coverage

In our context of model-based testing, the behavioural semantics of test models is expressed by the set of its *computations*, that is, its infinite sequences $c = s_0.s_1.s_2 \ldots$ of state changes that may be executed by the model in accordance with the rules of its operational semantics. Note that events and time are also encoded in the states s_i, so that each state is a complete snapshot of internal model state and interface activities.

A *trace* is a finite prefix of a valid model computation. When testing, only traces of the model can be stimulated and observed. An *input trace* is a finite sequence of inputs passed to the SUT model portion over its input interfaces. Interfaces are realised by property values travelling over ports along item flows, and by signal events.

We say that a trace $\pi = s_0 \ldots s_n$ *covers* a structural model element (flow ports, blocks) if the state sequence triggers state changes or behaviours of the element under consideration. Examples for traces covering structure are

- state sequences where the properties associated with a flow port change their value ("the trace covers the port"),
- state sequence leading to state machine transitions in a machine associated with a block ("the trace covers the block and its higher-level blocks").

Behaviour is expressed by operations and state machines in the model. Trace π is said to *cover a behaviour* of the model if it triggers the operations and (potentially concurrent) state machine transitions realising this behaviour.

[2]The notion of BMC instances is inspired by the term *SAT instance* used for Boolean formulas whose solvability is to be checked by SAT solvers.

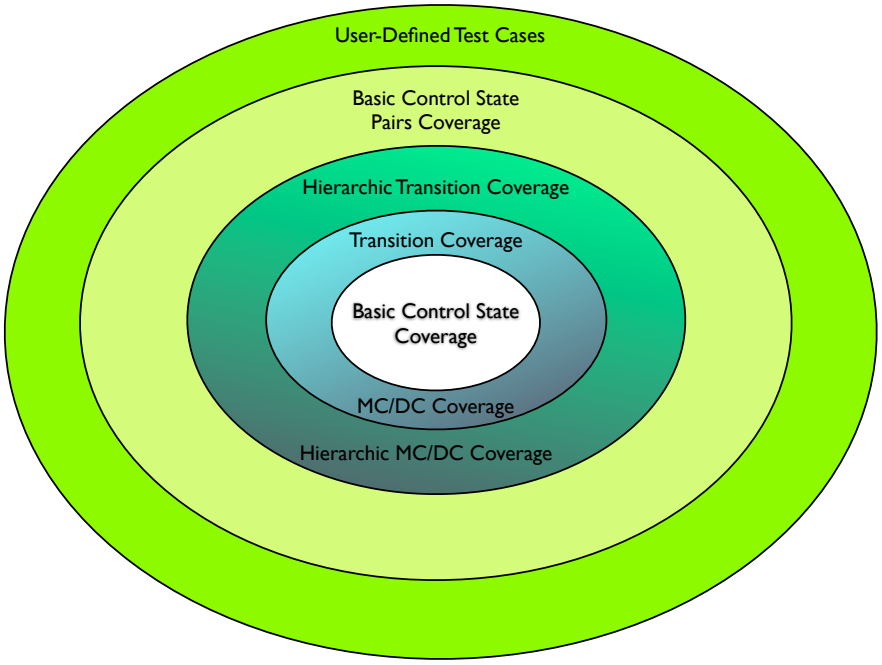

Figure 9.4: Behavioural model coverage strategies for MBT, I.

9.6 Model Coverage Strategies

Model-derived test strategies aim at covering certain parts of the model structure and (a subset of) its behaviours. The basic model coverage strategies are well-defined and well-explored, see, for example [98] for a comprehensive discussion. For concurrent real-time systems, however, and for large-scale concurrent systems, test strategies still require active research. In Fig. 9.4, 9.5, 9.6 an overview of these strategies is given, and we will discuss them in the paragraphs below.

BCS – Basic Control State Coverage. This type of behavioural coverage aims at covering each basic control state of each state machine at least once. No additional objectives are made about concurrent control states or accompanying variable valuations when reaching the control state under consideration. Using the general test case formula pattern

$$J(s_0) \wedge \bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \wedge G(s_0, \ldots, s_{n+1})$$

introduced in Section 9.3, BCS leads to formulas of the type

$$J(s_0) \wedge \bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \wedge G(s_{n+1})$$

with

$$G(s_{n+1}) \equiv s_{n+1}(\ell)$$

where ℓ is the name of the BCS to be covered. Control states are interpreted as Boolean variables, so $s_{n+1}(\ell)$ evaluating to `true` means that the corresponding state machine resides in location ℓ in the

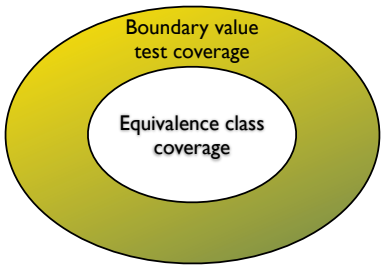

Figure 9.5: Behavioural model coverage strategies for MBT, II

state reached after the $(n+1)^{th}$ transition step. Using linear temporal logic [35] LTL, the goal can be expressed by

$$\mathbf{F}\ \ell$$

Observe that the goal shall be reached in state s_{n+1}. This means that at least one transition step will be performed: if basic control state ℓ is already active in s_0, that is, if $G(s_0)$ holds, we do not need to solve a BMC instance, but can directly mark G as covered. This observation applies throughout the paragraphs below where different types of goal will be introduced.

TR – Transition Coverage. Transition coverage aims at covering each transition τ of every state machine in the model. Again, no restrictions are made regarding variable valuations, control states and concurrent transitions to be performed when the one under consideration is triggered. The associated goal can be expressed as

$$G(s_{n+1}) \equiv s_{n+1}(\text{trigger}(\tau)) \ \text{ or in LTL as } \ \mathbf{F}\ \text{trigger}(\tau)$$

Intuitively speaking, trigger(τ) specifies that the required event triggering τ is available in the event pool, the guard condition evaluates to `true`, and no higher-priority transition is enabled in model state s_{n+1}.

MCDC – MC/DC Transition Coverage. *Modified condition/decision (MC/DC) coverage* is a variant of transition coverage, where non-atomic guard conditions are evaluated in a systematic manner.

- If a transition guard has the structure $a \wedge b$, then the transition should be tested several times, so that at least the condition valuations
$$a \wedge b$$
$$\neg a \wedge b$$
$$a \wedge \neg b$$
are covered.

- If a transition guard has the structure $a \vee b$, then the transition should be tested several times, so that at least the condition valuations
$$\neg a \wedge \neg b$$
$$\neg a \wedge b$$
$$a \wedge \neg b$$
are covered.

For the general case, we express guard conditions g in conjunctive normal form of atomic propositions g_i^j

$$g \equiv \bigwedge_{i=0}^{u} \bigvee_{j=0}^{k_i} g_i^j$$

(this transformation is performed automatically by RTT-MBT) and define

- *Stability test goals.* A minimal number of terms $\bigvee_{j=0}^{k_i} g_i^j$ – if possible for exactly one $i \in 0, \ldots, u$ – evaluate to `false`, all others evaluate to `true`. The stability test is performed for as many i as possible.

- *Progress test goals.* All conjuncts $\bigvee_{j=0}^{k_i} g_i^j$ evaluate to `true`, but in each conjunct only a minimal number of g_i^j – preferably just for one j – evaluate to `true`. The progress tests are performed for as many different combinations of g_i^j evaluating `true` as possible.

As test goals, stability tests can be expressed as

$$G_i(s_{n+1}) \equiv s_{n+1} \left(e \wedge \neg(\bigvee_{j=0}^{k_i} g_i^j) \wedge \sum_{h=0}^{u} (\bigvee_{j=0}^{k_h} g_h^j) = u \right), \quad i = 0, \ldots, u$$

or

$$\mathbf{F} \left(e \wedge \neg(\bigvee_{j=0}^{k_i} g_i^j) \wedge \sum_{h=0}^{u} (\bigvee_{j=0}^{k_h} g_h^j) = u \right)$$

In these definitions e denotes the event triggering τ. Identifying Boolean values `true`, `false` with $1, 0$, respectively, the term $\sum_{h=0}^{u} (\bigvee_{j=0}^{k_h} g_h^j)$ evaluates to the number of conjuncts evaluating to `true` in state s_{n+1}, so $\sum_{h=0}^{u} (\bigvee_{j=0}^{k_h} g_h^j) = u$ specifies that all but one conjuncts evaluate to `true`. If there is no solution for this constraint, one can try to solve $(\sum_{h=0}^{u} \bigvee_{j=0}^{k_h} g_h^j) \geq p$ for some suitable $p < u$.

For progress tests, the goals look like

$$G_{m_1 \ldots m_u}(s_{n+1}) \equiv s_{n+1} \left(\text{trigger}(\tau) \wedge \bigwedge_{i=0}^{u} \left(g_i^{m_i} \wedge \sum_{j=0}^{k_i} g_i^j = 1 \right) \right)$$

or, in LTL,

$$\mathbf{F} \left(\text{trigger}(\tau) \wedge \bigwedge_{i=0}^{u} \left(g_i^{m_i} \wedge \sum_{j=0}^{k_i} g_i^j = 1 \right) \right)$$

conjunct trigger(τ) states that the transition will be triggered. Every conjunct of the guard condition has exactly one disjunct $g_i^{m_i}$ which evaluates to `true`, all other ones evaluate to `false`; this is expressed by the constraint $\sum_{j=0}^{k_i} g_i^j = 1$. If possible, test cases should be generated for every combination of $(m_1, \ldots, m_u) \in \{1, \ldots, k_1\} \times \ldots \times \{1, \ldots, k_u\}$. If no solution exists fulfilling $\sum_{j=0}^{k_i} g_i^j = 1$, then weaker constraints $\sum_{j=0}^{k_i} g_i^j \leq c, c > 1$ can be used.

HITR – Hierarchic Transition Coverage. For transitions τ emanating from higher-level control states, different underlying basic control states can be active when τ is triggered. Hierarchic transition coverage aims at exercising τ once for every underlying basic control state being active. Let $\tau = (\ell_0, p, e, g, \alpha, \ell_1)$ a transition of some state machine sm and and

$$H = \{\ell \in BCS(sm) \mid \ell \neq \text{start}(p_{sm}(\ell)) \wedge \ell_0 \in [\ell, sm]\}$$

Set H contains all basic control states having ℓ_0 as their ancestor and which are not start states (because the state machine never resides in a start state). The HITR requires to test all objectives

$$G_\ell(s_{n+1}) \equiv s_{n+1}(\text{trigger}(\tau) \wedge \ell), \quad \ell \in H$$

expressed in LTL as

$$\mathbf{F}(\text{trigger}(\tau) \wedge \ell), \quad \ell \in H$$

EQ – Equivalence Class Coverage. The "classical" method of *equivalence class partition testing* is justified for applications where certain sets A of states are processed in an equivalent manner, typically by using the same data transformation f on each member of such a partition. In this situation, it is possible to select a few members from A for testing whether an illegal mutation of f has been implemented. As a consequence equivalence class partition testing has a relationship to mutation testing, where test data sets are expressly selected for the purpose of uncovering certain types of mutations.

For adequate test strength of equivalence class testing of reactive systems it is necessary to traverse on certain paths between classes. As a consequence, test objectives are of the form

$$G(s_0, \ldots, s_{n+1}) \equiv \bigwedge_{i=0}^{n+1} \left(s_i(\bigwedge_{p \in A_i} p) \right) \wedge \overline{G}(s_{n+1}, \ldots, s_{n+k})$$

where A_i are equivalence classes, each characterised as a set of atomic propositions p over the model variables. Test objectives of this type specify that the SUT performs a trace $\pi = s_0 \ldots s_{n+1}$, such that each s_i is member of a given class A_i. After the last class has been reached in state s_{n+1}, additional condition $\overline{G}(s_{n+1}, \ldots, s_{n+k})$ specifies a trace continuation $s_{n+2} \ldots s_{n+k}$ that is suitable to uncover errors in the data transformation applied on members of A_{n+1}. Depending on the type of data transformations f, different numbers of conditions $\overline{G}$ have to be applied, because a mutant f' of data transformation f may yield the same transformation results as f for some states of A_{n+1}. The classes A_i can be derived from the model AST in a syntactic way by collecting the constraints guarding behaviours using identical data transformations. The feasibility of traces $\pi = s_0 \ldots s_{n+1}$ visiting sequences $A_0 \ldots A_{n+1}$, however, has to be explored using an SMT solver.

In LTL such a test objective is expressed as

$$\mathbf{F}\left((\bigwedge_{p \in A_0} p) \wedge (\mathbf{X}(\bigwedge_{p \in A_1} p) \wedge (\ldots \wedge (\mathbf{X}((\bigwedge_{p \in A_{n+1}} p) \wedge \overline{G}) \ldots) \right)$$

A formalised approach to equivalence class partition testing has been worked out in [48], but without formal justification of the test strength of this method. Indeed, equivalence class partition testing can be applied to achieve exhaustive test results, so that any error in the SUT will be uncovered, if certain boundary conditions are met. This property can be applied to justify the partitions used in a testing campaign. This is explored in Detail in Chapter 10.

For SoS testing the EQ strategy will be applied to identify equivalent behaviours of constituent systems (for example, classes of certain mission threads), so that only some representatives of each class have to be executed during SoS system integration tests.

BV – Boundary Value Test Coverage. The well-known technique of Boundary value testing is closely linked to EQ testing, since it specifies special candidates to be selected from each equivalence class partition: by taking states s_i at the boundary of classes A_i, the error detection strength is generally increased, because typical bugs like the confusion of $<$ and $\leq$ can be uncovered using such boundary candidates.

For reactive real-time systems boundary value testing applies to timed traces $\pi = s_0 \dots s_{n+1}$, so that as many states s_i as possible should lie on the boundary of a class A_i. Boundary values come both from the value and from the time domain (*"just before the timer elapses, the expected event is received ..."*).

The logical formulas for expressing test objectives look like the ones defined for equivalence class testing objectives, but they additionally require that a subset of visited states should lie at the boundary of their class.

$$G_B(s_0, \dots, s_{n+1}) \equiv$$
$$\bigwedge_{i=0}^{n+1} \Big(s_i(\bigwedge_{p \in A_i}(p \wedge (i \in B \Rightarrow s_i(\text{boundary}A_i)))) \Big) \wedge$$
$$\overline{G}(s_{n+1}, \dots, s_{n+k}), \quad B \subseteq \{0, \dots, n+1\}$$

Predicates boundaryA_i can be mechanically generated from a syntactic analysis of the atomic proposition contained in A_i.

For robustness testing the conditions are changed in a way that state s_{n+1} lies in a neighbouring class of A_{n+1} and there just at its boundary.

MCDCHITR – MC/DC Hierarchic Transition Coverage. This HITR variant exercises high-level transitions with different guard valuations, as defined for MCDC coverage above.

The model-based coverage criteria specified so far are well known and accepted in MBT, and some of them have close relationships with code coverage strategies. This is not surprising, since programs are models, too, so code coverage can be regarded as model coverage. The following strategy is still a research topic, and it has been specifically designed by the authors for testing large concurrent systems, where complexity does not allow the representation of the SUT model as a single state machine, built from the product of several concurrent state machines[3].

BCSPAIRS – Basic Control State Pairs Coverage. A model state s is a function mapping all symbols (variables, basic control states, time) from a set V to their current value in some domain D. We also call s a *state vector* and optionally use vector notation $(s(v_1), \dots, s(v_n))$ if $V = \{v_1, \dots, v_n\}$. In model state s the active *basic control state vector* is the vector $(\ell_1, \dots, \ell_m)$ indexed of the number of concurrent state machines sm_i in the model, so that each basic control state of this vector is part of the active configuration in state s, that is, if $\bigwedge_{i=1,\dots,m} s(\ell_i)$ holds.

Large concurrent systems consist of so many state machines that test suites could never cover all basic control state vectors (let alone all complete state vectors) from the concurrent state machines involved. The basic control state pairs coverage considers pairs of state machines in writer/reader relationship. For these the possible combinations of control state pairs should be covered by test

[3]If such a product construction were possible, test strategies introduced successfully for verifying sequential systems could be applied for the concurrent one, too.

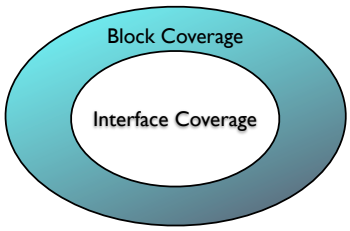

Figure 9.6: Structural coverage strategies for MBT.

suites, since exercising all of these pairs is likely to uncover errors in the writer/reader interaction. Let $RW \subseteq SM \times SM$ the relationship characterising writer/reader pairs. Then the BCSPAIRS strategy aims at covering all test cases of the form

$$G_{(\ell_1,\ell_2)}(s_{n+1}) \equiv s_{n+1}(\ell_1 \wedge \ell_2), \quad (\ell_1, \ell_2) \in RW$$

written in LTL as

$$\mathbf{F}(\ell_1 \wedge \ell_2), \quad (\ell_1, \ell_2) \in RW$$

IC – Interface Coverage. Interface coverage is achieved by changing the component values of a given interface. If the interface vector components are $(x_1, \ldots, x_n)$, the test goals are therefore

$$G_J(s_n, s_{n+1}) \equiv \bigwedge_{i \in J} s_{n+1}(x_i) \neq s_n(x_i), \quad J \subseteq \{1, \ldots, n\}$$

This goal specifies that for a given subset of interface components identified by indexes from J the valuation should change in the $(n+1)^{th}$ transition step. In LTL this is expressed as

$$\forall i \in J : \exists d_i \in D : \mathbf{F}\left((\bigwedge_{i \in J} x_i = d_i) \wedge \mathbf{X}(\bigwedge_{i \in J} x_i \neq d_i) \right)$$

Input interface coverage should be achieved for as many subsets $J \subseteq \{1, \ldots, n\}$ as possible, since some combinations of simultaneous changes may provoke faulty SUT reactions. For testing large GALS systems, the dimension n of interface vectors would generally be far too large to test all the $2^n - 1$ index combinations J. Here methods like pairwise testing in combination with orthogonal array are more adequate, but even the number $\binom{n}{2}$ of pairs will be too large for exhaustive coverage. We therefore apply data flow analysis techniques on the model in order to determine which pairs (x_i, x_j) are jointly used[4] in state machines or operations.

BC – Block Coverage. The lowest level of block coverage is achieved by triggering some behaviour associated with the block, such as a transition of an underlying state machine or the execution of an operation defined in the block. Since blocks and their descendants may be regarded as sub-models, all behavioural coverage criteria listed in this section may be applied locally to the block. This aspect is relevant in regression testing, where the test focus might be on single blocks of a model which has been changed since the last baseline. It is therefore adequate to allow MBT tool users to specify the strategies to be applied on a per-block basis (and not just for the whole model).

[4]The use may either be directly, when referring to x_i, x_j or indirectly when using some variable v where previous assignments from x_i, x_j have been performed.

9.7 User-Defined Test Cases

As described in Section 9.6, test cases specify traces $\pi = s_0 \dots s_n$ which are suitable *witnesses* to test a certain test objective. Obviously the BMC instance notation

$$J(s_0) \wedge \bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \wedge G(s_0, \dots, s_{n+1})$$

introduced above, while being appropriate as input to an SMT solver, is not a suitable candidate for manual specification of symbolic test cases. In the previous chapter, however, we have seen that certain types of BMC instances may be equivalently expressed by LTL formulas.

While LTL formulas are well-suited to specify computations fulfilling a wide variety of constraints, it has to be noted that it is also capable of defining properties of computations that will never be tested in practice, because they can only be verified on infinite computations and not on finite trace prefixes thereof (e. g., fairness properties). It is therefore desirable to identify a subset of LTL formulas that are tailored to the testers' need for specifying finite traces with certain properties. This subset is called *SafetyLTL* and described in the next section.

9.7.1 SafetyLTL

SafetyLTL has been introduced in [90], and is suitable for defining safety properties of computations, that is, properties that can always be falsified on a finite computation prefix. SafetyLTL can be syntactically characterised by the following rules.

- Negation is only allowed before atomic propositions (so-called *negation normal form*).
- Disjunction $\vee$ and conjunction $\wedge$ are always allowed.
- Next operators $\mathbf{X}$, globally operators $\mathbf{G}$ and weakly-until operators $\mathbf{W}$ are allowed.
- Semantically equivalent formulas also belong to SafetyLTL.

Recall that the weakly-until operator is defined as

$$\phi \,\mathbf{W}\, \psi \equiv_{\text{def}} (\phi \,\mathbf{U}\, \psi) \vee \mathbf{G}\phi$$

Using $\mathbf{W}$, the more common until operator can be expressed by

$$\phi \,\mathbf{U}\, \psi \equiv (\phi \,\mathbf{W}\, \psi) \wedge \mathbf{F}\psi$$

and for finite traces $\pi = s_0 \dots s_n$ formula $\mathbf{F}\psi$ can be expressed as

$$\psi \vee \mathbf{X}\psi \vee \mathbf{XX}\psi \vee \mathbf{XXX}\psi \vee \dots \vee \underbrace{\mathbf{X} \dots \mathbf{X}}_{n \text{ times}} \psi$$

For finite traces π, fulfilling $\mathbf{G}\psi$ means that π does not *violate* this formula. We could use the notation

$$\psi \wedge \mathbf{X}\psi \wedge \mathbf{XX}\psi \wedge \mathbf{XXX}\psi \wedge \dots \wedge \underbrace{\mathbf{X} \dots \mathbf{X}}_{n \text{ times}} \psi$$

as an alternative to $\mathbf{G}\psi$.

9.7.2 Encoding SafetyLTL Formulas as BMC Instances

The semantic rules for evaluating LTL formulas on finite trace segments $s_i.s_{i+1}\ldots s_k$ are specified using notation $\langle\varphi\rangle_i^k$. The recursive rules for evaluating the truth value of $\langle\varphi\rangle_i^k$ can be directly transformed into an algorithm unrolling $\langle\varphi\rangle_i^k$ into a proposition no longer involving any temporal operators ($\mathbf{F}, \mathbf{G}, \mathbf{X}, \mathbf{U}, \mathbf{W}$), but referring to variable valuations in states $s_i, s_{i+1}\ldots, s_k$ and Boolean operators $\neg, \wedge, \vee$ only. Observe that we omit the semantics for $\mathbf{G}$ here, because our witnesses are always represented by finite trace segments $s_i.s_{i+1}\ldots s_k$ without loops, whereas $\mathbf{G}\varphi$ only holds true if the trace segment has a lasso shape, where previous state on the segment is re-visited, thereby creating a cycle. The BMC semantics of $\mathbf{G}$ is discussed in detail in [21].

The remaining transformation rules applicable for data validation are (symbols p denote atomic propositions)

$$
\begin{array}{rcl}
\langle\varphi\rangle_i^k & \equiv & \text{false for all } i > k \\
\langle p\rangle_i^k & \equiv & p[s_i(v)/v \mid v \in free(p)] \text{ Note that bound}(p) = \varnothing \\
\langle\neg\varphi\rangle_i^k & \equiv & \langle\varphi\rangle_i^k \text{ is false} \\
\langle\varphi \wedge \psi\rangle_i^k & \equiv & \langle\varphi\rangle_i^k \text{ and } \langle\psi\rangle_i^k \text{ are true} \\
\langle\varphi \vee \psi\rangle_i^k & \equiv & \langle\varphi\rangle_i^k \text{ or } \langle\psi\rangle_i^k \text{ are true} \\
\langle(\exists b : \varphi)\rangle_i^k & \equiv & \langle\varphi\rangle_i^k \wedge \bigwedge_{j=i}^{k-1} (s_j(b) = s_{j+1}(b)) \\
 & & \text{Note that } b \text{ occurs free in RHS formula} \\
 & & \text{and extends domains of } s_j, s_{j+1}, \ldots, s_k \text{ by } b \\
\langle\varphi \mathsf{U} \psi\rangle_i^k & \equiv & \langle\psi\rangle_i^k \vee (\langle\varphi\rangle_i^k \wedge \langle\varphi[b'/b \mid b \in \text{bound}(\varphi)]\mathsf{U}\psi\rangle_{i+1}^k) \\
\langle\varphi \mathsf{W} \psi\rangle_i^k & \equiv & \langle\psi\rangle_i^k \vee (\langle\varphi\rangle_i^k \wedge ((i < k \wedge \langle\varphi[b'/b \mid b \in \text{bound}(\varphi)]\mathsf{W}\psi\rangle_{i+1}^k) \vee i = k)) \\
\langle\mathsf{X}\varphi\rangle_i^k & \equiv & \langle\varphi\rangle_{i+1}^k \\
\langle\mathsf{F}\varphi\rangle_i^k & \equiv & \bigvee_{j=i}^{k} \langle\varphi\rangle_j^k
\end{array}
$$

These transformation rules explain how to transform a LTL formula into an "ordinary" proposition $G(s_0, \ldots, s_{n+1})$, representing a test objective or a violation of desired model properties (see next section). In conjunction with the condition that the solution trace shall be a valid trace of the model. This results exactly in a BMC instance of the form discussed above, and this instance can be handled by the SMT solver.

Example 9.7.1. Consider the BMC evaluation of property $\phi \equiv (\exists b : y = b \wedge \mathbf{X}(y = b + 1))\mathbf{U}(x > 10)$, whose intuitive meaning is "variable y shall be incremented by 1 in each step, until a state is reached where $x > 10$ holds".

Suppose we wish to evaluate ϕ on trace segment $s_0.s_1.s_2$, that is,

$$\langle(\exists b : y = b \wedge \mathbf{X}(y = b + 1))\mathbf{U}(x > 10)\rangle_0^2$$

Applying the rules above, this is unrolled to

$$
\begin{array}{l}
\langle(\exists b : y = b \wedge \mathbf{X}(y = b + 1))\mathbf{U}(x > 10)\rangle_0^2 \equiv \\
\langle(x > 10)\rangle_0^2 \vee \\
(\langle(\exists b : y = b \wedge \mathsf{X}(y = b + 1))\rangle_0^2 \wedge \\
\langle(\exists b' : y = b' \wedge \mathsf{X}(y = b' + 1))\mathsf{U}(x > 10)\rangle_1^2) \equiv \\
(s_0(x) > 10) \vee \\
(\langle(y = b) \wedge \mathsf{X}(y = b + 1))\rangle_0^2 \wedge \\
\bigwedge_{j=0}^{1}(s_j(b) = s_{j+1}(b)) \wedge \\
\langle(\exists b' : y = b' \wedge \mathsf{X}(y = b' + 1))\mathsf{U}(x > 10)\rangle_1^2) \equiv \\
(s_0(x) > 10) \vee \\
((s_0(y) = s_0(b)) \wedge \mathsf{X}(s_1(y) = s_1(b) + 1)) \wedge \\
\bigwedge_{j=0}^{1}(s_j(b) = s_{j+1}(b)) \wedge \\
((s_1(x) > 10) \vee \\
(s_1(y) = s_1(b') \wedge s_2(y) = s_2(b') + 1) \wedge (s_1(b') = s_2(b')) \wedge \\
\langle(\exists b'' : y = b'' \wedge \mathsf{X}(y = b'' + 1))\mathsf{U}(x > 10)\rangle_2^2) \equiv \\
(s_0(x) > 10) \vee \\
((s_0(y) = s_0(b)) \wedge \mathsf{X}(s_1(y) = s_1(b) + 1)) \wedge \\
\bigwedge_{j=0}^{1}(s_j(b) = s_{j+1}(b)) \wedge \\
((s_1(x) > 10) \vee \\
((s_1(y) = s_1(b')) \wedge (s_2(y) = s_2(b') + 1) \wedge (s_1(b') = s_2(b')) \wedge \\
((s_2(x) > 10) \vee ((s_2(y) = s_2(b'')) \wedge \text{false})))
\end{array}
$$

□

9.8 Bounded Model Checking of LTL Properties

For bounded model checking, the same transformation of LTL formulas φ to BMC instances

$$J(s_0) \wedge \bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \wedge G(s_0, \ldots, s_{n+1})$$

is performed as explained in the previous section. The interpretation of results, however, will usually be different than in the case where φ denotes a symbolic test case.

- If φ denotes a symbolic test case, it is expected that a solution will be found, and this witness contains the concrete test inputs (input vectors and time stamps) for to the SUT to be applied when testing this test case.
- If φ denotes an unwanted property of the model, any solution will uncover a model error, and the witness represents a debugging aid in order to pinpoint the erroneous part of the model.

Chapter 10

Equivalence Class Testing

Note. The results presented in this chapter have been elaborated in cooperation with the EU FB7 research project COMPASS which is funded under grant agreement no.287829. The work has been submitted to ICTSS 2013.

10.1 Introduction

Motivation.

Equivalence class testing is a well-known heuristic approach to testing software or systems whose state spaces, inputs and/or outputs have value ranges of a cardinality inhibiting exhaustive enumeration of all possible values within a test suite. The heuristic suggests to create *equivalence class partitions* structuring the input or output domain into disjoint subsets for which *"the behavior of a component or system is assumed to be the same, based on the specification"* [91, p. 228]. If this assumption is justified it suffices to test "just a few" values from each class, instead of exploring the behavior of the system under test (SUT) for each possible value. In order to investigate that the SUT respects the boundaries between different equivalence class partitions *boundary values* are selected for each class, so that equivalence class and boundary value testing are typically applied in combination. As an alternative to deriving equivalence class partitions from the specification, the structure of the SUT or its model can be analyzed: classes are then defined as sets of data leading to the same execution paths [27, B.19].

For testing safety-critical systems the justification of the equivalence class partitions selected is a major challenge. It has to be reasoned why the behaviour of the SUT can really be expected to be equivalent for all values of a class, and why the number of representatives selected from each class for the test suite is adequate. While being quite explicit about the code coverage to be achieved when testing safety-critical systems, standards like [27, 88, 62] do not provide any well-defined acceptance conditions for equivalence class partitions to be sufficient.

Main Contributions.

In this chapter we present rules for generating input equivalence class partitions, whose justification is given by the fact that they lead to an *exhaustive* test suite: under certain hypotheses the generated classes and the test data selected from them *prove* conformance between a specification model and its implementation, if the latter passes all tests of this suite. The algorithm is applicable in a model-based testing context, provided that the behavioural semantics of the modelling formalism can be expressed

using Kripke Structures. The equivalence class partitioning strategy is elaborated and proven to be exhaustive on Kripke Structures. As an example of a concrete formalism, we illustrate how the strategy applies to SysML state machine models [77]. To our best knowledge, this is the first formal justification of the well-known equivalence class testing principle (see Section 10.6 for a discussion of related work).

Example 10.1.1. The following example describes a typical system of the class covered by our input equivalence class partition testing strategy. It will be used throughout the chapter for illustrating the different concepts and results described in this chapter. The example is taken – in simplified form, in order to comply with the space limitations of this publication – from the specification of the European Train Control System ETCS and describes the required behaviour of the *ceiling speed monitoring* which protects trains from overspeeding, as specified in [95, 3.13.10.3]. The interface is shown in Figure 10.1. The I/O variables have the following meaning.

Interface	**Description**
V_{est}	Current speed estimation [km/h]
V_{MRSP}	Applicable speed restriction [km/h] (MRSP = Most Restrictive Speed Profile)
W	Warning to train engine driver at driver machine interface (DMI) (1 = displayed, 0 = not displayed)
EB	Emergency brake (1 = active, 0 = inactive)

Figure 10.1: Interface of the ETCS ceiling speed monitoring function (simplified).

The behaviour of the ceiling speed monitoring function is specified by the UML (or SysML) state machine shown in Figure 10.2. The function gives a warning to the train engine driver if the currently applicable speed limit V_{MRSP} is not observed, but the actual estimated speed V_{est} does not exceed the limit too far. If the upper threshold for the warning status is violated (this limit is specified by guard conditions g_{ebi_1} or g_{ebi_2}), the emergency brake is activated. After such an *emergency brake intervention* has occurred, the brakes are only released after the train has come to a standstill. □

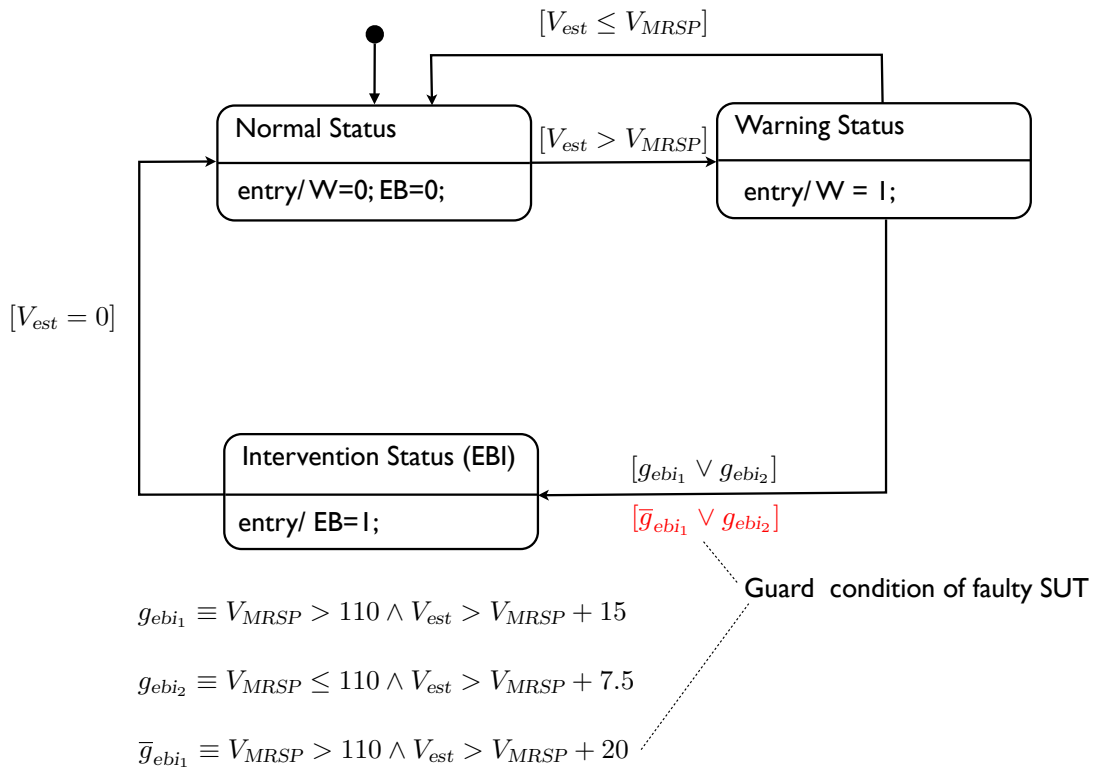

Figure 10.2: State machine of the ETCS ceiling speed monitoring function.

Overview and Proof Sketch.

In Section 10.2 the basic concepts about *Reactive Kripke Structures* – a specialisation of general Kripke Structures which is suitable for application in a reactive systems context – are introduced. In Section 10.3 it is shown how input equivalence class partitionings for Reactive Kripke Structures are constructed. In Section 10.4, two test hypotheses are presented, whose validity allows us to prove that our equivalence class partitioning and test data selection principle leads to an exhaustive test suite (Theorem 1). While this theorem states that I/O equivalence can be established using a finite input alphabet only (though the input data types may be infinite), it does not state whether the number of input traces needed is finite. In Section 10.5 we therefore show by means of this theorem, that Reactive Kripke Structures associated with input equivalence partitionings can be abstracted to deterministic finite state machines. Then the well-known W-Method can be applied to establish a *finite* exhaustive test suite proving I/O equivalence between specification model and SUT. Section 10.6 discusses related work, and we conclude with a discussion of the results obtained and a conjecture about an extension of the main theorem's validity in Section 10.7.

10.2 Reactive Kripke Structures

10.2.1 Notation and Definitions

Let $K = (S, S_0, R, L, AP)$ a Kripke Structure (KS) with state space S, initial states $S_0 \subseteq S$, transition relation $R \subseteq S \times S$ and labelling function $L : S \to \mathbb{P}(AP)$, where AP is a set of atomic propositions. We specialise on state spaces over variable valuations: let V a set of variable symbols for variables $v \in V$ with values in some domain $D = \bigcup_{v \in V} D_v$. The state space S of K is the set of all variable valuations $s : V \to D$.[1] It is required throughout the chapter that the labelling function shall be consistent with, and determined by these variable valuations, in the sense that AP contains propositions with free variable in V and[2]

$$\forall s \in S : L(s) = \{p \in AP \mid s(p)\}$$

Since s satisfies exactly the propositions contained in $L(s)$, is satisfies the negation of all propositions in the complement, that is, $\forall p \in AP - L(s) : \neg s(p)$.

K is called a *Reactive Kripke Structure (RKS)* if it satisfies the following additional properties.

1. V can be partitioned into disjoint sets $V = I \cup M \cup O$ called input variables, (internal) model variables, and output variables, respectively.[3]

2. Transitions leave either the input vectors or the internal and output state vectors unchanged, that is,
$$\forall (s, s') \in R : s'|_I = s|_I \vee s'|_{M \cup O} = s|_{M \cup O}$$

3. The state space can be partitioned into states from where only input changing transitions are possible, and those from where only internal and output changing transitions are possible. The former states are called *quiescent*, the latter *transient*.
$$\begin{aligned} \exists S_Q, S_T \subseteq S : S = S_Q \cup S_T \wedge S_Q \cap S_T = \varnothing \wedge \\ \forall (s, s') \in R : s \in S_Q \Rightarrow s'|_{M \cup O} = s|_{M \cup O} \wedge \\ \forall (s, s') \in R : s \in S_T \Rightarrow s'|_I = s|_I \end{aligned}$$

4. All initial states have the same internal and output variable valuations, and all possible inputs are allowed in initial states[4].
$$\exists \overline{s} : M \cup O \to D : S_0 = \{\{\vec{x} \mapsto \vec{c}\} \oplus \overline{s} \mid \vec{c} \in D_I\}$$

5. The input vector may change without any restrictions.
$$\forall s \in S_Q, s' \in S : s'_{M \cup O} = s|_{M \cup O} \Rightarrow (s, s') \in R$$

[1] The state space is always total in the sense that all $s : V \to D$ are elements of S. This allows us to assume that specification models K and implementations K' operate on the same state space S, possibly with differing subsets of *reachable* states.

[2] We use notation $s(p)$ for the Boolean expression p, where every free variable $v \in var(p)$ has been replaced by its current value $s(v)$ in state s. For example, $s(x < y)$ is true if and only if $s(x) < s(y)$ holds. Observe that this can be alternatively written as $s \models p$, or $p[s(v)/v \mid v \in V]$.

[3] Frequently we use input vectors $\vec{c}$ to the system, $\vec{c}$ is an element of $D_I = D_{x_1} \times \cdots \times D_{x_{|I|}}$, where $x_1, \ldots, x_{|I|}$ are the input variables. Changing the valuation of all input variables of a state s_0 to $\vec{c} = (c_1, \ldots, c_{|I|})$ is written as $s_1 = s_0 \oplus \{\vec{x} \mapsto \vec{c}\}$. State s_1 coincides with s_0 for all but the the input variables, and $s_1(x_i) = c_i, i = 1, \ldots, |I|$.

[4] Observe that initial states may be quiescent or transient.

6. Internal and output state changes are deterministic in the sense that they only depend on the current state valuation.

$$\exists T : S_T \to S_Q : \forall s \in S_T, s' \in S : (s, s') \in R \Rightarrow s' = T(s)$$

Function T can be extended to the complete state space by defining $\forall s \in S_Q : T(s) = s$.

7. Unreachable states s are elements of S_Q, so that the transition relation is also defined on $R(s, \cdot)$, but only input changes may occur.

The rules above imply that the transition relation of a RKS can be written as $R = \{(s, s') \mid s \in S_Q \wedge s'|_{M \cup O} = s|_{M \cup O}\} \cup \{(s, T(s)) \mid s \in S_T\}$. Observe that while transient states always have quiescent ones as post-states (this is stated in rule 7), quiescent states may have both transient and quiescent ones as post-states.

Example 10.2.1. Consider the UML/SysML state machine described in Example 1. Its behavioural semantics can be described by a RKS $K = (S, S_0, R, L, AP)$ with variable symbols from $V = I \cup M \cup O$, $I = \{V_{est}, V_{MRSP}\}$, $M = \{\ell\}$, and $O = \{\mathsf{W}, \mathsf{EB}\}$. Sets I and O contain the interface variable symbols with domains $D_{V_{est}} = D_{V_{MRSP}} = [0, 350] \subseteq \mathbb{R}$ (maximum speed of ETCS trains under consideration is 350km/h). Symbol ℓ ("location") has values in $D_\ell = \{\mathsf{NS}, \mathsf{WS}, \mathsf{IS}\}$ and its valuation signifies the current control state 'Normal Status', 'Warning Status', or 'Intervention Status', respectively. The output symbols have values in $D_{\mathsf{W}} = D_{\mathsf{EB}} = \mathbb{B} = \{0, 1\}$. The state space S contains all valuations of these symbols, $S = V \to D$, with $D = [0, 350] \cup D_\ell$. Setting $D_I = [0, 350] \times [0, 350]$, the initial states are elements of $S_0 = \{s_0 \in S \mid \exists (c_0, c_1) \in D_I : s_0 = \{V_{est} \mapsto c_0, V_{MRSP} \mapsto c_1, \ell \mapsto \mathsf{NS}, \mathsf{W} \mapsto 0, \mathsf{EB} \mapsto 0\}\}$. Fixing the variable order to vector $(V_{est}, V_{MRSP}, \ell, \mathsf{W}, \mathsf{EB})$, we will from now on describe states s by their value vector $(s(V_{est}), s(V_{MRSP}), s(\ell), s(\mathsf{W}), s(\mathsf{EB}))$, so that an initial state s_0 is written as $(c_0, c_1, \mathsf{NS}, 0, 0)$. The transition relation R is specified by the predicate (see [35] about how to express transition relations as first order predicates)

$$\begin{aligned}
&R((V_{est}, V_{MRSP}, \ell, \mathsf{W}, \mathsf{EB}), (V'_{est}, V'_{MRSP}, \ell', \mathsf{W}', \mathsf{EB}')) \equiv \\
&\quad \textstyle\bigvee_{i=0}^{7} \varphi((V_{est}, V_{MRSP}, \ell, \mathsf{W}, \mathsf{EB}), (V'_{est}, V'_{MRSP}, \ell', \mathsf{W}', \mathsf{EB}')) \\
&\varphi_0 \equiv (\ell = \mathsf{NS} \wedge V_{est} \leq V_{MRSP} \wedge \ell' = \mathsf{NS} \wedge \mathsf{W}' = \mathsf{W} \wedge \mathsf{EB}' = \mathsf{EB}) \\
&\varphi_1 \equiv (\ell = \mathsf{NS} \wedge V_{est} > V_{MRSP} \wedge \ell' = \mathsf{WS} \wedge \mathsf{W}' = 1 \wedge \mathsf{EB}' = \mathsf{EB} \wedge V'_{est} = V_{est} \wedge V'_{MRSP} = V_{MRSP}) \\
&\varphi_2 \equiv (\ell = \mathsf{NS} \wedge (g_{ebi_1} \vee g_{ebi_2}) \wedge \ell' = \mathsf{IS} \wedge \mathsf{W}' = 1 \wedge \mathsf{EB}' = 1 \wedge V'_{est} = V_{est} \wedge V'_{MRSP} = V_{MRSP}) \\
&\varphi_3 \equiv (\ell = \mathsf{WS} \wedge V_{est} > V_{MRSP} \wedge \neg(g_{ebi_1} \vee g_{ebi_2}) \wedge \ell' = \mathsf{WS} \wedge \mathsf{W}' = \mathsf{W} \wedge \mathsf{EB}' = \mathsf{EB}) \\
&\varphi_4 \equiv (\ell = \mathsf{WS} \wedge V_{est} \leq V_{MRSP} \wedge \ell' = \mathsf{NS} \wedge \mathsf{W}' = 0 \wedge \mathsf{EB}' = 0 \wedge V'_{est} = V_{est} \wedge V'_{MRSP} = V_{MRSP}) \\
&\varphi_5 \equiv (\ell = \mathsf{WS} \wedge (g_{ebi_1} \vee g_{ebi_2}) \wedge \ell' = \mathsf{IS} \wedge \mathsf{W}' = \mathsf{W} \wedge \mathsf{EB}' = 1 \wedge V'_{est} = V_{est} \wedge V'_{MRSP} = V_{MRSP}) \\
&\varphi_6 \equiv (\ell = \mathsf{IS} \wedge V_{est} > 0 \wedge \ell' = \mathsf{IS} \wedge \mathsf{W}' = \mathsf{W} \wedge \mathsf{EB}' = \mathsf{EB}) \\
&\varphi_7 \equiv (\ell = \mathsf{IS} \wedge V_{est} = 0 \wedge \ell' = \mathsf{NS} \wedge \mathsf{W}' = 0 \wedge \mathsf{EB}' = 0 \wedge V'_{est} = V_{est} \wedge V'_{MRSP} = V_{MRSP})
\end{aligned}$$

The quiescent states are characterised by the pre-conditions ("unprimed conjuncts") in $\varphi_0, \varphi_3, \varphi_6$, the transient states by the pre-conditions in $\varphi_1, \varphi_2, \varphi_4, \varphi_5, \varphi_7$. Observe that in order to enforce the RKS rule 6 (transient states are followed by quiescent states), φ_2 specifies the direct transitions from control state NS to IS. Initial state $s_0 = (0, 90, \mathsf{NS}, 0, 0)$, for example, is quiescent; this follows from φ_0. In contrast to this, $s_1 = (95, 90, \mathsf{NS}, 0, 0) \in S_0$ is transient (φ_1 applies). The latter initial state applies in a situation where the ceiling speed monitoring controller is re-booted while the train is driving ($V_{est} = 95$), and the state is immediately left, since V_{est} exceeds the allowed speed V_{MRSP}. The atomic propositions AP and the labelling function L will be discussed in the examples below. □

10.2.2 Quiescent Reduction

The notion of transient states in RKS is semantically redundant. They only help to facilitate the mapping of concrete modelling formalisms (such finite state machines or UML/SysML state machines) into RKS. The redundancy of transient states is captured in the following definition.

Definition 6. *Given a Reactive Kripke Structure $K = (S, S_0, R, L)$ the Kripke structure $Q(K)$ defined by*

$$Q(K) = (Q(S), Q(S_0), Q(R), Q(L)),\ \ Q(S) = S_Q$$
$$Q(L) = L|_{S_Q} : S_Q \to \mathbb{P}(AP),\ \ Q(S_0) = \{T(s_0) \mid s_0 \in S_0\}$$
$$Q(R) = \{(s, s') \mid s, s' \in S_Q \wedge (R(s, s') \vee (\exists s'' \in S_T : R(s, s'') \wedge s' = T(s'')))\}$$

is called the quiescent reduction of K. □

The state space of $Q(K)$ consists of quiescent K-states only, and its labelling function is the restriction of L to quiescent states. The initial states of $Q(K)$ consist of the union of the quiescent initial K-states and the quiescent post-states of transient initial K-states (recall that T maps quiescent states to themselves and transient states to their quiescent post-states). The transition relation $Q(R)$ relates quiescent states already related in K, and those pairs of quiescent states that are related indirectly in K by means of an intermediate transient state.

Example 10.2.2. For the RKS described in Example 1, the quiescent reduction $Q(K)$ has initial states $Q(S_0) = \{(V_{est}, V_{MRSP}, \ell, \mathsf{W}, \mathsf{EB}) \mid V_{est} \leq V_{MRSP} \wedge \ell = \mathsf{NS} \wedge \mathsf{W} = 0 \wedge \mathsf{EB} = 0\} \cup \{(V_{est}, V_{MRSP}, \ell, \mathsf{W}, \mathsf{EB}) \mid V_{est} > V_{MRSP} \wedge \neg(g_{ebi_1} \vee g_{ebi_2}) \wedge \ell = \mathsf{WS} \wedge \mathsf{W} = 1 \wedge \mathsf{EB} = 0\} \cup \{(V_{est}, V_{MRSP}, \ell, \mathsf{W}, \mathsf{EB}) \mid (g_{ebi_1} \vee g_{ebi_2}) \wedge \ell = \mathsf{IS} \wedge \mathsf{W} = 1 \wedge \mathsf{EB} = 1\}$. The transition relation is given by

$$Q(R)((V_{est}, V_{MRSP}, \ell, \mathsf{W}, \mathsf{EB}), (V'_{est}, V'_{MRSP}, \ell', \mathsf{W}', \mathsf{EB}')) \equiv \bigvee_{i=0}^{7} \psi_i((V_{est}, V_{MRSP}, \ell, \mathsf{W}, \mathsf{EB}), (V'_{est}, V'_{MRSP}, \ell', \mathsf{W}', \mathsf{EB}'))$$
$$\psi_0 \equiv (\ell = \mathsf{NS} \wedge V'_{est} \leq V'_{MRSP} \wedge \ell' = \mathsf{NS} \wedge \mathsf{W}' = 0 \wedge \mathsf{EB}' = 0)$$
$$\psi_1 \equiv (\ell = \mathsf{NS} \wedge V'_{est} > V'_{MRSP} \wedge \neg(g'_{ebi_1} \vee g'_{ebi_2}) \wedge \ell' = \mathsf{WS} \wedge \mathsf{W}' = 1 \wedge \mathsf{EB}' = \mathsf{EB})$$
$$\psi_2 \equiv (\ell = \mathsf{NS} \wedge (g'_{ebi_1} \vee g'_{ebi_2}) \wedge \ell' = \mathsf{IS} \wedge \mathsf{W}' = 1 \wedge \mathsf{EB}' = 1)$$
$$\psi_3 \equiv (\ell = \mathsf{WS} \wedge V'_{est} \leq V'_{MRSP} \wedge \ell' = \mathsf{NS} \wedge \mathsf{W}' = 0 \wedge \mathsf{EB}' = 0)$$
$$\psi_4 \equiv (\ell = \mathsf{WS} \wedge V'_{est} > V'_{MRSP} \wedge \neg(g'_{ebi_1} \vee g'_{ebi_2}) \wedge \ell' = \mathsf{WS} \wedge \mathsf{W}' = 1 \wedge \mathsf{EB}' = \mathsf{EB})$$
$$\psi_5 \equiv (\ell = \mathsf{WS} \wedge (g'_{ebi_1} \vee g'_{ebi_2}) \wedge \ell' = \mathsf{IS} \wedge \mathsf{W}' = \mathsf{W} \wedge \mathsf{EB}' = 1)$$
$$\psi_6 \equiv (\ell = \mathsf{IS} \wedge V'_{est} > 0 \wedge \ell' = \mathsf{IS} \wedge \mathsf{W}' = \mathsf{W} \wedge \mathsf{EB}' = 1)$$
$$\psi_7 \equiv (\ell = \mathsf{IS} \wedge V'_{est} = 0 \wedge \ell' = \mathsf{NS} \wedge \mathsf{W}' = 0 \wedge \mathsf{EB}' = 0)$$

□

10.2.3 Traces

Traces of K are finite sequences of states related by R, including the empty sequence ε.

$$\text{Traces}(K) = \{\varepsilon\} \cup \{s_0 \dots s_n \in S^* \mid n \in \mathbb{N}_0 \wedge s_0 \in S_0 \wedge \bigwedge_{i=0}^{n-1} R(s_i, s_{i+1})\}$$

The last state of a finite sequence of states is denoted by $last(s_0 \dots s_n) = s_n$, and $tail(s_0 \dots s_n) = (s_1 \dots s_n)$, $tail(s_0) = \varepsilon$. Given trace $s_0 \dots s_n$ we define its *restriction* to symbols from $X \subseteq V$ by $(s_0 \dots s_n)|_X = (s_0|_X) \dots (s_n|_X)$.

10.2.4 Input Traces

Given a RKS $K = (S, S_0, R, L)$, we consider the effect of input traces on states in K's quiescent reduction $Q(K)$: an *input trace* $\iota = \vec{c}_0.\vec{c}_1 \dots$ is a finite sequence of input vectors $\vec{c}_i \in D_I$, that is, $\iota \in (D_I)^*$. The application of an input trace ι to a quiescent state $s \in Q(S)$ is written as s/ι and yields a trace of $Q(K)$ which is recursively defined by $s/\varepsilon = s,\ s/(\vec{c}_0.\iota) = s.(T(s \oplus \{\vec{x} \mapsto \vec{c}_0\})/\iota)$.

As in the definitions above, T denotes the function mapping quiescent K-states to themselves and transient ones to their quiescent post-states. Obviously each pair of consecutive states in trace

s/ι is related by transition relation $Q(R)$. We will be frequently interested in the last element of an input trace application to a state; therefore the abbreviation $s//\iota = last(s/\iota)$ is used. Since RKS are deterministic with respect to their reactions on input changes, $s//\iota$ is uniquely determined.

10.2.5 I/O Equivalence

Model-based testing always investigates some notion of I/O conformance: stimulating the SUT with an input trace ι, the observable behaviour should be consistent with the behaviour expected for ι according to the model. The following definitions specify aspects of I/O equivalence, as they are relevant in the context of Reactive Kripke Structures.

Definition 7. *Given the quiescent reduction $Q(K)$ of some RKS K, and quiescent states $s_0, s_1 \in Q(S)$.*

1. *If $(s_0/\iota)|_O = (s_1/\iota)|_O$ holds for all input traces ι, s_0 and s_1 are called* I/O equivalent*, written as $s_0 \sim s_1$, otherwise s_0 and s_1 are called* I/O distinguishable*.*
2. *If $(s_0/\iota)|_O = (s_1/\iota)|_O$ for some input trace ι, s_0 and s_1 are called* ι-equivalent*, written as $s_0 \overset{\iota}{\sim} s_1$, otherwise s_0 and s_1 are called* ι-distinguishable*.*

Definition 8. *Two RKS K, K' over the same variable symbols V are called* I/O equivalent *(written $K \sim K'$) if their quiescent reductions are equivalent in the sense that $\forall(s_0, s_0') \in Q(S_0) \times Q(S_0') : (s_0|_I = s_0'|_I \Rightarrow s_0 \sim s_0')$.*

10.3 Input Equivalence Class Partitionings Over Reactive Kripke Structures With Finite Outputs

In the remainder of this chapter we study the special case where our specification models K and implementations K' only have output and internal variables with finite domains. The term "finite" is to be interpreted here in the sense that these values can be enumerated with reasonable effort. This contrasts with the domains of input variables, which we allow to have infinite range (such as real values) or to have "very large" finite cardinality (such as floating point or large integer types), where an enumeration would be impossible, due to time and memory restrictions. As a consequence, it is possible to further restrict the sets AP of atomic propositions under consideration. Since all possible values of internal states and output variables can be explicitly enumerated, AP can be structured into disjoint sets

$$
\begin{aligned}
&AP = AP_I \cup AP_M \cup AP_O \\
&AP_I \subseteq \{p \mid p \text{ is atomic and } var(p) \subseteq I\} \\
&AP_M = \{m = \alpha \mid m \in M \wedge \alpha \in D_m\} \\
&AP_O = \{y = \beta \mid y \in O \wedge \beta \in D_y\}
\end{aligned}
$$

Example 10.3.1. Consider a SysML state machine transition

$$\boxed{C_0} \overset{[x<m+y]}{\longrightarrow} \boxed{C_1}$$

with $x \in I, m \in M, y \in O$, where $D_x = \mathbb{R}, D_y = \{0, 1\}, D_m = \{10, 11\}$. When transforming this machine into a RKS, the atomic propositions AP can be strictly separated according to their free variables being from I, M, or O, respectively. For example, $AP = \{\ell = C_0, m = 10, y = 0, x < 10, x < 11, x < 12\}$. □

Definition 9. *Given RKS $K = (S, S_0, R, L, AP)$ with finite outputs and internal state, and AP partitioned into $AP = AP_I \cup AP_M \cup AP_O$ as described above. If*

$$\forall s_0, s_1 \in S : (L(s_0) = L(s_1) \Rightarrow L(T(s_0)) = L(T(s_1)))$$

then AP is called an input equivalence class partitioning (IECP) *of K, and its* input classes *are specified by*

$$\mathcal{I} = \{\{\, \vec{c} \in D_I \mid \bigwedge_{p \in L(s) \cap AP_I} p[\vec{c}/\vec{x}] \wedge \bigwedge_{p \in AP_I - L(s)} \neg p[\vec{c}/\vec{x}] \,\} \mid s \in S \,\}$$

In presence of an IECP AP, all input changes $\{\vec{x} \mapsto \vec{c}\}, \{\vec{x} \mapsto \vec{d}\}$ to a state s satisfying the same input-related atomic propositions, that is, $L(s \oplus \{\vec{x} \mapsto \vec{c}\}) \cap AP_I = L(s \oplus \{\vec{x} \mapsto \vec{d}\}) \cap AP_I$, lead to post states satisfying the same atomic propositions. In particular, these post states all have the same internal state and the same outputs. Recall that T maps quiescent states to themselves, so the IECP property is only non-trivial for the transient states of an RKS.

Example 10.3.2. For the ceiling speed monitoring function, whose RKS K has been constructed in Example 1, atomic propositions

$$AP = \{V_{est} = 0, V_{est} > V_{MRSP}, V_{MRSP} > 110, V_{est} > V_{MRSP} + 7.5, V_{est} > V_{MRSP} + 15, \\ \ell = \mathsf{NS}, \ell = \mathsf{WS}, \ell = \mathsf{IS}, \mathsf{W}, \mathsf{EB}\}$$

introduce an IECP for K. Consider, for example, the states s_0 labelled by $L(s_0) = \{V_{est} > V_{MRSP}, \ell = \mathsf{NS}\}$. Each of these s_0 is transient and has a post state s_1 satisfying $V'_{est} = V_{est} \wedge V'_{MRSP} = V_{MRSP} \wedge \ell' = \mathsf{WS} \wedge \mathsf{W}$. As a consequence, all of these post-states are labelled by $L(s_1) = \{V_{est} > V_{MRSP}, \ell = \mathsf{WS}, \mathsf{W}\}$. □

The following Lemma shows that input traces applied to the same state and passing through the same sequences of input equivalence classes produce identical outputs.

Lemma 1. *Given RKS $K = (S, S_0, R, L, AP)$ with finite outputs and internal state as described above, so that AP is an IECP for K with input classes $\mathcal{I}$. Let $\iota = \vec{c}_1 \ldots \vec{c}_k$, $\tau = \vec{d}_1 \ldots \vec{d}_k$, $\vec{c}_i, \vec{d}_i \in D_I, i = 1, \ldots, k$, such that*

$$\forall i = 1, \ldots, k, \exists X_i \in \mathcal{I} : \{\vec{c}_i, \vec{d}_i\} \subseteq X_i$$

Then $\forall s \in S_Q : (s/\iota)|_{M \cup O} = (s/\tau)|_{M \cup O}$.

Proof. Let $X_i \in \mathcal{I}$ satisfying $\{\vec{c}_i, \vec{d}_i\} \subseteq X_i, \forall i = 1, \ldots, k$. Let $s \in S_Q$. Denote $s/\iota = s_0.s_1 \ldots s_k$, $s/\tau = r_0.r_1 \ldots r_k$, where $s = s_0 = r_0$. We prove by induction over $i = 0, \ldots, k$ that $s_i|_{M \cup O} = r_i|_{M \cup O}$. For $i = 0$ it is trivial, since $s = s_0 = r_0$. Suppose that the induction hypothesis holds for $i < k$, $s_i|_{M \cup O} = r_i|_{M \cup O}$. Since $s_i \oplus \{\vec{x} \mapsto \vec{c}_{i+1}\}|_{M \cup O} = r_i \oplus \{\vec{x} \mapsto \vec{d}_{i+1}\}|_{M \cup O}$ and, according to the assumptions of the lemma, $\{\vec{c}_{i+1}, \vec{d}_{i+1}\} \subseteq X_{i+1}$, we conclude that $L(s_i \oplus \{\vec{x} \mapsto \vec{c}_{i+1}\}) = L(r_i \oplus \{\vec{x} \mapsto \vec{d}_{i+1}\})$.The IECP property of AP now implies that also $L(T(s_i \oplus \{\vec{x} \mapsto \vec{c}_{i+1}\})) = L(T(r_i \oplus \{\vec{x} \mapsto \vec{d}_{i+1}\}))$, and by definition $T(s_i \oplus \{\vec{x} \mapsto \vec{c}_{i+1}\}) = s_{i+1}, T(r_i \oplus \{\vec{x} \mapsto \vec{d}_{i+1}\}) = r_{i+1}$, therefore $s_{i+1}|_{M \cup O} = r_{i+1}|_{M \cup O}$. This proves the lemma. □

Lemma 2. *Given RKS K with finite outputs and internal state as described above, and AP an IECP. Let $p_1, \ldots, p_n$ a set of fresh atomic propositions not contained in AP, with $var(p_i) \subseteq I, i = 1, \ldots, n$. Then $AP_2 = AP \cup \{p_1, \ldots, p_n\}$ is another IECP, called the* refinement of AP. *The input classes of AP_2, constructed according to Definition 9, are denoted by $\mathcal{I}_2$.*

Observe that IECP refinement according to Lemma 2 introduces new propositions in AP_I only, while AP_M and AP_O remain unchanged.

10.4 Test Hypotheses and Proof of Exhaustiveness

The input equivalence class testing strategy to be introduced in this section yield exhaustive tests, provided that the following two test hypotheses are met.

(TH1) Testability Hypothesis. There exists a RKS $K' = (S, S'_0, R', L', AP')$ with finite internal states and output as introduced in Section 10.3 describing the true behaviour of the SUT, and its state space S consists of valuation functions $s : V \to D$ for variables from V as specified for the reference model $K = (S, S_0, R, L, AP)$.

(TH2) Existence of Refined Equivalence Class Partitioning. For specification model $K = (S, S_0, R, L, AP)$ and SUT $K' = (S, S'_0, R', L', AP')$, both atomic proposition sets AP, AP' are IECP of K and K' with input classes $\mathcal{I}, \mathcal{I}'$, respectively, and $AP_M = AP'_M$, $AP_O = AP'_O$. Moreover, there exists an input partition refinement $AP_2 = AP_{2I} \cup AP_M \cup AP_O$, in the sense of Lemma 2, such that

$$\forall X \in \mathcal{I}, X' \in \mathcal{I}' : \exists X_2 \in \mathcal{I}_2 : (X \cap X' \neq \varnothing \Rightarrow X_2 \subseteq X \cap X')$$

Validity of (TH2) induces a finite input alphabet to K and K' which will be shown below to suffice for uncovering any violation of I/O equivalence between K and K'.

Definition 10. *Given RKS K, K' with finite internal state and outputs, and input equivalence class partitionings AP, AP' and AP_2 according to test hypothesis (TH2). Let $\mathcal{A}_I$ denote a finite subset of input vectors $\vec{c} \in D_I$ satisfying $\forall X \in \mathcal{I}_2 : \exists \vec{c} \in \mathcal{A}_I : \vec{c} \in X$. Then $\mathcal{A}_I$ is called an* input alphabet *for equivalence class partition testing of K' against K. For any nonnegative integer k, ${\mathcal{A}_I}^k$ is the set of all $\mathcal{A}_I$-sequences of length less than or equal to k (including the empty trace ε).*

Example 10.4.1. Let K the RKS of the ceiling speed monitor model constructed in Example 1, with IECP AP as given in Example 2. Now suppose that the SUT implementing the monitor model has an error, as indicated in Figure 10.2: it uses a faulty guard condition $\overline{g}_{ebi_1} \vee g_{ebi_1}$ instead of $g_{ebi_1} \vee g_{ebi_1}$. Its IECP (which, of course, would be unknown in a black box test) is $AP' = \{V_{est} = 0, V_{est} > V_{MRSP}, V_{MRSP} > 110, V_{est} > V_{MRSP} + 7.5, V_{est} > V_{MRSP} + 20, \ell = \mathsf{NS}, \ell = \mathsf{WS}, \ell = \mathsf{IS}, \mathsf{W}, \mathsf{EB}\}$. The IECP refinement of AP, $AP_2 = \{V_{est} = 0, V_{est} > V_{MRSP}, V_{MRSP} > 110, V_{est} > V_{MRSP} + 7.5, V_{est} > V_{MRSP} + 15, V_{est} > V_{MRSP} + 18.75, V_{est} > V_{MRSP} + 22.5, \ell = \mathsf{NS}, \ell = \mathsf{WS}, \ell = \mathsf{IS}, \mathsf{W}, \mathsf{EB}\}$ fulfils test hypothesis (TH2). Consider, for example the intersection of K input class $X = \{(V_{est}, V_{MRSP}) \mid V_{MRSP} > 110 \wedge V_{est} > V_{MRSP} + 15\}$ and the K' input class $X' = \{(V_{est}, V_{MRSP}) \mid V_{MRSP} > 110 \wedge V_{est} > V_{MRSP} + 7.5 \wedge \neg(V_{est} > V_{MRSP} + 20)\}$. Then the input class $X_2 = \{(V_{est}, V_{MRSP}) \mid V_{MRSP} > 110 \wedge V_{est} > V_{MRSP} + 15 \wedge \neg(V_{est} > V_{MRSP} + 18.75)\}$ of the refined IECP AP_2 is contained in the intersection $X \cap X'$. Indeed, any input from X_2 applied to the SUT in control state WS would reveal the erroneous guard condition, because K transits into IS, while K' remains in WS.

For practical application (since the IECP of the SUT is unknown), the input space D_I is systematically partitioned by intersecting the input-related propositions from AP with interval vectors, partitioning D_I into $|I|$-dimensional cubes. □

Theorem 1. *Given RKS $K = (S, S_0, R, L, AP)$, $K' = (S, S'_0, R', L', AP')$, such that AP, AP' are IECP of K and K' with input classes $\mathcal{I}, \mathcal{I}'$, respectively, and AP_2 is a refinement of AP according to test hypothesis (TH2). $\mathcal{I}_2$ contains the input classes associated with AP_2. Let $\mathcal{A}_I$ an input alphabet derived from $\mathcal{I}_2$ according to Definition 10. Then for any quiescent states $s \in S_Q$, $s' \in S'_Q$ and any*

input trace ι, there exists an input trace $\tau \in {\mathcal{A}_I}^*$ with the same length, such that $s/\iota|_O = s/\tau|_O$ and $s'/\iota|_O = s'/\tau|_O$. Hence, $s \overset{\iota}{\sim} s'$ if and only if $s \overset{\tau}{\sim} s'$.

Proof. If ι is empty, there is nothing to prove, since $\varepsilon \in \mathcal{A}_I$. Suppose therefore, that $\iota = \vec{c}_1 \dots \vec{c}_k$ with $k \geq 1$ and let $s/\iota = s_0.s_1 \dots s_k$, and $s'/\iota = s'_0.s'_1 \dots s'_k$, where $s_0 = s, s'_0 = s'$.

Consider the associated sequences of input classes $X_1 \dots X_k \in \mathcal{I}$ and $X'_1 \dots X'_k \in \mathcal{I}'$, where $\vec{c}_i \in X_i$ and $\vec{c}_i \in X'_i$, for all $i = 1, \dots, k$. Since $\vec{c}_i \in X_i \cap X'_i \neq \varnothing, \;\; i = 1, \dots, k$, (TH2) implies the existence of $X_{21}, \dots, X_{2k} \in \mathcal{I}_2$ such that

$$X_{2i} \subseteq X_i \cap X'_i, \;\; i = 1, \dots, k \quad (*)$$

According to Definition 10, we can select $\vec{d}_1, \dots, \vec{d}_k \in \mathcal{A}_I$, such that $\vec{d}_i \in X_{2i}$ for all $i = 1, \dots, k$. (*) implies $\vec{d}_i \in X_i \cap X'_i \;\; i = 1, \dots, k$. Therefore, setting $\tau = \vec{d}_1 \dots \vec{d}_k$, Lemma 1 may be applied to conclude that $(s/\iota)|_O = (s/\tau)|_O$ and $(s'/\iota)|_O = (s'/\tau)|_O$. Therefore $s \overset{\iota}{\sim} s' \Leftrightarrow s \overset{\tau}{\sim} s'$, and this completes the proof. □

10.5 Test Strategy

Given specification model $K = (S, S_0, R, L, AP)$ and SUT $K' = (S, S'_0, R', L', AP')$, and the refined IECP AP_2 with input classes $\mathcal{I}_2$ according to test hypothesis (TH2). Let $\mathcal{A}_I$ the input alphabet constructed from $\mathcal{I}_2$ as specified in Definition 10. Then AP, $\mathcal{A}_I$ and each $s_0 \in Q(S_0)$ induce a deterministic finite state machine (DFSM) abstraction $M(K, s_0) = (\mathcal{Q}, q_0, \mathcal{A}_I, D_O, \delta, \omega)$ of K with state space $\mathcal{Q} = \{[s] \mid s \in S_Q\}$, initial state $q_0 = [s_0]$, and input alphabet $\mathcal{A}_I$, where $[s] = \{r \in S_Q \mid r \sim s\}$. Let O the set of output variables of K. The output alphabet of $M(K, s_0)$ is defined by $D_O = D_{y_1} \times \ldots \times D_{y_{|O|}}$. The state transition function $\delta : \mathcal{Q} \times \mathcal{A}_I \to \mathcal{Q}$ of $M(K, s_0)$ is defined by

$$\delta(q, \vec{c}) = q_1 \;\text{ if and only if }\; \exists s \in S_Q : q = [s] \wedge q_1 = [s//\vec{c}]$$

The output function $\omega : \mathcal{Q} \times \mathcal{A}_I \to D_O$ of $M(K, s_0)$ is defined by

$$\omega(q, \vec{c}) = \vec{e} \text{ if and only if } \exists s \in S_Q : q = [s] \wedge (s//\vec{c})|_O = \{\vec{y} \mapsto \vec{e}\}$$

We extend the domain of the state transition function to input traces, $\overline{\delta} : \mathcal{Q} \times \mathcal{A}_I^* \to \mathcal{Q}^*$ by setting recursively $\overline{\delta}(q, \varepsilon) = q, \;\; \overline{\delta}(q, \vec{c}.\iota) = q.\overline{\delta}(\delta(q, \vec{c}), \iota)$. The output function can be extended to $\overline{\omega} : \mathcal{Q} \times \mathcal{A}_I^* \to D_O^*$ by setting $\overline{\omega}(q, \iota) = \vec{e}_0 \dots \vec{e}_k$, if and only if $\overline{\delta}(q, \iota) = [s_0] \dots [s_k]$ and $s_i|_O = \{\vec{y} \mapsto \vec{e}_i\}, i = 0, \dots, k$.

Lemma 3. *The DFSMs $M(K, s_0) = (\mathcal{Q}, q_0, \mathcal{A}_I, D_O, \delta, \omega)$ introduced above are well-defined.*

Proof. Let $q = [s]$ and $[r] = [s]$ for some $s, r \in S_Q$. Then $r \sim s$, and therefore $s//\vec{c} \sim r//\vec{c}$, and this shows that $\delta(q, \vec{c})$ is well-defined. Since all members of $[s//\vec{c}]$ coincide on O, this also shows that ω is well-defined. □

By construction, the DFSMs are minimal, because each pair of different states $[s_0] \neq [s_1]$ can be distinguished by an input trace resulting in different outputs when applied to $[s_0]$ or $[s_1]$, respectively. Since AP is an IECP, all K-states s_0, s_1 carrying the same label $L(s_0) = L(s_1)$ are I/O-equivalent, so $\{s_1 \mid L(s_1) = L(s)\} \subseteq [s]$ for all quiescent states of K. It may be the case, however, that some states carrying different labels are still I/O-equivalent, that is, $L(s_0) \neq L(s_1)$, but $\{s \mid L(s) = L(s_0)\} \cup \{s \mid L(s) = L(s_1)\} \subseteq [s_0] = [s_1]$. In analogy to $M(K, s_0)$, DFSMs $M(K', s'_0)$ can be

constructed from $K', AP', s_0 \in Q(S'_0)$, and the same input alphabet $\mathcal{A}_I$ as has been used for the DFSMs $M(K, s_0)$.

We write $M(K, s_0) \sim M(K', s'_0)$ and $q_0 \sim q'_0$, if and only if $\overline{\omega}(q_0, \iota) = \overline{\omega}'(q'_0, \iota)$ for every $\iota \in \mathcal{A}_I^*$. Note that this differs from I/O equivalence between K and K', where $s_0 \sim s'_0$ if and only if $(s_0/\iota)|_O = (s'_0/\iota)_O$ for every $\iota \in D_I^*$. The following theorem states that I/O equivalence between specification model K and an implementation K' can be established by investigating the equivalence of their associated DFSM, that is, using $\iota \in \mathcal{A}_I^*$ only.

Theorem 2. *With the notation above, the following statements are equivalent.*

- *K and K' are I/O equivalent, $K \sim K'$.*
- *$\forall s_0 \in Q(S_0), s'_0 \in Q(S'_0) : (s_0|_I = s'_0|_I \Rightarrow M(K, s_0) \sim M(K', s'_0))$.*

Proof. Obviously, $M(K, s_0) \sim M(K', s'_0) \Leftrightarrow q_0 \sim q'_0 \Leftrightarrow (\forall \tau \in \mathcal{A}_I^* : s_0 \overset{\tau}{\sim} s'_0)$. By Theorem 1, we have $(\forall \iota \in D_I^* : s_0 \overset{\iota}{\sim} s'_0) \Leftrightarrow (\forall \tau \in \mathcal{A}_I^* : s_0 \overset{\tau}{\sim} s'_0)$. Hence $s_0 \sim s'_0 \Leftrightarrow M(K, s_0) \sim M(K', s'_0)$. Now the assertion follows directly from the definition of $K \sim K'$ (Definition 8). □

Definition 11. *With the terms introduced above, a* transition cover *of $M(K, s_0)$ is a set of input traces $\iota \in \mathcal{A}_I^*$ satisfying the following condition: for any reachable state $q \in \mathcal{Q}$ and any $\vec{c} \in \mathcal{A}_I$, there is an input trace $\iota \in TC$ such that $\overline{\delta}(q_0, \iota) = q$ and $\iota.\vec{c} \in TC$.*

Definition 12. *With the terms introduced above and minimal $M(K, s_0)$, define a* characterisation set *W of $M(K, s_0)$ as a set of traces $\iota \in \mathcal{A}_I^*$, such that for all $q_1, q_2 \in \mathcal{Q}$, there exists an input trace $\iota \in W$ such that $\overline{\omega}(q_1, \iota) \neq \overline{\omega}(q_2, \iota)$.*

On DFSM $M(K, s_0)$ we can apply Chow's W-method [31] to conclude that the following finite test suite is exhaustive for testing I/O equivalence between K and K'.

Theorem 3. *With the terms introduced above, and $M(K, s_0)$ and $M(K', s'_0)$ minimal for any $s_0 \in Q(S_0), s'_0 \in Q(S'_0)$ with $s_0|_I = s'_0|_I$, let $TC(s_0), W(s_0)$ the transition cover and characterisation set of $M(K, s_0)$ as introduced above. Assume that $M(K, s_0)$ has n states and that $M(K', s'_0)$ has at most m states. Let $m_0 = \max(n, m)$. Then*

$$\mathcal{W}(K) = \bigcup_{[s_0] \in Q(S_0)/\sim} \left(TC(s_0).\mathcal{A}_I^{m_0 - n}.W(s_0)\right)$$

is an exhaustive test suite for testing SUT K' against specification model K.

Proof. Applying Chow's W-method [31] to $M(K, s_0)$ and $M(K', s'_0)$, $M(K, s_0)$ and $M(K', s'_0)$ are I/O equivalent if and only if they are $TC(s_0).\mathcal{A}_I^{m_0 - n}.W(s_0)$ equivalent[5]. Hence the assertion follows directly from Theorem 2. □

10.5.1 Complexity Considerations

Definition 10 determines the size of the input alphabet $\mathcal{A}_I$ as the number $k_2 = |\mathcal{A}_I|$ of input classes in the refined equivalence partitioning AP_2 according to test hypothesis (TH2). The number n of states in the DFSM associated with K is less or equal to the number $\overline{n}$ of labels $L(s), s \in S_Q$ (we get $n < \overline{n}$

[5] Observe that in [31], the author uses a slightly different notation, where $\mathcal{A}_I^i$ denotes the set of input traces with length i, while we use this term to denote the traces of length less or equal i.

if different labels $L(s_0) \neq L(s_1)$ are associated with I/O equivalent states). Let $m_0 = \max(n, m)$, where n is the number of states in $M(K, s_0)$, and m the number of states in $M(K', s'_0)$. Then according to [31, 96], the number of input traces contained in $TC(s_0).\mathcal{A}_I^{m_0-n}.W(s_0)$ is bounded by $n^2 \cdot k_2^{m_0-n+1}$. We have to execute several test suites of this type, their number is equal to $k = |Q(S_0)/_\sim|$, the number of equivalence classes derived from initial states of the quiescent reduction of K. In the worst case, all classes of K can be reached from some transient initial state, so $k \leq n$. This results in an upper bound of $k \cdot n^2 \cdot k_2^{m_0-n+1} \leq n^3 \cdot k_2^{m_0-n+1}$ test cases (that is, input traces) to be performed.

10.6 Related Work

Notable examples for exhaustive test methods have been given in [31, 92, 78, 94]. There exists a large variety of research results related to testing against hierarchic state machines similar to Harel's Statecharts or to UML state machines. We mention [46] as one representative and refer to the references given there. These contributions, however, mainly deal with the state machine hierarchy and do not tackle the problem of attributes from large input domains, which is the main motivation for the results presented here. In [24, pp. 205] large data domains in the context of state machine testing are addressed, but no formal justification of the heuristics presented there are given.

In model-based testing, the idea to use data abstraction for the purpose of equivalence class definition has been originally introduced in [49], where the classes are denoted as *hyperstates*, and the concept is applied to testing against abstract state machine models. Our results presented here surpass the findings described in [49] in the following ways: (1) while the authors of [49] introduce the equivalence class partitioning technique for abstract state machines only, our approach extracts partitions from the models' semantic representation. Therefore an exhaustive equivalence class testing strategy can be elaborated for any formalism whose semantics can be expressed by Kripke Structures. (2) The authors sketch for white box tests only how an exhaustive test suite could be created [49, Section 4]: the transition cover approach discussed there is only applicable for SUT where the internal state (respectively, its abstraction) can be monitored during test execution. (3) The authors only consider finite input sets whose values have been fixed *a priori* [49, Section 2], whereas our approach allows for inputs from arbitrary domains.

The approach to define equivalence classes as vectors of propositions over state variables introduces a natural abstraction of the state space. In the field of model checking, this data abstraction has been extensively investigated, see [35]. In particular, the concept of counter example guided abstraction refinement (CEGAR) introduces strategies for the gradual refinement of equivalence classes [33]. This has inspired the partition refinement introduced in this chapter for the ensuring compliance with the fault hypotheses.

The original application for CEGAR paradigm as advocated in [32, 33] starts with a coarse abstraction of the state space and tries to prove the validity of an ACTL formula. If a counter example exists, the abstraction is refined in order to investigate whether the violation of the assertion is just a false alarm only occurring in the original abstraction. In contrast to this original application of the CEGAR principle, it has been suggested in [20] to apply refinements in situations where it cannot be decided whether certain atomic propositions will evaluate to true or false in a given abstracted state. This idea can be applied in the context of the present chapter, when transforming a concrete model (for example, a SysML state machine) into a RKS. Our approach, however, is distinguished from [20] in the following ways: (1) while the application domain in [20] is property checking of concrete PLC code, we focus on black-box equivalence testing for a wider range of reactive control systems where

the SUT code is not available, but a specification model exists instead. (2) In [20] state spaces are abstracted using *conservative approximations*, so that no potential error states can be missed. In contrast to this, we deliberately use *under approximations* for which we are able to show that they suffice to uncover deviations of the SUT from the model, as long as certain fault hypotheses are fulfilled: these under approximations correspond to the input equivalence classes to be constructed for refined partitionings as specified in test hypothesis (TH2). Our refinement concept can also be applied for creating decision coverage of test models in general [11].

10.7 Conclusion and Future Work

In this chapter, a novel exhaustive test strategy for input equivalence class testing has been established. The main result (Theorem 1) shows that even in presence of infinite input data domains, a finite input alphabet can be identified, so that for every trace performed by specification model or implementation, there exists a trace using inputs from this finite alphabet only, but producing the same outputs as the original one. This result holds for arbitrary modelling formalisms, whose semantics may be expressed by Reactive Kripke Structures with input domains that may be infinite (or too large to be explicitly enumerated), but with internal states and outputs having a sufficiently small range to be enumerated in an explicit way. With the main theorem at hand, the well-known W-Method can be applied to identify a finite and at the same time exhaustive test suite. Using an abstraction of the Kripke Structures under consideration to deterministic finite state machines, we have proven that this method is applicable.

Further research will focus on the generalisation of the test strategy to Reactive Kripke Structures with arbitrary data domains for internal states and outputs. According to our conjecture, a result similar to Theorem 1 should hold in the general case. The equivalence classes under consideration, however, will no longer refer to system inputs only, but will be characterised by more general atomic propositions with inputs, internal state and outputs as free variables.

Chapter 11

Stochastic Model Checking

Note. The contents of this chapter has been published in Jan Peleska and Oliver Schulz: Reliability Analysis of Safety-Related Communication Architectures In E. Schoitsch (Ed.): SAFECOMP 2010. LNCS 6351, pp. 1-14, 2010 Reliability Analysis of Safety-Related Communication Architectures.

11.1 Introduction

11.1.1 Background: Safety Versus Reliability in Communicating Railway Control Systems

In safety related communication domains there are two important characteristics of communication architectures: Safety and reliability. In the railway domain the standard EN 50159-2 defines a basic design of communication architectures for safety related equipment. In general the standard splits the architecture into two parts: A safety layer, which must fulfil a specific safety integrity level (SIL) and a "grey channel" without any safety responsibility (see Fig. 11.1 and 11.2). Safety layers have to detect six different types of message errors to grant functional safety. The standard EN 50159-2 defines a defence matrix against these threats (Table 11.1, [28, 29]). The safety reaction on such errors must be a safe state, which usually stops the communication service until the system is reinitialised or reset by an operator. Therefore a safe communication reduces the fault tolerance against arbitrary transmission errors and lowers the reliability of the communication architecture. To improve the fault tolerance against message errors it is necessary to use a reliable message transmission service (e.g. ARQ, Automatic Repeat Request) before the safety check is executed. A reliable transmission service can be included in the safety layer, in the upper protocol layer of the grey channel or in both layers (Fig. 11.2).

A "naive" combination of fault-tolerance mechanisms in the grey channel and safety layers will not necessarily increase the overall fault-tolerance: if, for example, lost messages in the grey channel lead to re-transmissions after timeouts, the message eventually passed to the receiving safety layer may be out-dated and therefore has to be discarded. As a consequence, it is necessary to perform analyses whether – given a trustworthy estimate for the occurrence of basic transmission faults as classified in Table 11.1 – the fault-tolerance mechanisms deployed in the grey channel will really increase the overall reliability of the distributed safety-critical control system.

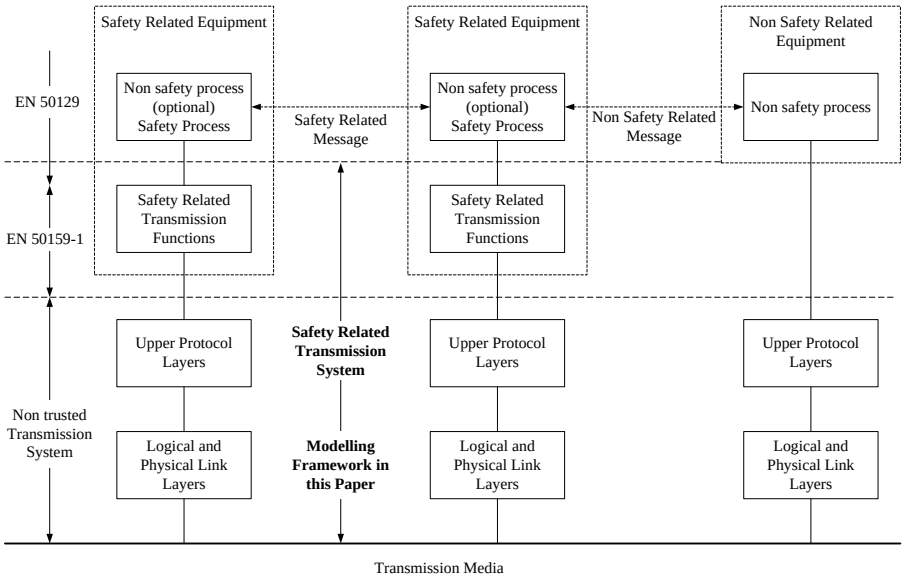

Figure 11.1: Structure of safety-related communication architecture (from [29]). The term "Non Safety Process (optional)" in the *Safety Related Equipment* block indicates that also processes without safety-relevance can be deployed in the safety-critical equipment.

Table 11.1: Threats

Defences / Threat	Sequence number	Time stamp	Time out	Src. and dst. ID	Feed-back message	Identification procedure	Safety code
Repetition	x	x					
Deletion	x						
Insertion	x			x	x	x	
Resequence	x	x					
Corruption							x
Delay		x	x				

11.1.2 Objectives and Contributions

In this paper we present a novel method for reliability analysis of safety-related communication architectures structured into safety layers and grey channels as described in the previous section. In this context reliability is defined as the probability that the overall system will perform its (deterministic) safety-related services in a given operational time period $[t_1, t_2]$ without interruption and resulting transition into stable safe state, though transmission faults may occur in the grey channel with a given probability (see IEC 60050(191) [61] for the general definition).

Our analysis approach uses a domain-specific modelling language (DSL) developed by the authors. This DSL facilitates modelling communication architectures and protocols, together with the fault hypotheses concerning the probabilistic occurrence of the basic faults listed in Table 11.1. These communication models are used to create *mutants*, that is, derived models showing erroneous behaviour resulting from one or more basic faults occurring in compliance with the fault hypotheses at various places in the communication architecture. For each mutant the probability of its occurrence

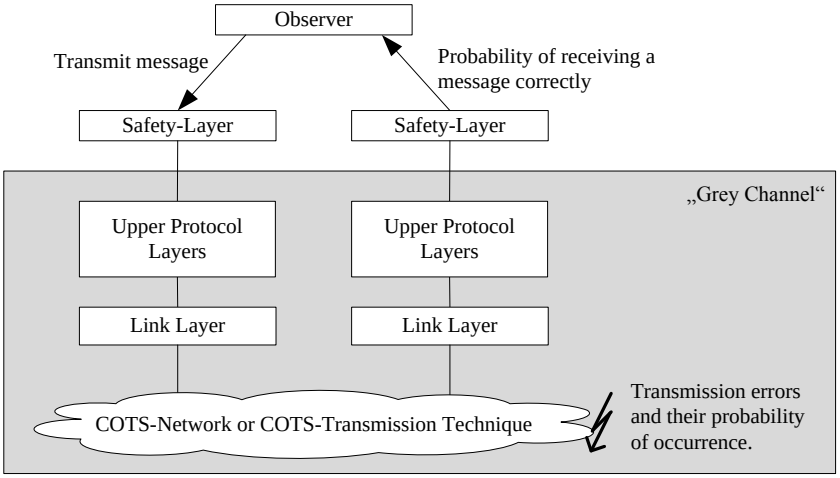

Figure 11.2: General Modelling Architecture

can be calculated. Since the mutants themselves show deterministic (erroneous) behaviour, conventional non-probabilistic model checkers can be used to analyse whether the safety-related services will still operate properly in presence of the behaviour specified by the mutant. Time constraints play an important rôle in the behaviour of the system layers involved; therefore we have chosen Timed Automata [7] for modelling the mutant behaviour and use the UPPAAL tool to perform the associated analyses. UPPAAL verifies or falsifies TCTL (Timed Computation Tree Logic) statements like $A\mathbf{G}\phi$[1] or $E\mathbf{F}\phi$[2] on a given UPPAAL model [17]. The verification goal $A\mathbf{G}(\text{SAFE} \wedge \neg\phi)$ is to show that the safety layer will always satisfy its safety-specification SAFE and never transit into stable safe state ϕ, despite of the faults occurring in the grey channel according to the mutant model under investigation. If a combination of faults on the grey channel leads to a violation of $A\mathbf{G}$SAFE the design has to be changed in any case, since a design-intrinsic safety violation that can be provoked by erroneous grey channel behaviour is not to be tolerated, regardless of the probability of its occurrence. If all mutants satisfy $A\mathbf{G}$SAFE, they are classified by their occurrence probability, and according to their satisfaction or violation of $A\mathbf{G}(\neg\phi)$. Then the resulting reliability of the overall system is calculated as the probability that only correct behaviour or mutants satisfying $A\mathbf{G}(\neg\phi)$ occur during the given operational time period.

Our modelling approach requires *transaction-oriented* processing of safety-related communication functions: it is assumed that each activity consists of a bounded number of communication and processing steps, such that (1) the success or failure of the activity can be clearly determined after this sequence, and, (2) the success of the actual transaction is stochastically independent on the success of preceding actions. In the context of safety-related communication architectures this restriction is not a severe one: applications usually proceed according to different protocol phases like system setup, connection request, transmission of one application-specific datagram, and going through each of these phases corresponds to processing transactions of a specific type $T_\ell, \ell = 1, \ldots, q$. A minor limitation is discussed in Section 11.5.

We have developed an integrated tool chain starting with the modelling phase supported by the MetaEdit+ meta case tool [66] which was also used to design the DSL. A model-to-text generator

[1] "Always globally ϕ": in every computation possible according to the model, and in every state of such a computation, predicate ϕ holds.

[2] "Exists finally ϕ": there exists a model computation where finally a state satisfying ϕ is reached.

creates an internal representation of the DSL model. A mutation generator creates the mutants from this model and calculates their occurrence probability. Each mutant is expressed by an XML text representation conforming to the internal input format for UPPAAL models.

Our main contributions consist in the design of the DSL, the automated generation of the mutants and the calculation of their occurrence probability. Furthermore, our approach avoids the occurrence of state space explosions arising when all possible faulty behaviours are simultaneously considered in one probabilistic model (see further comments in Section 11.1.3). Finally, the different mutants can be analysed independently; therefore our analysis tool distributes the UPPAAL model checking tasks over several computers and CPU cores, so that model checking of different mutants can be performed simultaneously.

11.1.3 Related Work

Model-checking has been widely used for the verification of communication protocols and also for checking safety-properties of systems, see [40, 1, 89] and the references given there for related work in the railway domain. Reliability aspects have mostly been approached by means of probabilistic model checking, see, for example, [71, 39].

Our solution differs from the latter in that we deliberately do not use probabilistic model checking for these reliability aspects: extensive experiments performed by our group with the PRISM tool [39] showed that (1) the lack of real-time modelling capabilities enforces abstractions which either over-simplify the real communication behaviour or leads to unnecessarily complex constructions involving clock tick counters or similar devices, and (2) the incorporation of *all* possible faulty behaviours in one model lead to unacceptable checking times and even state explosions for the more sophisticated models. Indeed, since the probability that all possible faults occur while processing one transaction is so low that it can be neglected anyway, such a model would contain many computations of no practical relevance. Finally, (3) tools like PRISM only handle numeric probability values, but do not allow to investigate symbolic ones. As a consequence, parameter-dependent analyses require to re-run the time-consuming model checks for every parameter value to be considered.

Our approach tackles the combinatorial problem by checking many models instead of a single one and profit from the smaller size of each model: the complexity of evaluating one (probabilistic) model incorporating all possible faults is considerably higher than checking many simpler models, in particular, if the simpler models can be checked in parallel. Additionally, we calculate algebraic representations of occurrence probabilities. As a consequence, parameter-dependent analyses can be made by just inserting concrete probability values into the parameters of the formula.

11.1.4 Overview

In Section 11.2 we sketch the work flow supporting reliability analysis and the tool components involved. Section 11.3 introduces the DSL CAMoLa, our description formalism for communication architectures. In Section 11.4 the principles of mutation generation and the reliability calculation based on mutant evaluation are described. Section 11.5 contains a discussion of results and prospects for future work.

11.2 Workflow and Tool Chain

The reliability analysis workflow starts with modelling a communication architecture in the domain-specific *Communication Architecture Modelling Language (CAMoLa)*, using the informal commu-

nication architecture specification with associated protocol descriptions as input (Fig. 11.3). Next, CAMoLa's model-to-text generator transforms the CAMoLa model into an UPPAAL model, enriched with syntactic markers for the so-called *behaviour switches* which are part of the CAMoLa formalism and used to model possible deviations from normal behaviour (see Section 11.3 below). Now the mutation generator tool inserts *behaviour-vectors* (Section 11.3) into the UPPAAL model to create mutations with different message transmission behaviour. Intuitively speaking, each vector specifies which deviations from normal behaviour are applied to message sequences passing at specific locations in the model, and each model location where faulty behaviour is anticipated is associated with such a vector. The mutation generator records the algebraic formula for each mutation's occurrence probability in a table. Each formula is an arithmetic expression over the occurrence probability parameters associated with each fault type (see Table 11.1) possibly occurring in some part of the model when processing a message. Then the UPPAAL tool is activated to verify the reliability property on the mutation; this process is parallelised over several CPU cores and computers to increase performance. For each mutant, it is recorded in the table whether it shows reliable behaviour or leads to a transition into stable safe state.

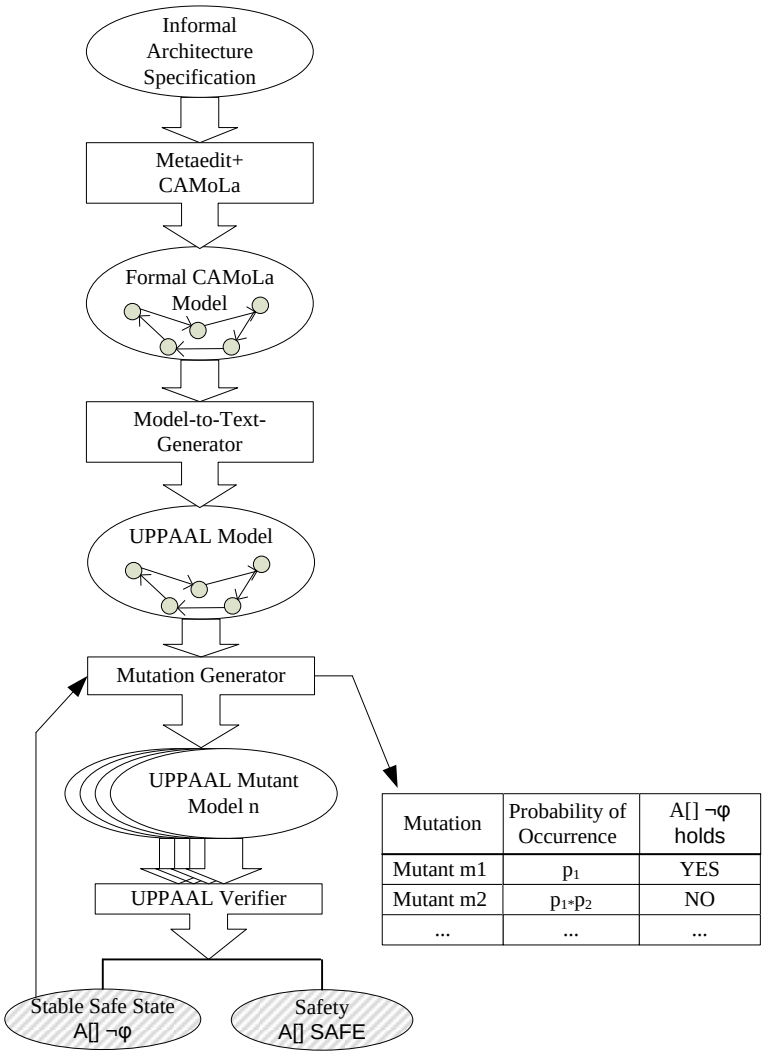

Figure 11.3: Workflow of the presented Framework.

11.3 The Communication Architecture Modelling Language CAMoLa

CAMoLa was designed for modelling communication architectures and associated protocol behaviours. Each model is derived from the informal specification of the architecture and consists of synchronised processes representing protocol components, transmission channels or additional components simulating environment behaviour or acting as observers in the verification process. CAMoLa and its model-to-text generator were designed with the tool Metaedit+ [72], which is a meta-modelling and modelling-workbench [66]. The DSL supports two hierarchical views on communication architectures: A view on all components with their interactions (Fig. 11.4) and a process view on each component behaviour in timed automata notation (Fig. 11.5).

CAMoLa extends the usual timed automata notation by the notion of *behaviour-switches* bs, representing controlled normal and exceptional behaviour transitions between locations (see Fig. 11.5). Each possible controlled transition is identified by a marker from set $o_{bs} = \{0, \ldots, n, stop\}$. The transition connected to one distinguished switch position (position 1 in Fig. 11.5) is associated with normal behaviour at this model location, so the error-free timed automata model can be extracted from the CAMoLa model by deleting at every behaviour switch all outgoing transitions but the one associated with normal behaviour. Each other switch position gives rise to a type of mutated behaviour.

In order to reflect the possibility of different types of transient errors occurring at a specific model location, mutant models are not simply generated from the CAMoLa model by fixing switch positions, but by associating each behaviour switch with *behaviour-vectors* v^d: if $o_{bs} = \{0, \ldots, n, stop\}$, then $v^d \in \{0, \ldots, n\}^d$, and it specifies that the first d messages $m_1, \ldots, m_d$ passing along the model location controlled by bs trigger transitions $v^d(1), \ldots, v^d(d) \in \{0, \ldots, n\}$, respectively (Fig. 11.6).

The semantics of this construction is defined by translating the CAMoLa process containing the pair bs, v^d into an ordinary timed automaton utilising an additional auxiliary variable j counting the number of messages passing along the behaviour switch, that is, the number of outgoing transitions of bs which have been triggered so far, and an auxiliary location l_{stop}: suppose that bs is located at source location l and that the switch controls outgoing transitions with identifiers $0, \ldots, n$, leading to target locations $l_0, \ldots, l_n$. Then the associated timed automaton has outgoing transitions

$$
\begin{aligned}
& l \xrightarrow{j<d \wedge v^d(j)=0 / j:=j+1;} l_0 \\
& l \xrightarrow{j<d \wedge v^d(j)=1 / j:=j+1;} l_1 \\
& \vdots \\
& l \xrightarrow{j<d \wedge v^d(j)=n / j:=j+1;} l_n \\
& l \xrightarrow{j \geq d} l_{stop}
\end{aligned}
$$

at location l (j is initialised to 0 when the automaton is initialised).

While the designers specify the behaviour-switches and model the possible deviations from normal behaviour, behaviour-vectors are generated automatically by the mutation generator (Section 11.4). In order to control this generation process, each behaviour-switch position carries an upper bound indicating up to how many times the transition can be taken. The bound can be taken from the set $\mathbb{N}_0 \cup \{*\}$ (in the sample state machine of Fig. 11.5 only 0 and $*$ are used): symbol $*$ indicates that the mutant generator can select this transition an unbounded number of times when generating behaviour-vectors; a bound $b_e \in \mathbb{N}_0$ associated with transition e constrains the behaviour-vector generation in such a way that e occurs at most b_e times in the vector: $num(v, e) \leq b_e$. A bound b_e reduces the

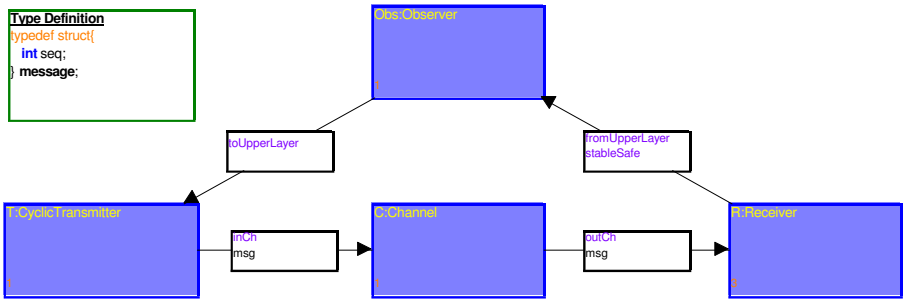

Figure 11.4: Simple Architecture, System View

amount of model-mutations but leads to an under-approximation in the reliability calculation (see further comments in Section 11.4).

Observe that all locations introduced on behalf of the behaviour-switch are urgent, since the switch is only a selector of normal or mutated behaviour, and does not consume processing time in the real world.

11.4 Mutation Generation and Reliability Calculation

Generation Concept.

Suppose we have created a CAMoLa model for each transaction type T_ℓ occurring in our communication architecture. The mutation generator creates concrete mutants as timed automata models, where all nondeterminism regarding fault occurrences has been eliminated. This is achieved by means of the behaviour-vectors: let $\{bs_1, \ldots, bs_k\}$ the behaviour-switches in the CAMoLa model associated with transaction type T_ℓ. Given a bound $max \in \mathbb{N}$, the mutation generator creates tuples of behaviour vectors $V = (v_1^d, \ldots, v_k^d)$, such that each behaviour switch bs_i is associated with one behaviour vector v_i^d of dimension d, and the following conditions are fulfilled: (1) $d \leq max$, (2) each vector component $v_i^d(j), i = 1, \ldots, k, j = 0, \ldots, d-1$ is in range $\{0, \ldots, n_i\}$, such that an outgoing transition with identifier $v_i^d(j)$ exists at behaviour switch bs_i, (3) the mutants $\mathcal{M}(V)$ associated with V satisfy $A\mathbf{G}(\neg\phi)$[3] (we call them *reliable mutants*), and, (4) reducing the dimension of any v_i^d by one will result in an unreliable mutant satisfying $E\mathbf{F}\phi$. Conditions (3) and (4) are checked by means of the UPPAAL model checker.

Calculation of Overall Reliability.

It is our objective to calculate an approximation of the communication architecture's expected reliability which is *conservative* in the sense that the real reliability is equal to or better than the calculated estimate. The calculations performed below are based on the assumptions that (1) no other faults occur in the communication system than the anticipated ones that have been represented in the CAMoLa model by means of behaviour-switches, (2) all faults occur in a stochastically independent manner, and, (3) the safety-related services are performed in a transaction-oriented manner as explained in Section 11.1.2, so that the outcome of transactions is again stochastically independent.

[3]Recall that ϕ denotes the property that the system is in stable safe state, that is, still safe, but no longer operable.

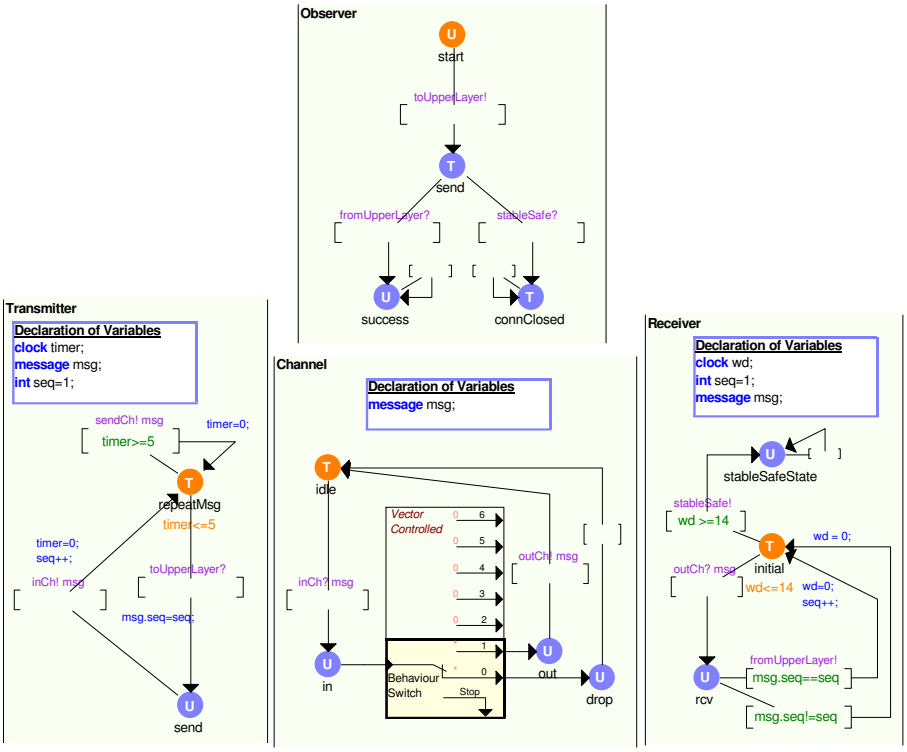

Figure 11.5: Simple Architecture, Process View

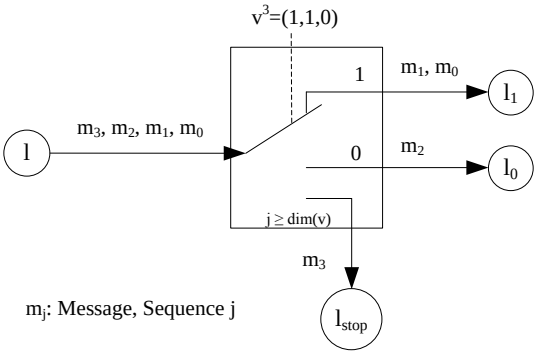

Figure 11.6: Vector controlled Behaviour-Switch

If these hypotheses are satisfied it is possible to approximate the reliable system operation $R(t_0, t_1)$ over a time period $[t_0, t_1]$ by means of the reliability of single transactions: suppose that R_{T_ℓ} is the probability that execution of transactions of type T_ℓ will not transit into stable safe state, but perform the specified service, and that different transaction types $T_\ell, \ell = 1, \ldots, q$ have to be considered. For each transaction type T_ℓ let $c_\ell^{max} \in \mathbb{N}$ the maximal number of T_ℓ-transactions which are possible per time interval $[t_0, t_1]$, and $\delta_\ell > 0$ the minimal duration of such a transaction. Then the overall reliability $R(t_0, t_1)$ can be approximated conservatively by

$$R(t_0, t_1) \geq \min\{\prod_{\ell=1}^{q} (R_{T_\ell})^{c_\ell} \mid 0 \leq c_\ell \leq c_\ell^{max} \wedge t_1 - t_0 \leq \sum_{\ell=1}^{q} c_\ell \cdot \delta_\ell \leq t_1 - t_0 + \varepsilon\}$$

for ε satisfying $0 \leq \varepsilon < \max\{\delta_\ell \mid 1 \leq \ell \leq q\}$. The right-hand side of the above formula represents the worst-case situation, where a maximal number of transactions is performed during time interval $[t_0, t_1]$, and the combination of transactions performed in this interval is technically still possible, but represents the least reliable combination which may occur. It remains to determine the reliability of each transaction type T_ℓ. To this end, we observe that the occurrence probability of a reliable mutant $\mathcal{M}(V)$ is

$$P_V = \prod_{i=1}^{k} \prod_{j=0}^{d-1} p^i_{v_i^d(j)}$$

where p_e^i denotes the occurrence probability of the basic fault (or normal behaviour) associated with outgoing transition number e at behaviour switch bs_i (so $\sum_{e=0}^{n_i} p_e^i = 1$ for all $i = 1, \ldots, k$). The probability that a transaction of type T_ℓ will terminate successfully without transition into stable safe state is

$$R_{T_\ell} = \sum_{\{V \mid \mathcal{M}(V) \models A\mathbf{G}(\neg\phi)\}} P_V + \sum_{\{\pi, V \mid \mathcal{M}(V) \models E\mathbf{F}\phi \wedge \pi \models \mathbf{G}(\neg\phi)\}} P_\pi \cdot P_V$$

where π denotes a computation of mutant $\mathcal{M}(V)$ and P_π the probability of $\pi\,'s$ occurrence: R_{T_ℓ} is the sum of all occurrence probabilities of reliable mutants plus the occurrence probabilities of paths in unreliable mutants leading to successful completion of the transaction. If we neglect the occurrence probability of reliable computations π in unreliable mutants and only consider reliable mutants whose behaviour vectors v are of dimension $\dim(v) <= max$, and each transition e emanating from a behaviour switch occurs at most $b_e \in \mathbb{N}_0$ times in v, this results in the conservative approximation

$$R_{T_\ell} \geq \sum_{\{V \mid \mathcal{M}(V) \models A\mathbf{G}(\neg\phi) \wedge \forall v, e: \dim(v) \leq max \wedge num(v,e) \leq b_e\}} P_V$$

The right-hand side of this inequation can be computed by the mutant generator in combination with the model checker.

Example.

As an example we demonstrate the calculation of the reliability of the example architecture in Fig. 11.4. This architecture consists of a transmitter, channel, receiver and observer, transmitter and receiver being allocated in the safety-layer. The observer performs a safety-related transaction which completes successfully in terminal state success if a message sent on channel toUpperLayer is finally received on channel fromUpperLayer (see Fig. 11.5). The transmitter sends messages in fixed cycles of 5 time units. It repeats a message with the same sequence number until a next message has

to be transmitted. The receiver removes duplicated messages indicated by identical sequence numbers. It also monitors the operability of the transmission channel: at least one message within 14 time units is expected (regardless of the sequence number). If no message is received within 14 time units, the receiver transits into stableSafeState, so we are interested in the probability that the complete system satisfies $A\mathbf{G}\neg$Receiver.stableSafeState, or, equivalently, $RQ \equiv A\mathbf{F}$Observer.success ("RQ" standing for *Reliability Query*). The communication channel includes a behaviour-switch bs_1 with the set of outgoing transitions identified by $\{0, 1\}$. The outgoing transition number 0 models the message-loss-error and transition 1 transmits the message correctly. The *-character in the behaviour switch denotes that the transition can be take arbitrarily many times, so there are no restrictions regarding the creation of behaviour-vectors for bs_1. The mutation generator generates the initial vectors $v^1_{1,1} = (1)$, $v^1_{1,2} = (0)$ and starts the model-checking processes to verify RQ on each mutation. The model mutation induced by $v^1_{1,1} = (1)$ satisfies RQ, but the mutation induced by $v^1_{1,2} = (0)$ violates RQ, because the mutant derived from $v^1_{1,2}$ will drop the first message and block as soon as the second message arrives. In the next generation step, the mutation generator extends all vectors which are not satisfying RQ by all possible outgoing transitions of the behaviour-switch – this results in $v^2_{1,1} = (0, 1)$, $v^2_{1,2} = (0, 0)$ – and resumes the verification process. The tool iterates until the dimension of the vectors have reached a predefined limit (in the example we set the limit to 4, because we know that RQ can never be satisfied in presence of more than 3 message-drops). In Fig. 11.7 the whole set of generated behaviour-vectors is shown, each inducing one mutant model.

All behaviour-vectors whose mutants satisfy RQ represent reliable computations of the communication architecture: each transmission where only fault-combinations still ensuring $A\mathbf{G}\neg$Receiver.stableSafeState occur is still reliable. The probability of transmitting a sequence of messages specified in a behaviour-vector is calculated due to the known probability for an error-type to occur. In our example there is a probability to drop (p_0) or to transmit ($1 - p_0$) a message. The probability that a sequence of controlled transitions occurs is the product of each transition probability in a behaviour-vector (e.g. $v^3_{1,1} = (0, 0, 1)$, probability of occurrence: $p(v^3_{1,1}) = p_0 \cdot p_0 \cdot (1 - p_0)$). We assume that all events are stochastically independent. Now the reliability of a communication model is the sum of all mutation-occurrence probabilities satisfying RQ. For the example system this results in the reliability formula $R_{Ex} = (1 - p_0) + p_0 \cdot (1 - p_0) + {p_0}^2 \cdot (1 - p_0)$ which can be reduced to $R_{Ex} = 1 - {p_0}^3$. □

11.5 Discussion and Future Work

The reliability analysis of communication architectures according to the concepts introduced in this article allows users to compare different architectural designs and fault-tolerance mechanisms of communication protocols in safety-related domains. Furthermore, the analysis results induce requirements on message error probabilities. These probabilities represent decision criteria whether specific transmission techniques like WLAN, IP-Networks or xDSL should be allowed or forbidden in safety-related communication architectures with high reliability requirements. We have successfully analysed the reliability of the safety protocol SAHARA over UDP [65], a proprietary session-layer over the HDLC (High-Level Data Link Control) protocol, PROFIsafe over PROFINET and PROFINET DCP (Basic Discovery and Configuration Protocol). The results of these analyses imply maximal error probabilities and properties like maximal latencies of transmission techniques which are still acceptable in presence of the high levels of overall reliability required. Additionally the knowledge about the communication behaviour in presence of errors and error combinations has led to improvements of protocol specifications. Due to the divide-and-conquer approach the availability of an array of computers and multiple CPU cores makes model checking feasible on a large amount of error com-

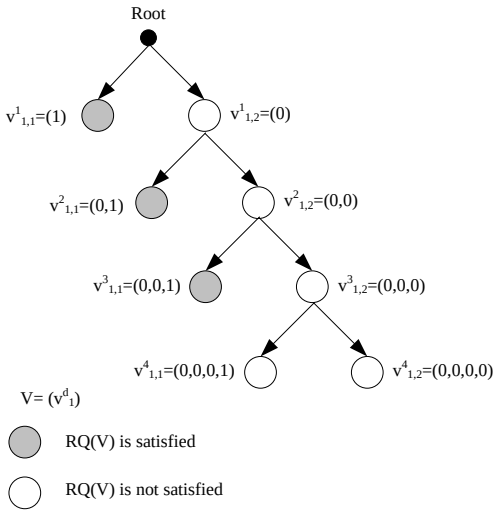

Figure 11.7: Generated Vector Tree

binations (i. e., mutants). We have successfully analysed an architecture with about 34 million error combinations which takes about 60 hours with an array of 25 computers (each 3 GHz).

In the future, we will analyse further architecture specifications, especially with reliable transport protocols like TCP and SCTP. Furthermore, we will improve the DSL CAMoLa for modelling communication architectures in a more generic way, such that pre-defined error behaviours applicable in specific communication domains can be re-used my means of building blocks from libraries. Additionally, it is planned to allow deviations from the transaction-oriented approach, in the sense that some system variables will be allowed to evolve across sequences of transactions. This will be helpful if, for example, fault counters are introduced in the system and incremented across transitions, so that shutdowns can be enforced if the fault rate is considered to be too high: in such a situation the success of a transaction also depends on the probability that the fault counter has reached its admissible limit before start of transaction.

Chapter 12

Case Studies

12.1 Cabin Smoke Detection in Airplanes

The smoke detection protocol – as layed out in Airbus specifications [3, 4, 5] – is a CAN-based protocol, where the individual smoke detection units are connected via short can bus segments in a *daisy-chain* layout. Start- and endpoint of the this chain is a controller unit (called CIDS), which collects configuration and system data and infers topology changes in case of failure (non-responding smoke detectors).

The case study encompasses high-level modeling of selected aspects thereof. In particular the parts that concern *scalability* (length of the chain) and *asynchronous operations* (smoke detection / can bus operations) are highlighted by this selection.

12.1.1 Goals of the Smoke Detection Case Study

The goals of this case-study are layed out as follows.

1. **Evaluate modeling power of our CDSL**

   The modeling power of the (drafted) contract domain-specific language (CDSL)–see section 4– shall be evaluated with respect to adequateness of the provided modeling facilities.

2. **Evaluate usability of the modeling elements**

   The adequateness of the CDSL language components (usability) with respect to the avionics domain shall be evaluated. The guiding principle here is that what is simple to comprehend should be simple to express in the model.

3. **Evaluate correspondence with GTL representation**

   The possibility of (automatic) translation of the CDSL to the GTL shall be assessed. For this, a manual translation of the model shall be constructed.

4. **Evaluate performance of instance concept (scalability)**

   Adequateness of the instance concept (many concrete components derived from one generic pattern) shall be evaluated. In particular the representation of *interfaces* between components shall be considered.

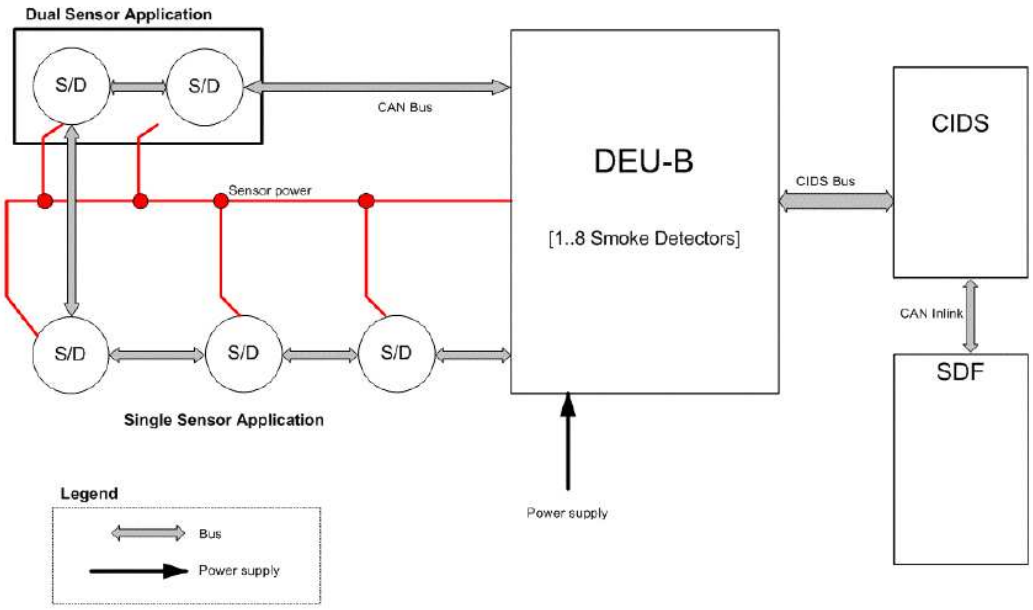

Figure 12.1: Example of A350xwb application architecture.

12.1.2 The A350xwb Smoke Detection Protocol

Selection of Smoke Detection Requirements

The requirements concerning the A350xwb smoke detection protocol fills several specification documents [3, 4, 5]. They encompass possible configurations of the physical layout, operation modes, power supply architecture, definition of the application layer of the CAN protocol, and controller unit software requirements.

Moreover, several smoke detection configuration modes are supported, like grouping some (or all) of the smoke detectors ("Dual Sensor Application") or having fire extinguishing data converters (FEDCs) as part of the layout. Figure 12.1 gives an example of a possible application architecture. It also shows that the smoke detection facility (SDF) is not directly connected to the smoke detector CAN bus segments, but relays communication via two other hardware units named CIDS and DEU-B.

Not all of this is relevant for us: we are interested in sample requirements that can be subject to (automated) verification in the GALS system architecture.

Therefore we focus on the following properties that can be considered essential with respect to the unique characteristics of this protocol:

- Controller and smoke detectors access a number of CAN bus segments. Each segment can only carry (undirected) payload that is available to both connected units. The CAN priority mechanism resolves conflicts, if both parties attempt write-access to the CAN bus segment.

- Data flow follows both directions of the CAN chain. The protocol definition has to ensure that no relevant information is "lost" by the CAN priority mechanism.

- There are two basic operational modes:

  (A) Ident Request Mode

  Here the controller asks every smoke detector unit to report identity and health status by broadcasting a message.

  (B) Status Polling Mode

  Here the controller sends addressed requests (uni-cast) for the smoke status of every smoke detector in a round-robin fashion.

- An Ident Request initiated by the controller reaches the controller again, when the end of the chain is reached.

  CAN bus segment failures are detected during the Ident request mode by either

  (A) acknowledgment timeout at smoke detector N while forwarding the request to smoke detector $N + 1$; this timeout generates a (downstream) reply to the controller

  (B) Ident Request Timeout at the controller (missing reply from both directions)

Selection of verification goals (or user requirements).

REQ-SD-01 If no CAN bus segment failure occurs, then an initiated upstream Ident request reaches the controller again before timeout.

REQ-SD-02 If a CAN bus segment failure occurs, this is properly identified during Ident Request, i.e., the controller reaches a state appropriate to the location of the failure.

REQ-SD-03 Smoke alarms that are detected in smoke detectors which have an operative chain of CAN bus connections to the controller will reach the controller within an appropriate time limit.

The Smoke Detection Model Abstraction

It is conceivable that a full-blown model of the complex A350xwb smoke detection facility exceeds the scope of this case study.

In the following we will strip away parts of the specification, while preserving the essence of the protocol. The guiding principle is that the user requirements layed out in section 12.1.2 must still be meaningful.

Component and configuration simplifications.

1. DEU-B, CIDS, and SDF are abstracted into one *controller* unit which we refer to as "CIDS" in the model.

2. Dual sensor application is omitted, every smoke detector is single sensor.

3. Fire extinguishing data converter (FEDCs) are omitted.

Data simplifications.

1. The *payload* of the CAN data protocol is omitted; the model preserves only the CAN identifier data, which suffices to
   - communicate the role of the message in the protocol (identification request, identification reply, acknowledgment, smoke warning)
   - determine the message priority
   - read out addressing information (broadcast/uni-cast)
2. Smoke warning is reduced to one bit of information ("smoke detected or not").

Each of this simplifications may be dropped later on in order to add complexity to the smoke detection example.

Overview on the Resulting Model. The resulting composite structure of the model is displayed in Figure 12.2.

The constants used for this model are listed in `<<enumeration>>` containers.

- `CAN_MSG_IDENT_CONSTS` (data present on CAN bus segments)
- `TIMING_CONSTANTS` (for timing configuration of the model behavior)

The dynamic parts of the models are captured via class diagrams, which all are linked to state machines to explain the dynamic behavior.

- CAN (CAN bus segments; the prioritization of incoming data is modeled as behavior)
- CIDS (the controller component)
- `SMOKE_DETECTOR` (receiving smoke warning, reading/writing to CAN in both directions)
- `SMOKE_SENSOR` (reporting the measured smoke status)

The stereotype `<<interface>>` is used for data streams.

- `UP_IN` (data flow: upstream, as seen from one CAN bus segment)
- `DOWN_IN` (data flow: downstream, as seen from one CAN bus segment)
- `OUT` (the actually visible value on the CAN bus, after prioritization)
- `SMOKE_WARNING` (to communicate the information "does it smoke?")

A detailled description follows below on page 108.

GALS-Model of the A350xwb Smoke Detection Protocol

This Section describes the GALS system of systems model developed as part of the A350xwb smoke detection protocol case study. We use the Enterprise Architect tool as sketched in section 4.4 to graphically represent the system under consideration.

Functionality and structure is abstracted to a level suitable for the generation of contracts arguing over each locally synchronous system. As such, it will be used as the basis for the automated derivation of contracts in later phases.

The following elaborates on model layout sketched in section 12.1.2. The individual components are described in detail.

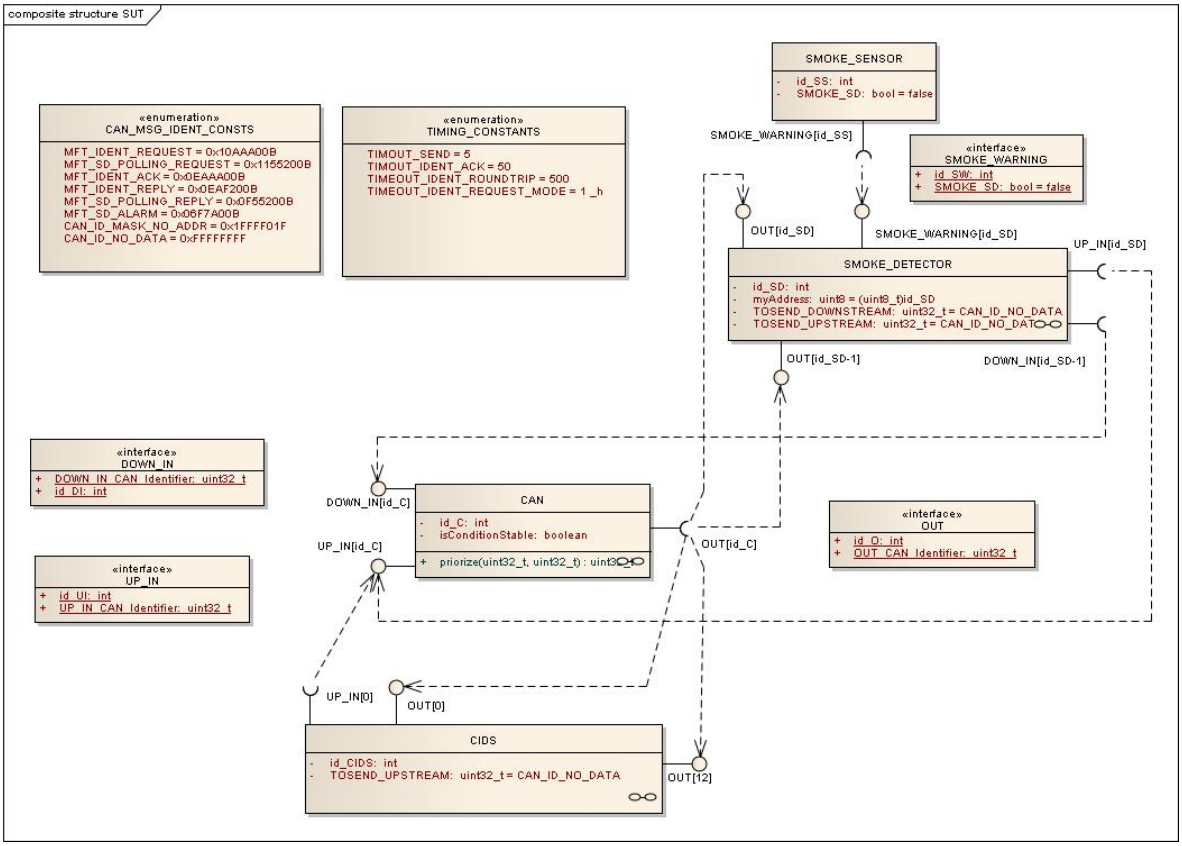

Figure 12.2: Composite structure — System under test

Asynchronous components

Figure 12.2 shows the composite structure diagram used to decompose the globally asynchronous system of systems into its subsystems and interfaces needed to communicate.

The class **CAN** represents (the behavior of) an individual CAN bus segment within the ring bus topology. Each instance is parametrized using the unique identifier `id_C`. Since each CAN segment can be written to from both its ends, the two interfaces **DOWN_IN** and **UP_IN** are provided for upstream and downstream assignments respectively. These are again parametrized using the unique identifiers `id_D` and `id_U` respectively. To allow other systems to write onto a CAN bus segment, the class **CAN** provides these interfaces. By convention, each class instance **CAN[id_C]** will provide interface instances **DOWN_IN[id_D = id_C]** and **UP_IN[id_U = id_C]** respectively. Conversely, systems can read a CAN bus segment by accessing an instance of interface **OUT**, which is parametrized using identifier `id_O`. Therefore, each class instance **CAN** requires interface instance **OUT[id_O = id_C]**.

The class **CIDS** represents the cabin intercommunication and data system, the controller responsible for smoke detection. Since it writes to CAN segment 0 (upstream), it requires interface instance **UP_IN[id_U = 0]**. For reading upstream and downstream CAN segments, interface instances **OUT[id_O = 0]** and **OUT[id_O = n]** are provided, respectively. Note that **n** denotes the number of smoke detector instances in this context.

The class **SMOKE_DETECTOR**, combined with the class **SMOKE_SENSOR**, represents an

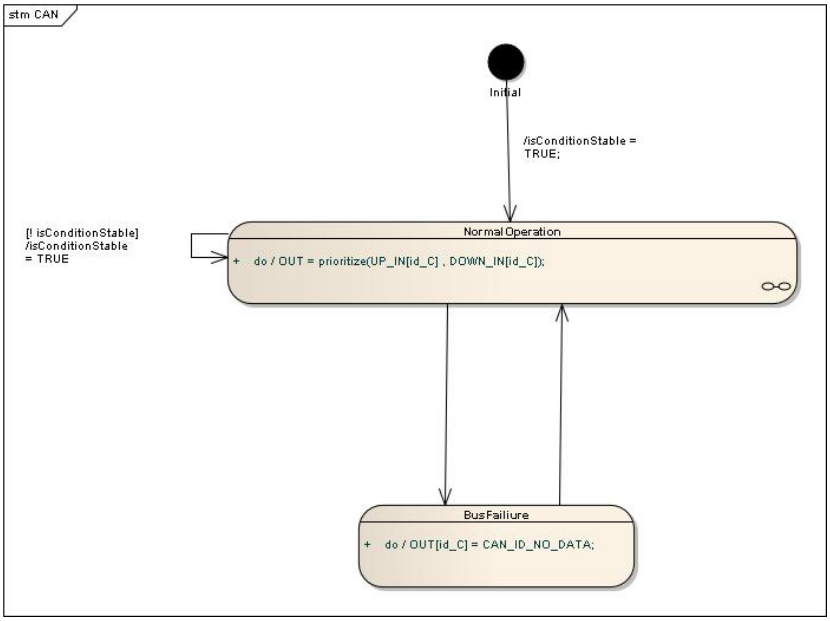

Figure 12.3: State chart — CAN bus

individual smoke detector as integrated into the airplane cabins. It is parametrized using identifier `id_SD`. When sending data upstream, each instance requires interface instance **UP_IN[id_U = id_SD]**. Sending downstream is accomplished via interface instance **DOWN_IN[id_D = id_SD −1]**. The interface **SMOKE_WARNING** is used for 1-on-1 communication between smoke detectors and their respective smoke sensors.

Synchronous Component — CAN bus

A synchronous component **CAN** distinguishes two separate control states, state *Normal Operation* and state *BusFailure*. This is depicted in Figure 12.3. The system can transition between these two control states non-deterministically to reflect the fact, that the bus system may fail (or resume operation) at any time.

Within state *BusFailure* no data can be read from the bus, so the constant CAN identifier `CAN_ID_NO_DATA` is output to the appropriate interface instance.

Within state *NormalOperation* the CAN bus prioritizes its output according to the CAN message identifiers read from upstream and downstream input interfaces. This is realized in the form of the decision tree depicted in figure 12.4.

Whenever the downstream input CAN message identifier read from the corresponding interface has a higher priority than the available upstream input, the downstream input is relayed to the output interface. Otherwise, the upstream input is relayed. The auxiliary attribute `isConditionStable` is used to reevaluate the decision tree whenever changed inputs demand this.

Synchronous Component — CIDS

The **CIDS** subsystem is decomposed into two functions **CIDS_LOGIC** and **UPSTREAM** running in parallel. Figure 12.5 shows the corresponding composite structure diagram. Function **CIDS_LOGIC** realizes the Ident request protocol as intended for the CIDS-side of the protocol. Within its definition,

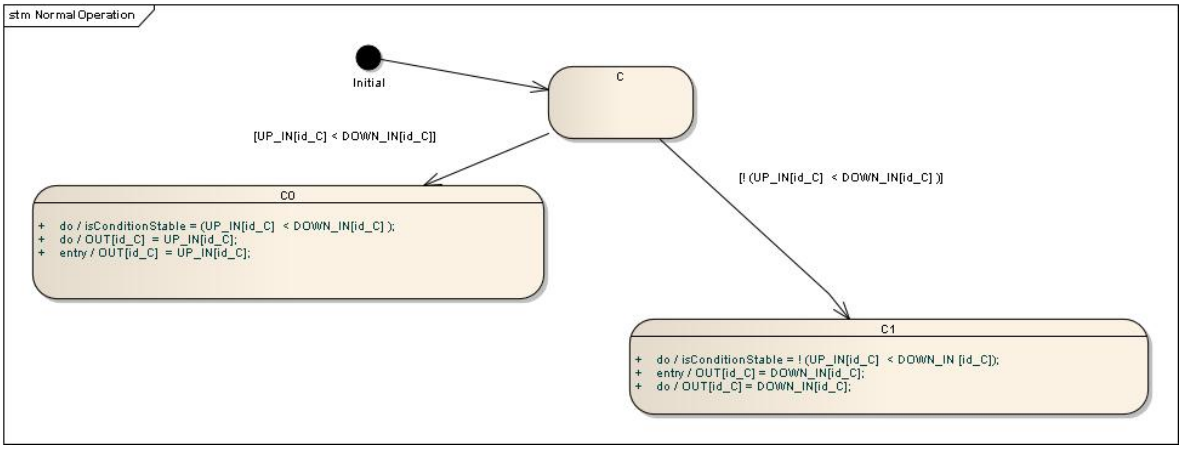

Figure 12.4: State chart — CAN bus normal operation

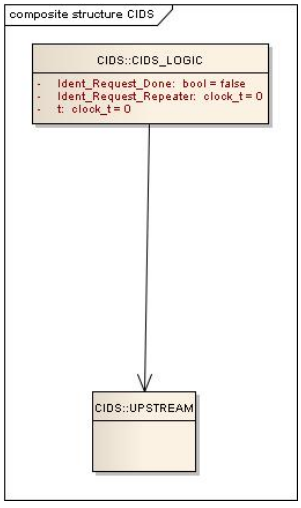

Figure 12.5: Composite structure — CIDS

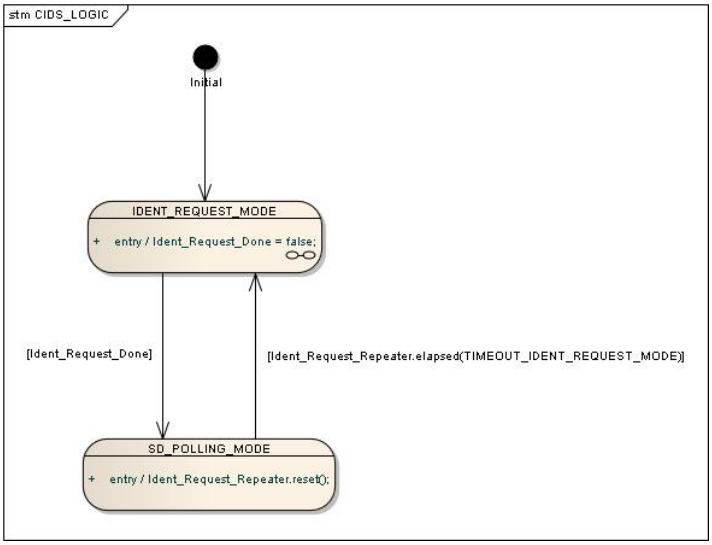

Figure 12.6: State chart — CIDS Logic

it determines what CAN messages to send in its upstream direction, and places the corresponding CAN message identifier in attribute `TOSEND_UPSTREAM`. Function **UPSTREAM** is then responsible for placing the corresponding CAN message on the upstream CAN bus and supervising its delivery.

Figure 12.6 shows the state chart for CIDS function **CIDS_LOGIC**. This state chart defines the protocol logic for the Ident request functionality. Initially, CIDS has to send an Ident request upstream, and does so in control state *TRANSMIT_IDENT_REQUEST*. After that, an acknowledgment is expected from the immediate upstream neighbor smoke detector. If this is not received after a timeout, the system transitions into state *CANFAIL_NOACK*, and the in-operational first CAN segment can be inferred. A received acknowledgment lets the system transition into state *ACK_RECEIVED*.

In state *ACK_RECEIVED* CIDS expects to receive its own Ident request from the immediate downstream neighbor smoke detector to indicate a complete round trip and function bus topology. In the event of a bus failure between two smoke detectors (or between the last smoke detector and CIDS) an Ident reply is received from the upstream smoke detector, which contains the address of the last smoke detector to have received the original Ident request. This again lets the CIDS infer the position of the inoperative CAN bus segment in question, and the system transitions into state *CANFAIL_REPLY_RECEIVED*. If neither message arrives within a given timeout, the protocol has failed, and no reliable diagnosis is possible.

Synchronous Component — Smoke Detector

The **SMOKE_DETECTOR** subsystem is decomposed into four functions **SD_LOGIC**, **UPSTREAM_SD**, **DOWNSTREAM_SD**, and **Ident_Request_Watchdog**. Figure 12.9 shows the corresponding composite structure diagram. Function **SD_LOGIC** realizes the fundamental smoke detector behavior, which is complying with the upstream/downstream protocol. According to the direction of the input, either **UPSTREAM_SD** or **DOWNSTREAM_SD** is active, organizing the respective data flow direction. The state machine **Ident_Request_Watchdog** is used to measure timely acknowledgment of Ident requests that are forwarded upstream.

Figure 12.10 shows the state chart for smoke detector function **SD_LOGIC**. Since CAN bus access is prioritized, only one data flow direction can be active at any one time. The data flow with

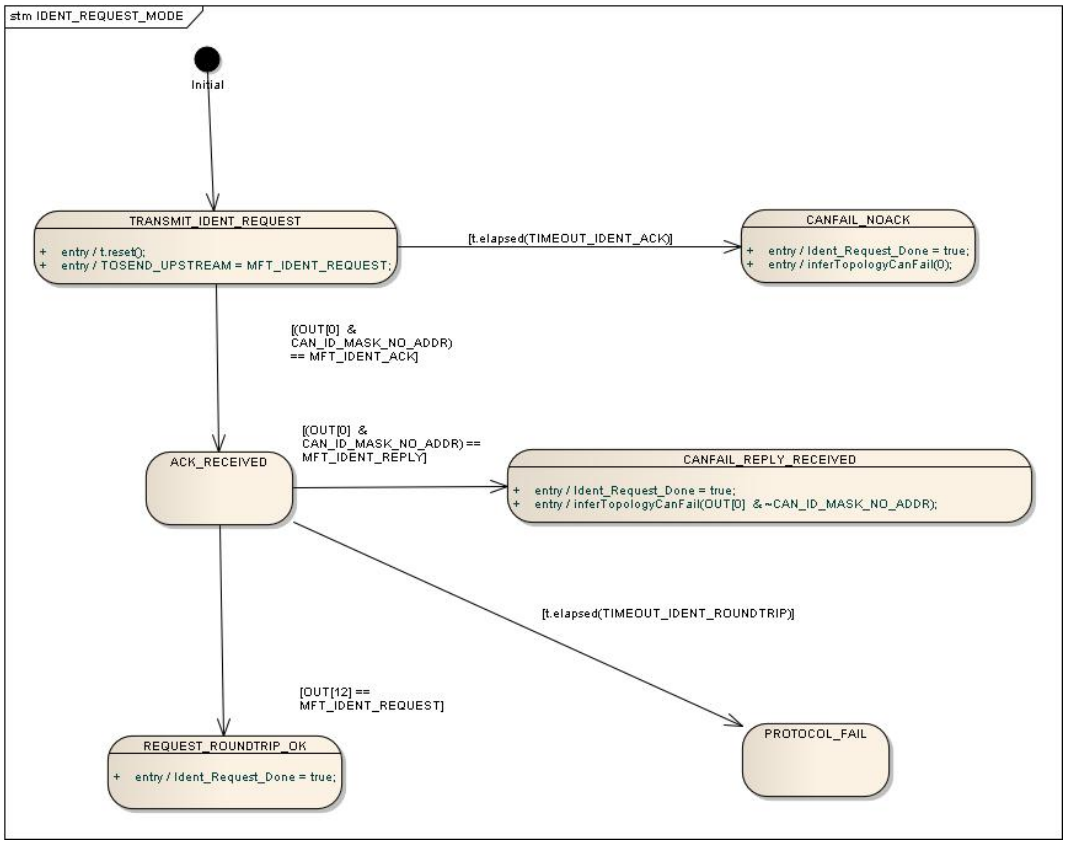

Figure 12.7: State chart — Ident request mode

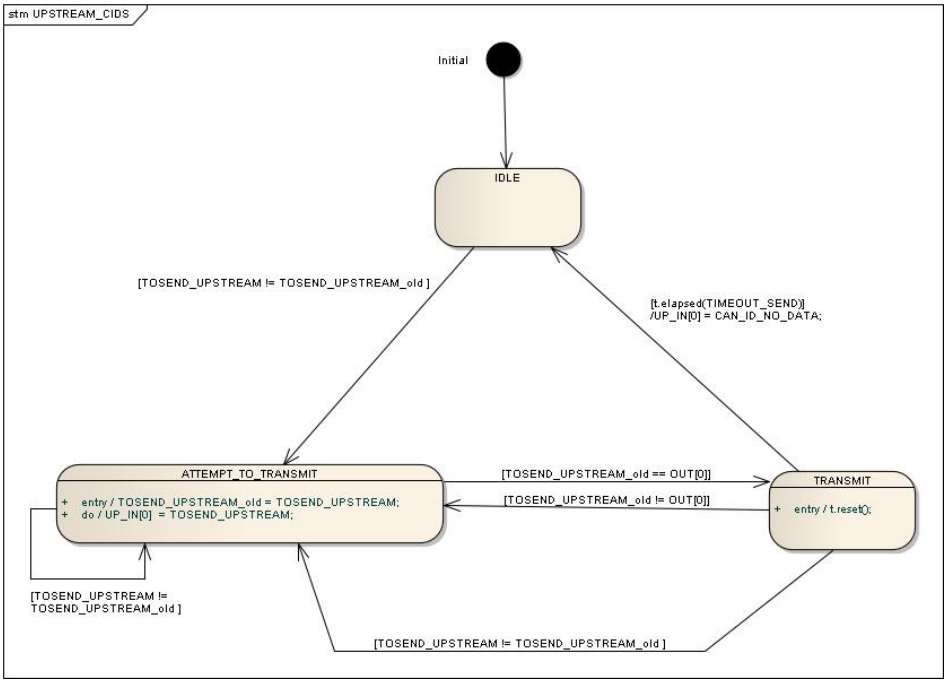

Figure 12.8: State chart — CIDS Upstream

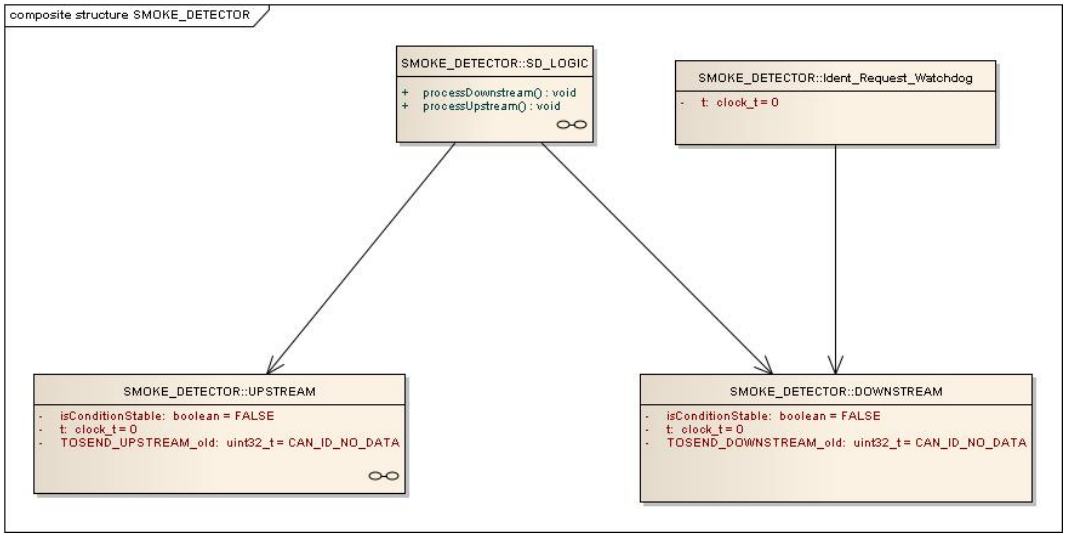

Figure 12.9: Composite structure — Smoke detector

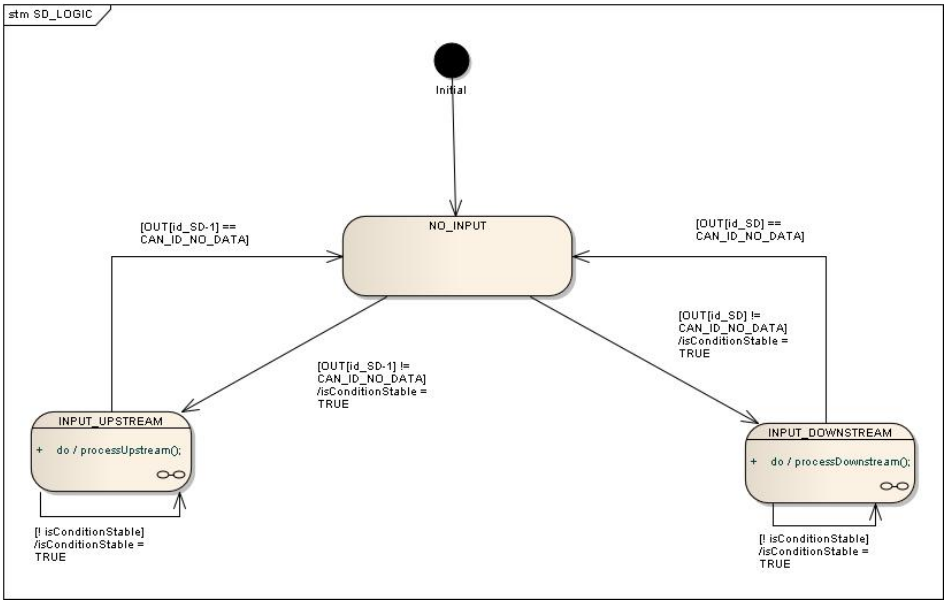

Figure 12.10: State chart — SD Logic

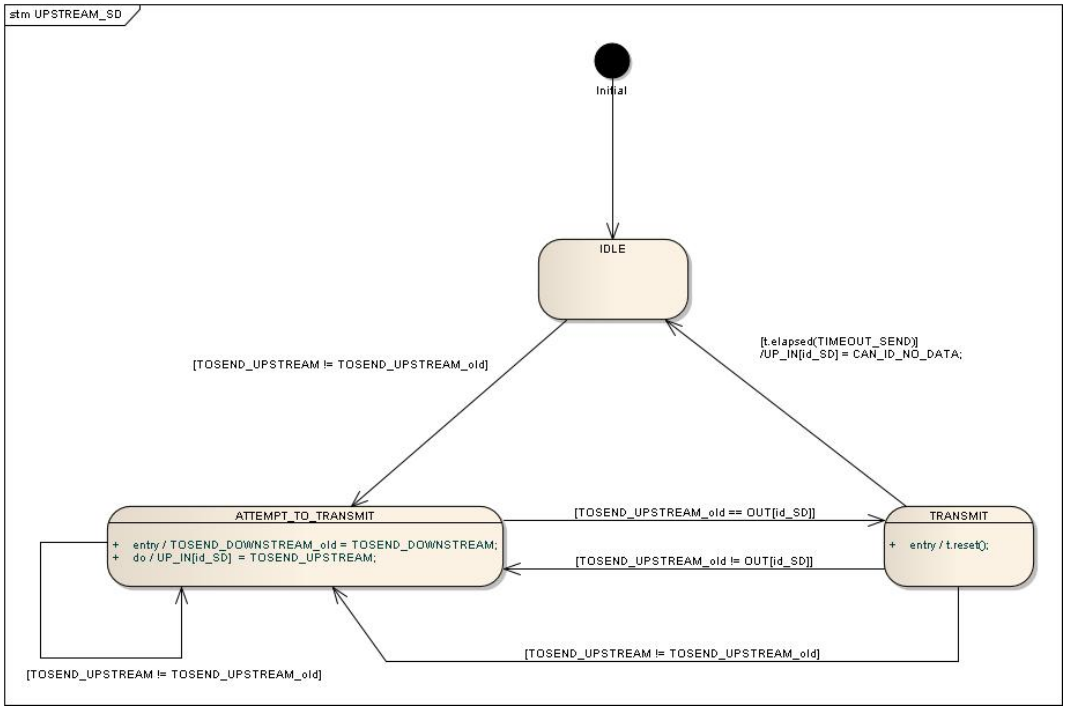

Figure 12.11: State chart — SD Upstream

lower priority is necessarily lost. Depending on the direction of the input, this state chart splits up into the sub-behavior corresponding to the currently active data flow direction (upstream or downstream). Smoke alarms are processed in the upstream context: If status polling is received in this data flow direction, then the (downstream) reply contains the smoke alarm information.

Figures 12.11 and 12.12 show the state charts **UPSTREAM_SD** and **DOWNSTREAM_SD** relevant for putting the intended data on the CAN bus, They follow the same systematic like `Upstream_CIDS`, compare Figure 12.8 (page 113).

Figure 12.13 shows the state chart for the smoke detector function **Ident_Request_Watchdog**. This is part of the protocol behavior connected to the Ident request mode, where smoke detector identity requests are forwarded upstream until they reach CIDS again at the end of the chain (in the nominal case). Forwarded Ident requests have to be *acknowledged* by the next smoke detector further down the chain, in order to determine sanity of the connection. The `MFT_IDENT_REPLY` is the error message to be reported back downstream, if the acknowledged does not arrive in time.

Data stream processing via Decision Trees.

The “do”-actions `processUpstream()` and `processDownstream()` in Figure 12.10 are in fact state machines, that process the data processing of CAN information received in directions upstream and downstream respectively.

The state charts in Figure 12.14 and Figure 12.15 give decision-tree style definitions of this operations. Note that the leaf-nodes of the trees essentially assign output variables and enforce re-evaluation, if input changes (`isConditionStable` becomes `false`).

This is discussed later in more detail, see Section 12.1.3.

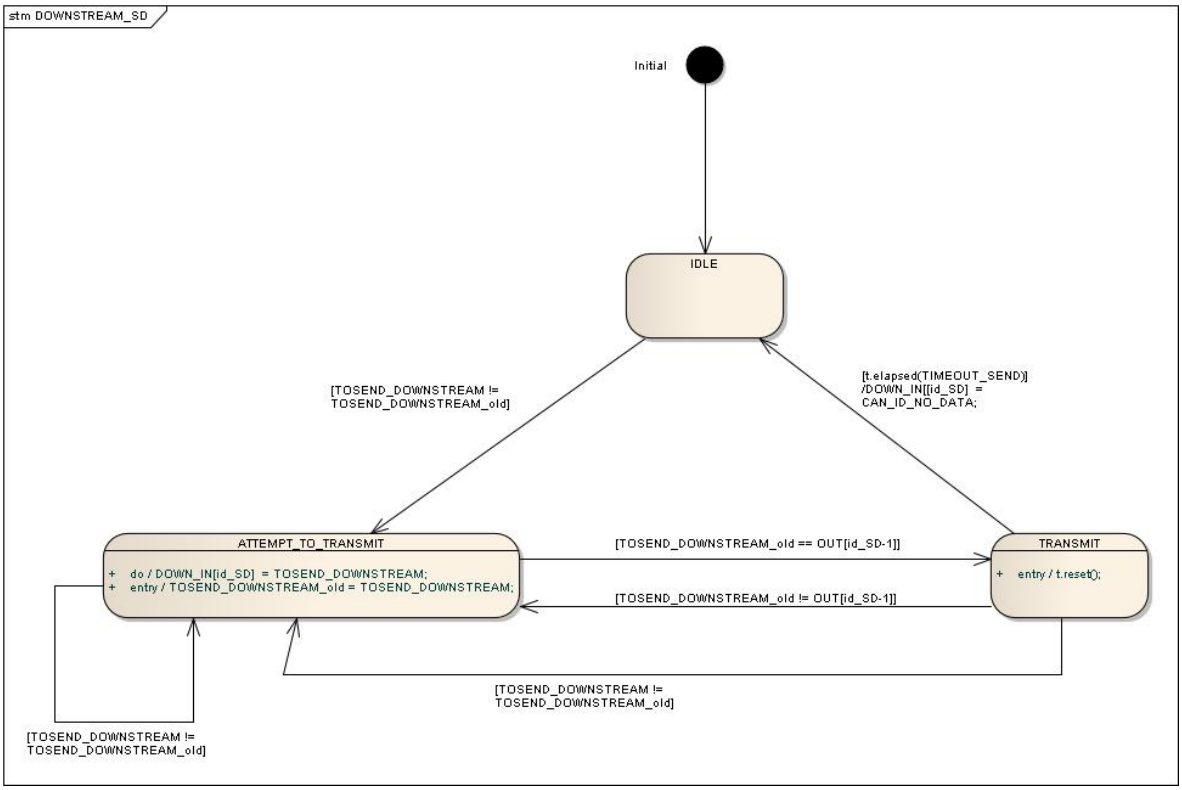

Figure 12.12: State chart — SD Downstream

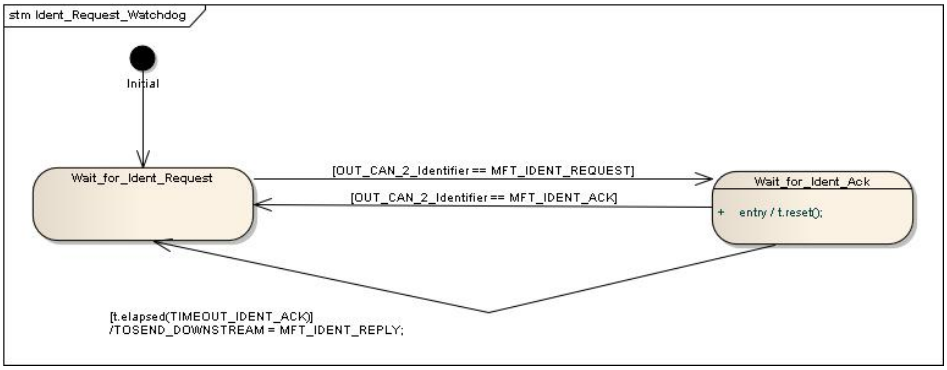

Figure 12.13: State chart — Ident request watchdog

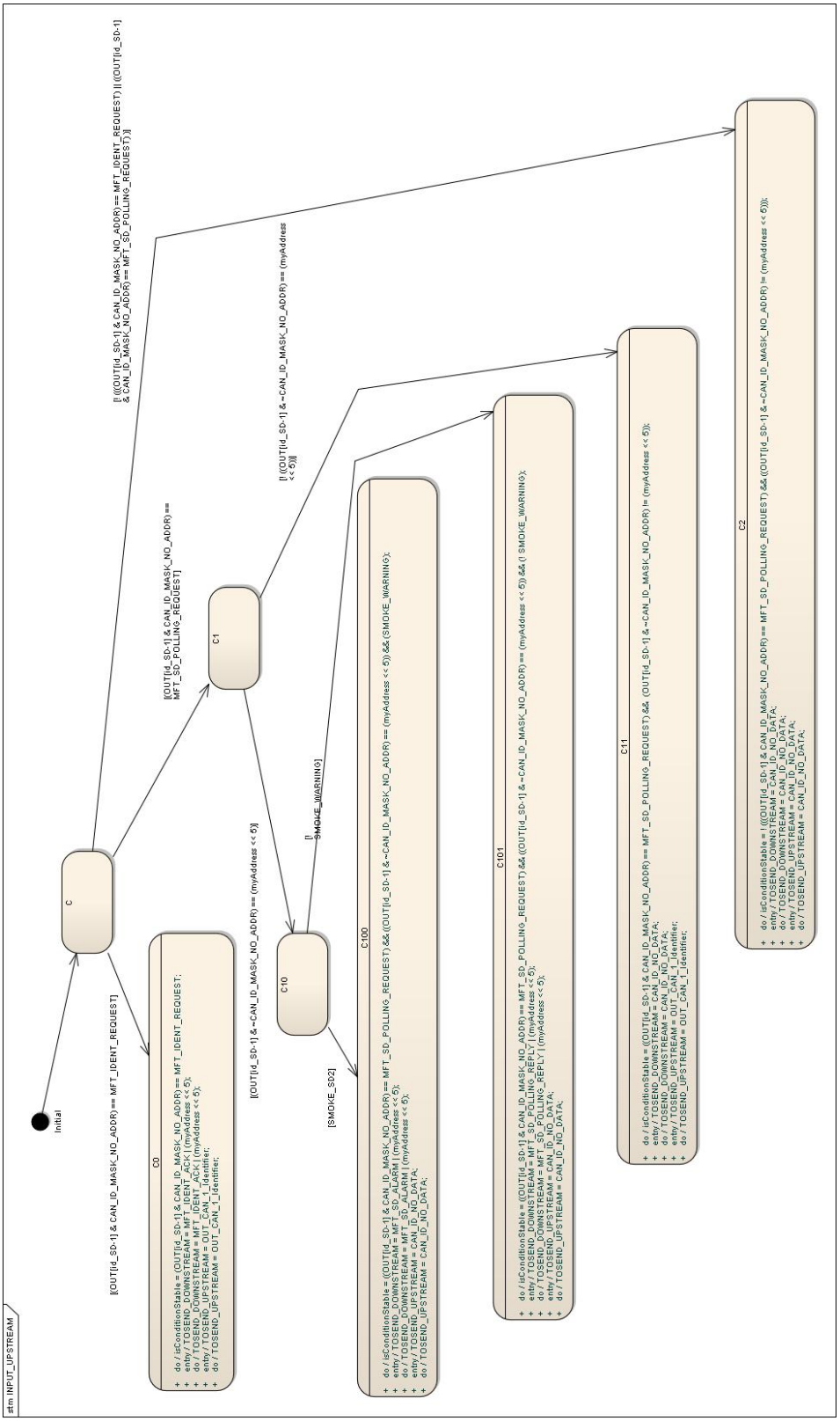

Figure 12.14: State chart — Input Upstream

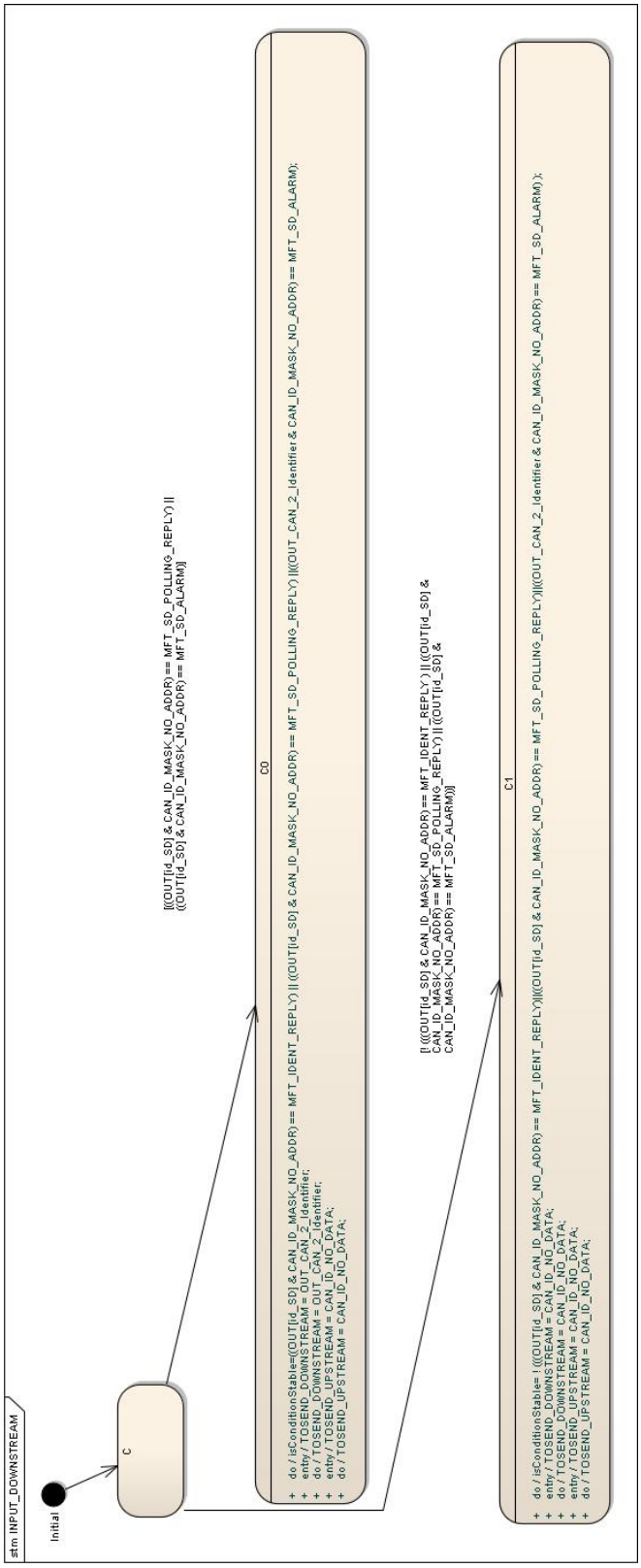

Figure 12.15: State chart — Input Downstream

12.1.3 Analysis of the Smoke Detection Model

This section contains the analytic part of the case study, where the details are explained and elaborated.

Instance Concept Realization

The possibility to derive a large number of similar components (instances) from one template (class) is an important feature of the domain specific language. This is illustrated in the smoke detection example, where it is desirable to model the contract network without committing to the size of the smoke detector chain a priori. Instead, we want to assume a parameter N and formulate N smoke detectors connected via $N + 1$ CAN bus segments to one CIDS. Every smoke detector is connected to a smoke sensor, so we have N smoke sensors.

For the modeling formalism, every class naturally offers the possibility to construct several instances. The interesting question is how to identify and connect them properly.

(1) Every class comes equipped with a class variable named `id_<SHORT_CLASSNAME>` which identifies the instance; (i.e. for the first instance it will have value 1, for the second instance value 2, and so on).

(2) Every interface is essentially a data container and comes *also* equipped with a property variable named `id_<SHORT_INTERFACENAME>`.

(3) The *connection* of data streams is identified by the relative index. Smoke detector `SMOKE_DETECTOR[id]` accesses the CAN input from upstream via interface `CAN[id]` and the interface from downstream via `CAN[id-1]`.

(4) A separate *instance table* defines the range of instances for each class.

In Figure 12.2, there is but one place where the N explicitly shows, namely the index of the right (upstream) input interface connection "`OUT[12]`". The instance table for N=12 is displayed in Figure 12.16.

Both the model and the instance table are inputs to the transformation mechanism to the GTL.

More details can be found in the CDSL description.

Discussion: Decision Trees vs. Decision Tables

Part of the operations in the model concern transformation of input data to output data.

More precisely, data arriving in upstream direction can be forwarded to the next upstream smoke-detector (via CAN) and at the same time trigger a response in the downstream direction.

Data arriving in downstream direction is plainly forwarded.

```
1  void processUpstream(OUT_CAN_1_Identifier, TOSEND_UPSTREAM, TOSEND_DOWNSTREAM){
2    switch(OUT_CAN_1_Identifier & CAN_ID_MASK_NO_ADDR) {
3    case MFT_IDENT_REQUEST:
4      TOSEND_UPSTREAM = OUT_CAN_1_Identifier;
5      TOSEND_DOWNSTREAM = MFT_IDENT_ACK | (myAddress << 5);
6      break;
7    case MFT_SD_POLLING_REQUEST:
8      if((OUT_CAN_1_Identifier & ~CAN_ID_MASK_NO_ADDR) == (myAddress << 5)){
9        TOSEND_UPSTREAM = CAN_ID_NO_DATA;
10       if(SMOKE_SD2){
11           TOSEND_DOWNSTREAM = MFT_SD_ALARM | (myAddress << 5);
```

INSTANCES OF	SMOKE_DETECTOR	
id_SD	1	12
INSTANCES OF	SMOKE_SENSOR	
id_SS	1	12
INSTANCES OF	SMOKE_WARNING	
id_SW	1	12
INSTANCES OF	DOWN_IN	
id_DI	0	12
INSTANCES OF	UP_IN	
id_UI	0	12
INSTANCES OF	CAN	
id_C	0	12
INSTANCES OF	OUT	
id_O	0	12
INSTANCES OF	CIDS	
id_CIDS	0	0

Figure 12.16: Instance Table for Smoke Detection Model

```
12          }
13          else{
14              TOSEND_DOWNSTREAM = MFT_SD_POLLING_REPLY | (myAddress << 5);
15          }
16        }
17        else{
18          TOSEND_UPSTREAM = OUT_CAN_1_Identifier;
19          TOSEND_DOWNSTREAM = CAN_ID_NO_DATA;
20        }
21        break;
22      default:
23        TOSEND_UPSTREAM = CAN_ID_NO_DATA;
24        TOSEND_DOWNSTREAM = CAN_ID_NO_DATA;
25        break;
```

Listing 12.1: Operation processUpstream()

```
1  void processDownstream(OUT_CAN_2_Identifier, TOSEND_UPSTREAM, TOSEND_DOWNSTREAM){
2    switch(OUT_CAN_2_Identifier & CAN_ID_MASK_NO_ADDR) {
3    case MFT_IDENT_REPLY:
4    case MFT_SD_POLLING_REPLY;
5    case MFT_SD_ALARM;
6      TOSEND_UPSTREAM = CAN_ID_NO_DATA;
7      TOSEND_DOWNSTREAM = OUT_CAN_2_Identifier;
8      break;
9    default:
10     TOSEND_UPSTREAM = CAN_ID_NO_DATA;
11     TOSEND_DOWNSTREAM = CAN_ID_NO_DATA;
12     break;
13   }
14 }
```

`OUT[id_SD-1] & CAN_ID_MASK_NO_ADDR == MFT_IDENT_REQUEST`	Y	N	N	N	N
`OUT[id_SD-1] & CAN_ID_MASK_NO_ADDR == MFT_SD_POLLING_REQUEST`	*	Y	Y	Y	N
`(OUT[id_SD-1] & ~CAN_ID_MASK_NO_ADDR) == (myAddress << 5)`	*	Y	Y	N	*
`SMOKE_WARNING[id_SD]`	*	N	Y	*	*
`TOSEND_UPSTREAM = OUT[id_SD-1];`	x			x	
`TOSEND_UPSTREAM = CAN_ID_NO_DATA;`		x	x		x
`TOSEND_DOWNSTREAM = MFT_IDENT_ACK | (myAddress << 5);`	x				
`TOSEND_DOWNSTREAM = MFT_SD_ALARM | (myAddress << 5);`		x			
`TOSEND_DOWNSTREAM = MFT_SD_POLLING_REPLY | (myAddress << 5);`			x		
`TOSEND_DOWNSTREAM = CAN_ID_NO_DATA;`				x	x

Figure 12.17: Decision Table for processUpstream()

Listing 12.2: Operation processDownstream()

This can be expressed with C-style code, see Listing 12.1 (Upstream) and Listing 12.2 (Downstream).

Note that it is clearly not desirable to allow arbitrary C/C++ code snippets in the CDSL. This would make the model semantic so powerful that it is doubtful that a translation to GTL is still possible.

Formulation as Decision Tree. In the CDSL, one straightforward choice of representation is a state machine in the form of a *decision tree*, where every leaf node corresponds to the final assignment. Since the operation is a (do/entry) action associated with one state, the decision-tree-style state machine is a sub-state, the leaf node has to be left if one of the input changes. In the model, the auxiliary Boolean variable "`isConditionStable`" is used to enforce re-traversal of the tree, if an input changes. Note that the parent state-machine `SD_Logic` (Figure 12.10) enforces re-entry of the sub-states, if `isConditionStable` evaluates to `false`.

For `processUpstream()`, Figure 12.14 displays the corresponding decision tree.

For `processDownstream()`, Figure 12.15 displays the corresponding decision tree.

Formulation as Decision Table. An alternative formulation of operations is the *decision table*. There are several competing definitions of decision table. We follow the suggestion from [84], i.e., use the upper half as conditions and the lower half as actions.

The upper right-hand part (condition entry) is filled with "Y"es for *true*, "N"o for *false*, and "*" for don't-care; a set of conditions hold, if all condition entries in a column are either "*" or match the truth value ("Y"/"N") of the condition stub in their row.

Since our actions are always *assignments* to model variables, we use C-style assignment notation and mark a column with an "x", if the assignment shall be executed (under the matching set of conditions). Moreover, we use the same condition parts for the assignments of both output variables (instead of splitting up into one decision table for each output variable). The separation of the two action parts it indicated by horizontal double-bar (=).

For `processUpstream()`, Figure 12.17 displays the corresponding decision table.

For `processDownstream()`, Figure 12.18 displays the corresponding decision table.

`OUT[id_SD] & CAN_ID_MASK_NO_ADDR == MFT_IDENT_REPLY`	Y	N	N	N
`OUT[id_SD] & CAN_ID_MASK_NO_ADDR == MFT_SD_POLLING_REPLY`	N	Y	N	N
`OUT[id_SD] & CAN_ID_MASK_NO_ADDR == MFT_SD_ALARM`	N	N	Y	N
`OUT[id_SD] & CAN_ID_MASK_NO_ADDR == MFT_SD_ALARM`	N	N	N	N
`TOSEND_UPSTREAM = CAN_ID_NO_DATA`	x	x	x	x
`TOSEND_DOWNSTREAM = OUT[id_SD];`	x	x	x	
`TOSEND_DOWNSTREAM = CAN_ID_NO_DATA;`				x

Figure 12.18: Decision Table for processDownstream()

Comparison. The following collects positive (⊕) and negative (⊖) aspects of each formalism, as elaborated for the two examples `processUpstream()` and `processDownstream()`.

Pro and Contra points for the decision tree formulation:

- ⊕ No extra model element required
- ⊕ Systematic structure
- ⊖ Fairly big for even a moderate example
- ⊖ Several sub-expressions get repeated often (difficult to keep changes consistent)
- ⊖ Difficult to read for examples with long identifiers/operations
- ⊖ Introduces dependencies across two levels of the state machine hierarchy (variable `isConditionStable`)
- ⊖ Prone to modeling mistakes (e.g., forgetting one assignment, negating one condition, etc.)

Pro and Contra points for the decision table formulation:

- ⊕ Widely understood notation
- ⊕ Compact and precise; virtually no duplication
- ⊕ Easily verifiable (“have I written what I thought?”)
- ⊖ Requires introduction of new model element (stereotype)

In conclusion, the decision trees do not seem adequate for modeling. The decision table notation is far more compact and readable in this context. Thus, it should be included in the CDSL.

Model Transformation to GTL

The transformation follows the principles layed out in section 4.5. For the Smoke Detection Case study, the transformation has been done manually.

The full result (instantiation with 2 Smoke Detectors / 3 CAN bus segments) is displayed in Listing A.1 on page 177.

```
1   state INPUT_UPSTREAM {
2     (((OUT_CAN_Identifier_DOWNSTREAM = MFT_IDENT_REQUEST)
3       => ((TOSEND_UPSTREAM = OUT_CAN_Identifier_DOWNSTREAM) and (
            TOSEND_DOWNSTREAM = MFT_IDENT_ACK + myAddress)))
4      and
5      (((OUT_CAN_Identifier_DOWNSTREAM = MFT_SD_POLLING_REQUEST + myAddress))
6       => (TOSEND_UPSTREAM = CAN_ID_NO_DATA))
7      and
8      (((OUT_CAN_Identifier_DOWNSTREAM = MFT_SD_POLLING_REQUEST + myAddress) and
           (smoke = true))
9       => (TOSEND_DOWNSTREAM = MFT_SD_ALARM))
10     and
11     (((OUT_CAN_Identifier_DOWNSTREAM = MFT_SD_POLLING_REQUEST + myAddress) and
           (smoke = false))
12      => (TOSEND_DOWNSTREAM = MFT_SD_POLLING_REPLY))
13     and
14     (((OUT_CAN_Identifier_DOWNSTREAM = MFT_SD_POLLING_REQUEST) and (false))
15      => ((TOSEND_UPSTREAM = OUT_CAN_Identifier_DOWNSTREAM) and
16          (TOSEND_DOWNSTREAM = CAN_ID_NO_DATA)))
17     and
18     ((not ((OUT_CAN_Identifier_DOWNSTREAM = MFT_IDENT_REQUEST) or
19            (OUT_CAN_Identifier_DOWNSTREAM = MFT_SD_POLLING_REQUEST)))
20      => ((TOSEND_UPSTREAM = CAN_ID_NO_DATA) and
21          (TOSEND_DOWNSTREAM = CAN_ID_NO_DATA)))
22     );
23     transition[OUT_CAN_Identifier_DOWNSTREAM = CAN_ID_NO_DATA]
24       NO_INPUT;
25   }
```

Listing 12.3: Translation of processUpstream() to GTL

```
1   state INPUT_DOWNSTREAM {
2     //using: OUT_CAN_Identifier_UPSTREAM instead of (correct:)
          OUT_CAN_Identifier_UPSTREAM  & CAN_ID_MASK_NO_ADDR
3     ((((OUT_CAN_Identifier_UPSTREAM = MFT_IDENT_REPLY) or
4        (OUT_CAN_Identifier_UPSTREAM = MFT_SD_POLLING_REPLY) or
5        (OUT_CAN_Identifier_UPSTREAM = MFT_SD_ALARM))
6       => ((TOSEND_UPSTREAM = CAN_ID_NO_DATA) and
7           (TOSEND_DOWNSTREAM = OUT_CAN_Identifier_UPSTREAM)))
8      and
9      ((not ((OUT_CAN_Identifier_UPSTREAM = MFT_IDENT_REPLY) or
10            (OUT_CAN_Identifier_UPSTREAM = MFT_SD_POLLING_REPLY) or
11            (OUT_CAN_Identifier_UPSTREAM = MFT_SD_ALARM)))
12      => ((TOSEND_UPSTREAM    = CAN_ID_NO_DATA) and
13          (TOSEND_DOWNSTREAM = CAN_ID_NO_DATA)))
14     );
15     transition[OUT_CAN_Identifier_UPSTREAM = CAN_ID_NO_DATA]
16       NO_INPUT;
17   }
```

Listing 12.4: Translation of processDownstream() to GTL

Of particular interest is the translation of the operations `processUpstream()` (Listing 12.1) and `processDownstream()` (Listing 12.2). They are translated to invariants in the GTL model, see Listing 12.3 and Listing 12.4. Notice that a `switch`-like structure gets transformed to a logical

expression of the following form:

$$\bigwedge_{\mathsf{case}_i} (\mathsf{condition}_i \Rightarrow \mathsf{action}_i) \wedge \left(\neg(\bigvee_{\mathsf{case}_i} \mathsf{condition}_i) \Rightarrow \mathsf{action}_{\mathsf{default}}\right)$$

Formulation of the Verification Goals. The user requirements (page 107) are the basis of formulating verification goals concerning the system in terms of model elements. For the case study, the verification goals are formulated manually in GTL, see Listing 12.5 on page 124.

```
1  verify {
2    // REQ-SD-01: no CAN bus fault implies IdentRequest Round-Trip
3    (always (not can0.BusFault) and (not can1.BusFault) and (not can2.BusFault))
4      implies
5      (true until (CIDS.IdentRequestDone and (0 = CIDS.inferTopologyCanFail)));
6
7    // REQ-SD-02: CAN bus faults detected
8    always (((can0.BusFault) or (can1.BusFault) or (can2.BusFault))
9            implies
10           (true until (not (0 = CIDS.inferTopologyCanFail))));
11
12   // REQ-SD-03: (formulated for SD1 smoke and at most 1 CAN but fail)
13   (always
14    (((always ((not can1.BusFault) and (not can2.BusFault))) or
15      (always ((not can0.BusFault) and (not can2.BusFault))) or
16      (always ((not can0.BusFault) and (not can1.BusFault))))  and
17     sensor1.smoke)
18    implies
19    (true until (not (0 = cids0.smokeAlarm)));
20 }
```

Listing 12.5: Smoke Detection Verification Goals, formulated for 2 Smoke Detectors

Post-Processing the GTL — transformation to PROMELA. The GTL formulation of the smoke detection model (Listing A.1) has been post-processed with the `gtl.exe` as provided by TU Braunschweig.

The post-processing was only partially successful with gtl version `0.1 (9 Jun)`[1] was only partially successful: the resulting PROMELA file was too big (117MB), so the SPIN output file (`pan.c`) failed to compile.

However, it was possible to track down the output size problem to the formulation of the invariant corresponding to `processUpstream()`, Listing 12.3.

Constructing a *partial* model `smoke-part-large.gtl` with only one instance of that model (SDLOGIC), the `gtl.exe` output for (Listing A.2) is 18MB large (too big).

Now if the invariant is omitted, then the `gtl.exe` output for this simplified model (Listing A.3) is only of size 1.6KB.

Apparently the invariants are very closely related to the size of the PROMELA output.

[1] Output of `gtl --version`:

```
This is the GALS Translation Language of version 0.1.
Built on Thu Jun 9 10:01:01 UTC 2011.
Built from git tag:  a5357135ce507990999c43e1280c8857b61a785d.
Built from git branch:  master.
```

12.1.4 Assessment of Results

Evalation of Goals of the Case Study

Below is the evaluation of the goals as layed out in Section 12.1.1.

→ GOAL #1 [Evaluate modeling power of our CDSL]

During the case study the concepts of instances, interfaces, and behavioral state machines did suffice to express the desired behavior of the smoke detection system.

See sections 12.1.2, 12.1.2.

Goal Assessment: The CDSL is adequate in modeling power.

→ GOAL #2 [Evaluate usability of the modeling elements]

Most modeling elements allowed for a straightforward formulation of the concepts at hand and proved to be expressive. Hierarchical state machines are a powerful tool for behavioral description.

An exception to this is the modeling of data transformations (`processUpstream`, `processDownstream`, which seems cumbersome to express with the (language built-in) concept of decision trees.

See discussion in section 12.1.3.

Goal Assessment: CDSL should include a concept of decision tables.

→ GOAL #3 [Evaluate correspondence with GTL representation]

The guiding translation principles seem to be fairly straightforward to automate. Minor obstacles can be expected with respect to the transformation of transition actions to invariants.

See section 12.1.3

Goal Assessment: Automatic transformation from the CDSL to GTL should be implemented as sketched.

→ GOAL #4 [Evaluate performance of instance concept (scalability)]

The instance concept consisting of a template model and a instance table is sufficiently expressive to describe the (semi-)ring topology in the case study. In particular referencing the interface via index appears to be flexible and allows precise specification with only little overhead (like one smoke detector to connect to two CAN bus segment interfaces, identified as `OUT[id]` and `OUT[id-1]`).

See section 12.1.3.

Goal Assessment: Providing instances of interfaces and refer to them via an index seems adequate.

Further Lessons Learned

→ [Necessity to model LSCs as network of State-Machines]

If Locally Synchronous Components (LSCs) exhibit complex behavior it is very desirable to allow for *parallelism* in the description of the component.

In our case study, the smoke detector (see Figure 12.9) processes two data streams (one upstream, one downstream), so that it seems natural to decouple this behavior into separate state machines (Figure 12.11, Figure 12.12).

Building a product automaton for parallel state machines manually is both tedious and error-prone. Therefor, automated parts of the tool chain should perform this task:

(a) Either the transformation from CDSL to GTL, or

(b) The GTL should allow to group models into "one" locally synchronous component (LSC).

Of these options, (b) is the more attractive one: Other front-ends of the GTL language can benefit from this feature as well.

→ **[GTL should include bit-operators (bitwise and, or, negate, shift)]**
Some expressions are difficult to formulate (or at least: difficult to read), if basic bit-operators are missing. See Listing 12.4: The expression to compare with should be correctly

```
OUT_CAN_Identifier_UPSTREAM  & CAN_ID_MASK_NO_ADDR
```

While this can be equivalently expressed with integer arithmetic, the following operators should be supported by the GTL on integers for readability.

`A & B` bit-wise AND operation

`A | B` bit-wise OR operation

∼`A` one's complement operator (toggle all bits)

`A << CONST` bit-wise SHIFT-LEFT operation

`A >> CONST` bit-wise SHIFT-RIGHT operation

Here, `A` and `B` are expressions and `CONST` is a literal (`0..31`).

The semantics should be according to the `C` programming language. This is not assumed to cause problems, since the operations can be carried over (untranslated) to PROMELA, SCADE, or UPPAAL output.

→ **[GTL Performance Problem (large PROMELA output)]**
The case study did uncover a performance problem in the output generated by the GTL parser (here: to PROMELA). It was also possible to narrow it down somehow, see section 12.1.3. At this point in time this prevents further explorations of the tool chain.

It also shows that the naive translation from LTL to Büchi automata implemented in the current early prototype of the GTL tool is insufficient and needs to be optimized. We expect that this will happen in the course of the future work packages 2 and 3 of VerSyKo.

12.2 Level Crossing Systems

A level crossing is a location where a railway line is intersected by a road. To ensure a safe crossing of railway and road traffic, a level crossing system must be installed. Every time a train is approaching and entering the block section of the level crossing, the installed level crossing system has to ensure that the train is reliably recognized, and as a result, the road traffic is prevented from crossing the railroad in order to guarantee a safe passage for passing trains. A level crossing generally consists of different subsystem-components, distributed to different locations. Each component is a safety critical component with software realizing the functional behavior. Therefore, the software implementation tool SCADE is suitable for the development. As the synchronous components must exchange data to fulfill the overall system functionality, a communication infrastructure must be provided. Due to the

mentioned distributed localization of the (synchronous) components, the communication infrastructure is embedded in an asynchronous environment.

The level crossing regarded in the case study consists of the following modules that have been implemented as synchronous components in SCADE:

- Four traffic signal installations (traffic lights),
- one monitoring signal for the railroad traffic,
- two barriers,
- a detection systems called an axle counter[2] (including detection points and calculator),
- and an automatic control system responsible to navigate or control the overall system.

The model provides a perfect example of a GALS system and can therefore be used to serve as a benchmark for the GALS verification framework to be developed as part of the VerSyKo project. In the following, we will describe the user-level requirements for the level crossing system (Sec. 12.2.1). Based on the user-requirements, the synchronous subsystem-components have been developed in SCADE, whose implementation is introduced in Sec. 12.2.2 and 12.2.3. The generated code, derived from the SCADE implementations with the kcg-compiler of SCADE, has been used to verify the level crossing system at the source code level using SPIN/PROMELA. The results are provided in Sec. 12.2.4, showing that a direct verification at the source code level is not feasible. Indeed, suitable abstractions in form of contracts are necessary as envisioned by the GTL contract specification/verification framework of project VerSyKo.

Fig. 12.19 displays a graphical description of the level crossing system. The level crossing consists of a two-lane street open to be traveled in both directions. The street is intersected by a single unidirectional rail track. There are two traffic lights in each direction of the road traffic, a single monitor signal for the railroad traffic[3], an axle counter with a detection point at both ends of the level crossing, and an automatic control system. The automatic control system is connected to all other systems to be able to control these subcomponents.

The implementation of the level crossing system has been realized with the software modeling tool SCADE. As SCADE is used to implement reactive software systems, the implementation of the level crossing only includes the software part. Hardware parts are generally disregarded[4]. To be more accurate, hardware of the level crossing system is abstracted by interfaces.[5]

12.2.1 Requirements

For the level crossing system there is a detailed document of requirements [93]. This section will not provide a translation of all requirements of the requirement documentation. Only requirements that are necessary to understand the functionality of the level crossing system (called user-requirements) will be discussed.

[2] `http://en.wikipedia.org/wiki/Axle_counter`

[3] As the train can only pass the level crossing in one direction as described above.

[4] Hardware systems are not part of the research project. Only synchronous software systems built in SCADE are part of consideration.

[5] The abstraction of hardware via interfaces will be explained in Sec. 12.2.2.

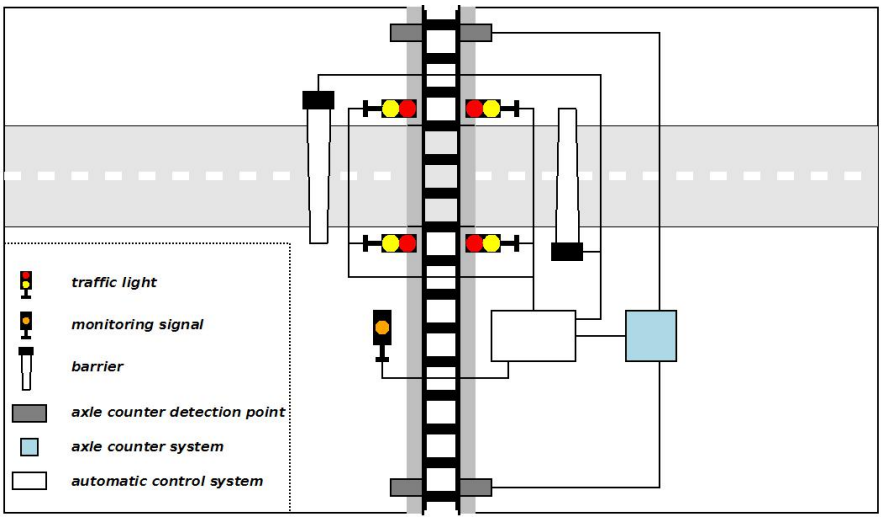

Figure 12.19: General layout of the level crossing regarded in the case study.

Cyclic execution behavior
The overall system functionality of the level crossing system is depicted in Fig. 12.20. The functionality is abstracted by a state machine (UML[6] style), where transitions and states are labeled with pseudo code. Generally, the level crossing system has a cyclic behavior. Every time a train enters the block section of the level crossing and is detected by the axle counter system, the road traffic is thereafter prevented from crossing the railroad to ensure that the train can safely cross, which is realized by the subsystem's automatic control system, traffic light(s) and barrier(s). In the state machine description of Fig. 12.20, the cyclic behavior starts in the state *Released*, where the road traffic is allowed to pass. If a train enters the block section of the level crossing, the level crossing must be safeguarded for the train (*Safeguarding I*, *Safeguarding II* and *Safeguarded*). Hence, the traffic lights are switched on to display a halt signal and the barriers are closed, before the monitor signal provides a proceed signal to the train conductor. After the train has passed the crossing street and exited the block section of the level crossing, the level crossing is again released (*Release*): All signals are switched off and barriers are opened and the road traffic is re-allowed to (safely) pass the level crossing.

Initialization
Before the cyclic behavior of Fig. 12.20 can be executed, the level crossing system, and therefore all of its subcomponents, must be initialized. The reader is referred to Fig. 12.21, where the initialization is visualized via a use case diagram.[7] The initialization includes the execution of a built in self-test of the traffic lights, as well as a built in self-test of the monitoring signal. After these self-tests have been completed successfully, the lights of the signals at the level crossing (traffic light(s) and monitoring signal) have to be turned off. Additionally, after the built-in self-tests of all traffic lights and the monitoring signal were confirmed, the barriers have to be brought to an open position. Thereafter, the level crossing system will change its state from *Initialized* to *Released* (see Fig. 12.20).

Safeguarding
Every time a train is approaching and detected by the level crossing system to have entered its block

[6]Unified Modeling Language — `http://www.omg.org`
[7]These information can also be withdrawn from the state machine diagram in Fig. 12.20.

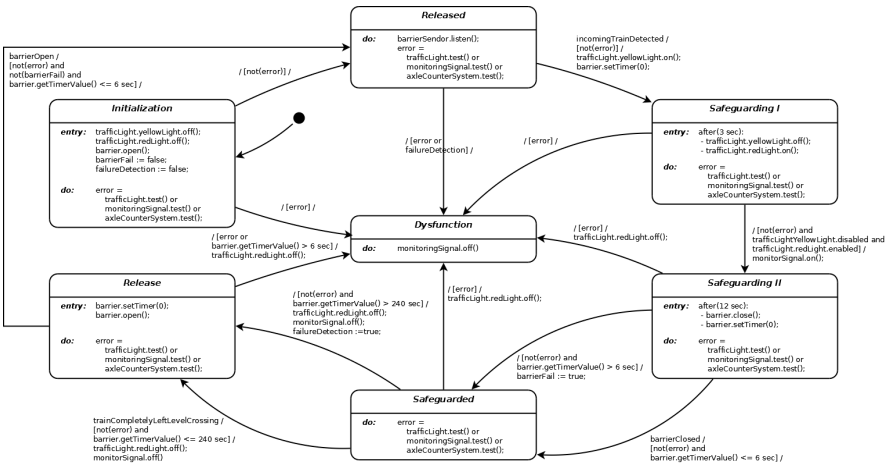

Figure 12.20: Abstract state machine model of the level crossing system.

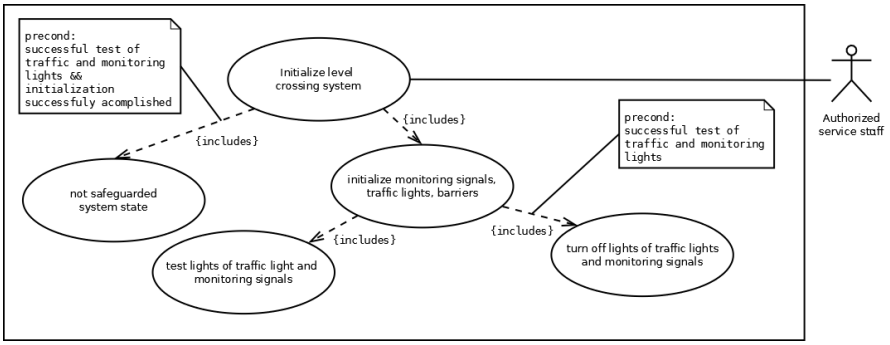

Figure 12.21: Use case of the use-requirement initialization of level crossing.

section, a safe passage of the train must be guaranteed. Hence, the train traffic must be protected from the road traffic and vice versa. Therefore, the traffic lights must indicate a stop/halt signal to the road traffic and the barriers have to be closed. At least, if all traffic lights indicate the stop/halt signal, the minimal precondition(s) for a safeguarding of the level crossing have to be fulfilled. As a result, the monitoring signal of the level crossing can signal the conductor that he is allowed to pass the level crossing by turning the light of the monitoring signal on (proceed signal).

With respect to the safeguarding, preconditions and timing issues are: Before a stop/halt signal (red light) can be indicated by the traffic lights, the traffic light must first switch to a warning signal (yellow light) and must hold this yellow light 3 seconds before turning it off and switch to the red light. Only if all traffic light instances provide a stop/halt signal to the road traffic, the monitoring signal is allowed to display a proceed signal for the train traffic. As well, twelve seconds after the traffic lights are signaling the stop/halt signal the barriers must be closed.

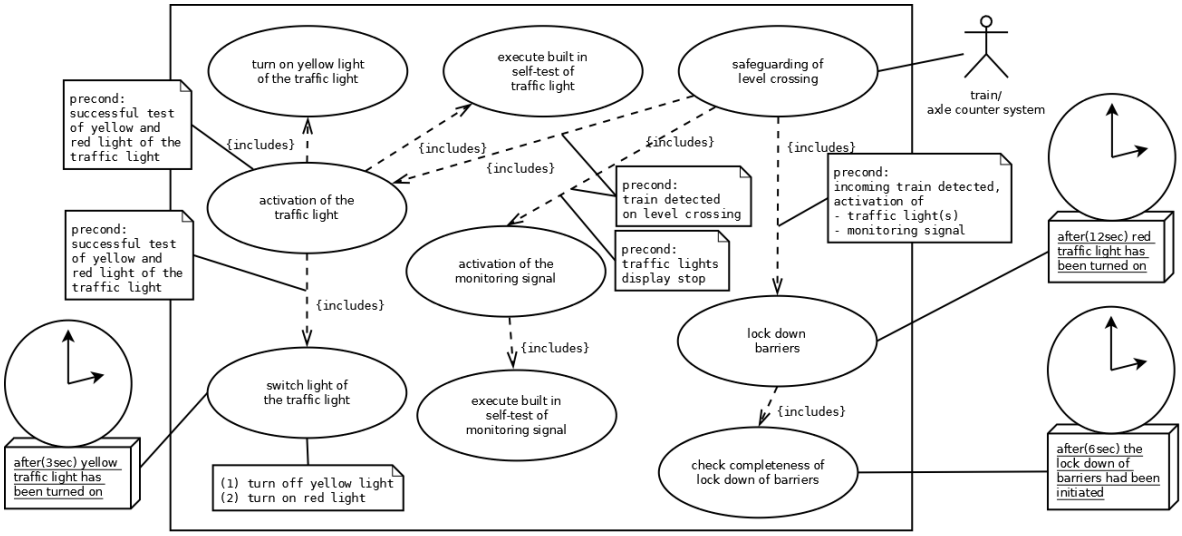

Figure 12.22: Use case of user-requirement safeguarding of level crossing.

The described situation is displayed in Fig. 12.19 and also in Fig. 12.22. The barriers and therefore the closing of the barriers are of secondary importance, i. e., they are an additional physical obstacle but not necessary for a safeguarding of the level crossing. The safeguarding by the traffic lights and the monitoring signal is considered as sufficient. Anyway, a feedback for the closing of the barriers is required by the level crossing system: Within 6 seconds after the closing of the barriers has been initialized a feedback is expected at latest. To verify that the barriers have closed within the fixed time frame, a timer is started and a sensor gives the necessary feedback. The kind of feedback and for what it is used for will be regarded later in this section when introducing the dysfunctional behavior.

Release

After all signals have switched to the right signal and the barriers have closed, the train can safely pass the level crossing. If the train left the block section of the level crossing, it is no longer necessary to prevent the road traffic from passing the railroad. As a result, the detection of the train exiting the block section of the level crossing will transfer the level crossing back into the unguarded state *Released* (see Fig. 12.19). The corresponding use case is shown in Fig. 12.23. The release of the level crossing includes the deactivation of both signal systems (traffic light system and monitoring signal

system), as well as the opening of the barriers. As a matter of course, the monitoring signal has to be turned off first, before the traffic lights can be switched off afterwards to communicate a proceed signal to the road traffic.[8]

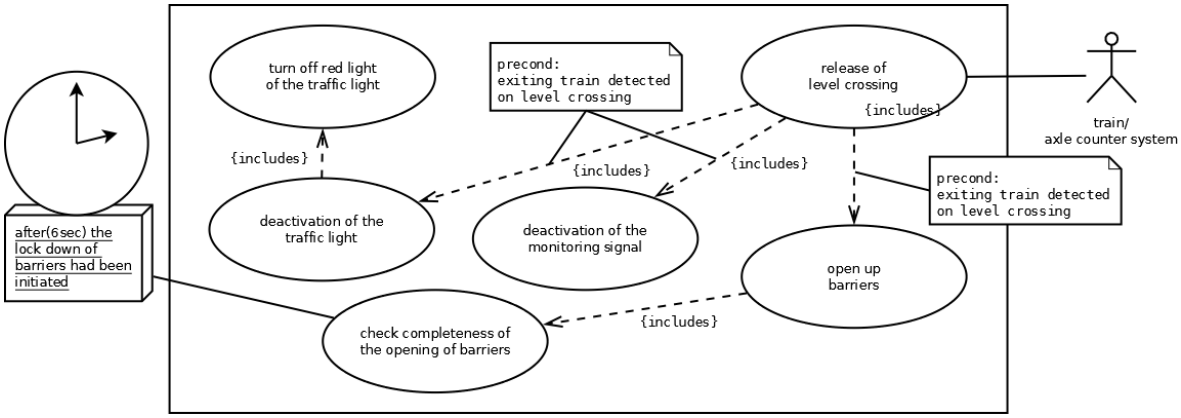

Figure 12.23: Use case of user-requirement release of level crossing.

Similarly to the use case of safeguarding, the position of the barrier is verified, that is, 6 seconds after the opening of the barriers has been initiated, a feedback is expected indicating that the barrier reached its opening position.

Dysfunction

A negative feedback or the absence of a feedback in the defined time interval of 6 seconds (concerning the barrier) is interpreted by the level crossing system as a dysfunction. This is true for the opening, as well as for the closing of the barriers. As mentioned before in paragraph ***Safeguarding***, the barrier is not necessary in the first place. But, if a dysfunction of a barrier is detected, it is stored, and after the safeguarded state of the level crossing has been exited, the cyclic execution behavior of safeguarding and release of the level crossing (system) will be left and a dysfunctional state will be entered (see Fig. 12.20).[9] The dysfunctional state is some kind of safe state for the overall system level crossing, where all participants (road and railway traffic) are protected with respect to a collision.

Other subsystem-components can also cause a dysfunction. As depicted in Fig. 12.24, a not successfully completed built in self-test of a traffic light or a monitoring signal is interpreted as a dysfunction. As well, there is a time frame of 240 seconds for trains entering the block section of the level crossing. If that time frame is violated, that is, if exiting of a train is not detected by the axle counter subsystem 240 seconds after the entering of the same train has been detected, it is interpreted as a dysfunction. Sooner or later the detection of a dysfunction will lead to the overall safe system state (Fig. 12.19: *Dysfunction*), where all signals (traffic lights and monitoring signal) are shut-down, and the barriers, as the case may be, are opened. This allows the road traffic to safely pass the level crossing, as no train is allowed to cross.

[8] A proceed signal of a traffic light conforms to a traffic light where all light (yellow and red) are turned off.

[9] The reader is referred to Fig. 12.19, where the dysfunction of a barrier is stored in the variable *barrierFail*. If this variable will be set to true, finally the state *Dysfunction* will be reached, where no train can pass the level crossing to protect the road traffic participants.

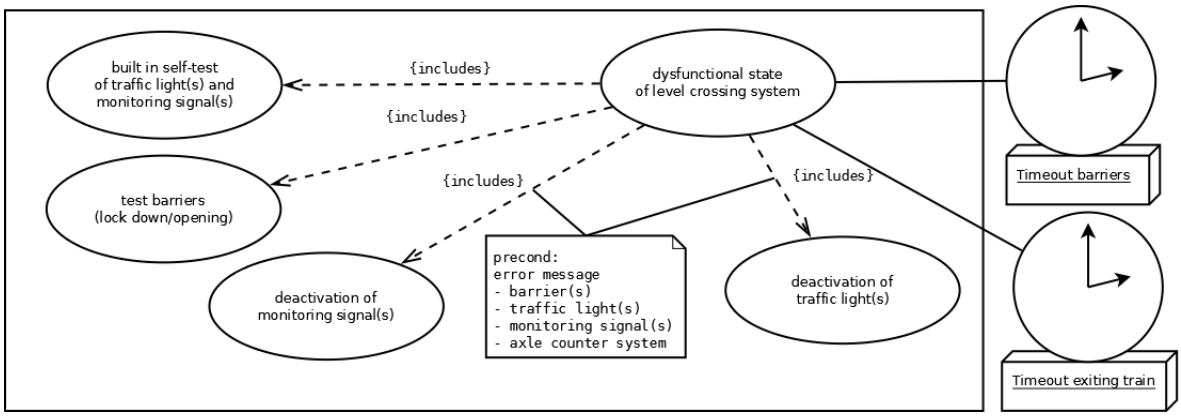

Figure 12.24: Use case of user-requirement dysfunction of level crossing.

12.2.2 System architecture

The architecture of the system is displayed in Fig.12.19. Each of the displayed modules (e. g. traffic light or axle counter) is a synchronous component implemented in SCADE. The solely synchronous components are connected via a communication infrastructure in an asynchronous environment (see Fig. C.2 of appendix C[10]). The bright yellow components displayed in Fig. C.2 are the solely synchronous system parts of the overall asynchronous system architecture. These synchronous subsystem-components are connected to each other by an asynchronous communication infrastructure (represented as wires) to exchange data via their defined input and output interfaces.

As a result, the overall architecture is a so called GALS architecture specified as system of systems, as described in Sec. 1.1. Some of synchronous components are connected directly (e. g. *StrassenSignal* and *UeberwachungsSignal*), but in the majority of cases the components are connected indirectly via the controlling system component *BahnUebergangsSteuerung* (the automatic control system). As can be seen in Fig. C.2, not all interfaces of the synchronous components have a connection to other subsystem-components. They have been left unconnected:

- to be able stimulate the model or some of its subcomponents, and
- to observe outputs for the interpretation of results.

Input interfaces are generally used for providing stimuli that in most cases replace and simulate hardware parts. For example the input interface *UmgebungZug* replaces the hardware detection points of the axle counter system, i. e., the interface is an abstraction or rather a simulation of the axle detection point hardware. The same is true for the hidden interfaces that are located at the synchronous components: E. g. *AZRTestGestoert* is the abstraction of the built in self-test of the axle counter system component simulating the hardware. The Boolean input stimulus can simulate a successful or failed built in self-test of the system component.

For the verification it is necessary to interpret and observe the system behavior. As a result, additional output interfaces have been implemented in the model of the synchronous components. For example *UmgebungZugAnachB* is used to be able to observe the behavior of the component

[10]Due to space, some figures and tables have been located outside this section. The figures and tables of the *Case Study Level Crossing* is located in appendix C.

StrassenSignal (traffic light) or the overall system behavior (level crossing), respectively, and therefore what is communicated to the road traffic participants. Similar the output interface *UmgebungZug* has been implemented for the synchronous component *UeberwachungsSignal* (monitoring signal) observing the commitment to the train conductor. Input and output interfaces are used as a kind of black-box-view of the system and to specify test-cases. More detailed information about specifying *LTL* test-cases using input and output interfaces are given in Sec. 5.

12.2.3 SCADE Models of System Components

Tokens, exchanged by the synchronous components via the asynchronous communication infrastructure, are in the majority of cases enumerations. Only some interfaces are Boolean typed. The used non-Boolean data types are listed in Tab. C.1 located in the appendix C of this document.

The first eight data types listed in Tab. C.1 (<identifierName>Kommando, <identifierName>Zustand) are used to broadcast the state of the corresponding subsystem to other subsystem-components of the system, or to communicate directives to it. For example the component *UeberwachungsSignal* (monitoring signal) has an interface of type *TUeberwachungKommando* and another interface of type *TUeberwachungZustand*. The interface typed *TUeberwachungKommando* is an input interface and used to receive directives from the automatic controller subsystem, that is, the subsystem *BahnUebergangsSteuerung*. If a signal *UeberwachungTest* of type *TUeberwachungKomanndo* is received, it is used as a directive to trigger the execution of the built in self-test for the subsystem-component *UeberwachungsSignal*. On the other hand, the output interface of *UeberwachungsSignal* typed *TUeberwachungZustand* provides information about the state of the subsystem *UeberwachungsSignal*. For example, if the built in self-test has successfully been finished, the signal *TestOB* will be forwarded via the interface of type *TUeberwachungZustand* to outside.

The other last four data types, listed in Tab. C.1, are data types that are either used to simulate hardware (the axle counter detection points, coexisting hardware of the submodule the software is located at etc.), to simulate manipulations from outside (service interfaces), or used to communicate the state of the system (a la black-box) by interfaces to outside. The latter has already been described in the last two paragraphs of Sec. 12.2.2.

In the following paragraphs, the implementation of the level crossing as a system of system will be introduced. The description is provided component by component and is guided by the description of the user requirements provided in Sec. 12.2.1.

BahnUebergangsSteuerung (Controlling system unit)
The component *BahnUebergangsSteuerung* is the heart of the GALS system, as it includes the logic of the level crossing system and is responsible for the overall system functionality, introduced as user requirements in Sec. 12.2.1. As a result, there exists two interfaces (one input and one output interface) for each existing subsystem-component of the level crossing system used for the exchange of data. Fig. 12.25 provides an overview of the subcomponent *BahnUebergangsSteuerung* and its provided (output) and required (input) interfaces. In addition to this, an overview of the types and the signals of the provided and required interfaces are given in Tab. C.2 and C.3 (appendix C). For each instance of the included subsystem-components of the level crossing, an input respectively an output interface can be identified. If more than one instance of the same component is part of the overall system model (e. g. the barrier or the traffic lights), these inputs are combined to a vector.

Generally, output interfaces of the subcomponent *BahnUebergangsSteuerung* are used to control and forward instructions to the other subsystem-components of the level crossing system. The possible instructions are communicated via signals. With the directives, listed in Tab. C.3, the submodule

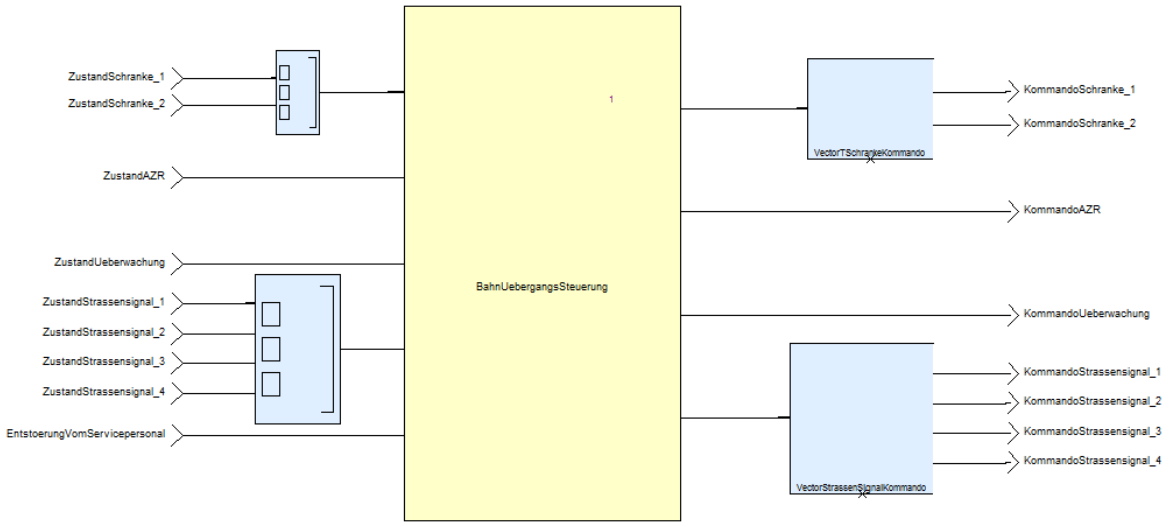

Figure 12.25: Subcomponent BahnUebergangsSteuerung including provided and required interfaces.

BahnUebergangsSteuerung tries to enable a faultless execution. This also includes that the system is conveyed to a safe state, if a malfunction occurs. To verify that instructions have been executed correctly, and to be able to calculate necessary instructions for the further execution steps, feedback information are received via the input interfaces (listed in Tab. C.2). These input interfaces receive state information about the other subsystems of the level crossing, as described in the second paragraph of Sec. 12.2.3. For example *ZustandAZR* requires and is provided with information about the state of the submodule *AchsZaehlRechner*, *ZuststandSchranke* requires respectively is provided with state information of submodule Schranke and so on.

Disregarding hidden inputs, there only exists one visible (not hidden) input interface which did not receive state information from another subsystem-component of the level crossing system model: *EntsicherungVomServicePersonal*. This interface has been implemented to reset the subsystem component *BahnUebergangsSicherung*, that is, if any malfunction or disturbance has arisen and the subsystem has fall back to a safe state, it can be restored to its initial state (see Fig. 12.21) via the service interface by the authorized service staff. The interface is not listed in Tab. C.2, as it is simply Boolean typed. If it is true, the subsystem is restored to its initial state.

The component *BahnUebergangsSteuerung* is at the top level and can be further decomposed. The result of decomposition can be regarded in Fig. 12.26, where three main states can be identified:

- *InitialisierungBahnuebergangskomponenten*,
- *NormalbetriebBahnuebergang* and
- *StoerungBahnuebergang*.

The state *InitialisierungBahnuebergangskomponenten* is responsible to realize the initialization, as discussed in the paragraph ***Initialization*** of Sec. 12.2.1 and figured out as use case in Fig 12.21. The state *NormalBetriebBahnuebergang* controls the cyclic behavior as described in paragraphs ***Cyclic execution behavior***, ***Safeguarding*** and ***Release*** of Sec. 12.2.1 and realizes the user requirements *Safeguarding* and *Release*, displayed in Fig. 12.22 and 12.23.

The last state, that is, state *StoerungBahnuebergang* implements the user requirement *dysfunction*, as it is described in paragraph ***Dysfunction*** of Sec. 12.2.1. For the corresponding use case description, the reader is referred to Fig. 12.24. The states *InitialisierungBahnuebergangskomponenten* and *NormalBetriebBahnuebergang* again include a deeper hierarchical structure and can therefore be further decomposed.

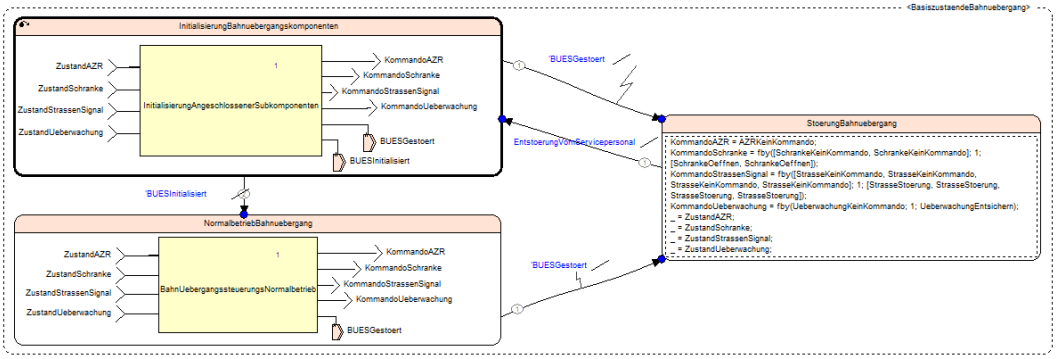

Figure 12.26: Three main states of the subsystem-component *BahnUebergangsSteuerung*.

Initialization

Decomposing the state *InitialisierungBahnuebergangskomponenten* means to decompose the included node of the state *InitialisierungBahnuebergangskomponenten*. The implementation rejected by the decomposition of the node *InitialisierungAngeschlossenerSubkomponenten* leads to the implementation of Fig. 12.27. For each subsystem-component instance in the overall GALS system level crossing, there exists node responsible for: The initialization of the component instance and the validation of a successful initialization of the corresponding subsystem.

Concerning the functional behavior, as defined by the user requirements (see Sec. 12.2.1, paragraph ***Initialization***, Fig. 12.20, and Fig. 12.21), the build in self-test of the subsystem-components traffic light(s) (instances of *StrassenSignal*), monitoring signal (*UeberwachungsSignal*) and the axle counter system (*AchsZaehlRechner*) must be initiated. After a responds of a successful self-test execution, the subsystem-components must be brought to their corresponding initial state, that is, the lights of the traffic light(s) as well as the monitoring signal are turned off, and the barriers are brought to an open position.

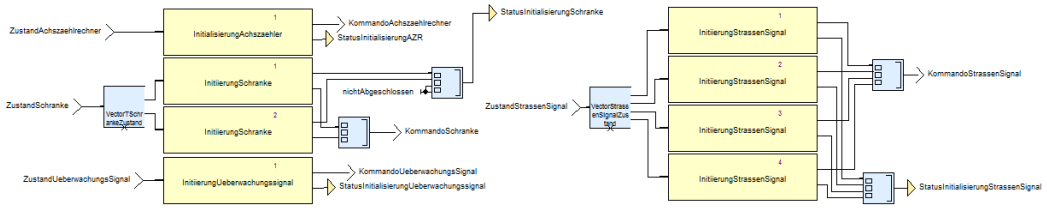

Figure 12.27: Initialization-nodes for the subsystem-components of the level crossing system.

The initialization of the subsystem-components takes place inside the nodes displayed in Fig. 12.27 (*InitialisierungAchazaehler*, *InitialisierungSchranke*, *InitialisierungStrassenSignal* etc.). The implementations which is located inside these nodes are quite similar. Generally, there are differences regarding the used data types of the signal that is received from or send to the corresponding

subcomponent of the level crossing system and used for calculation. An example is given in Fig. C.1 in appendix C: Regarding the upper and lower implementation model, only slightly differences can be identified. First, the communication between the corresponding subcomponent is handled by different type of signals. Second, the component *AchsZaehlRechner* is additionally in an error-state, if the block section is occupied by a train before the initialization phase has not yet been finished. Contrary, in the implementation model of the monitoring signal, only a failed execution of a build in self-test is interpreted as an error.

For the reason of similarity and to avoid too much redundancy regarding the implementation of the nodes depicted in Fig. 12.27, only node *InitialisierungUeberwachungsSignal* (the monitoring signal depicted in the upper synchronous implementation model of Fig. 12.27) is introduced. The other nodes can be derived from the SCADE model and the following description of *InitiierungUeberwachungsSignal*. As required, signal *UeberwachungTest* is being emitted to the subcomponent *UeberwachungsSignal* (see Fig. 12.19) to trigger the build in self-test. If the signal has been emitted, the sub-node *InitialisierungUeberwachungsignal* waits for a response that may either be positive (*UeberwachungTestOB*) or negative (*UeberwachungGestoert*). The response is validated by the implemented SCADE state machine displayed in Fig. C.1. In case the built in self-test has been successfully executed, the implemented state machine *Validation* will change its state from *Testaufruf* to *ErfolgreichAbgeschlossenerTestUndInitiierung*. If not, that is, the signal *TestGestoert* is received, state machine *Validation* changes state to *TestaufrufFehler*. For the latter case, the result of a failed initialization is forwarded to the outside of the node, i. e., to the implementation displayed in Fig. 12.26, leading to a change of states from *InitialisierungBahnuebergangskomponenten* to the state *StoerungBahnuebergang* (for a detailed description see paragraph ***Dysfunction*** below in this section). Otherwise, if a positive feedback is received (*TestOB*), the initialization phase will proceed: The signal *UeberwachungInit* is emitted to the subsystem-component *UberwachungsSignal* (monitoring signal). *UeberwachungIint* should have the effect that the signal light of the *UeberwachungsSignal* is turned off. After receiving an acknowledgment that the light of the monitoring signal has been turned off, the initialization of the level crossing system is successfully finished.

If all subcomponents successfully execute their initialization, validated by the implementation depicted in Fig. 12.28, the result is forwarded to outside to the state machine implementation of Fig. 12.26, leading to a change of state from *InitialisierungBahnuebergangskomponenten* to the state *NormalbetriebBahnuebergang*.

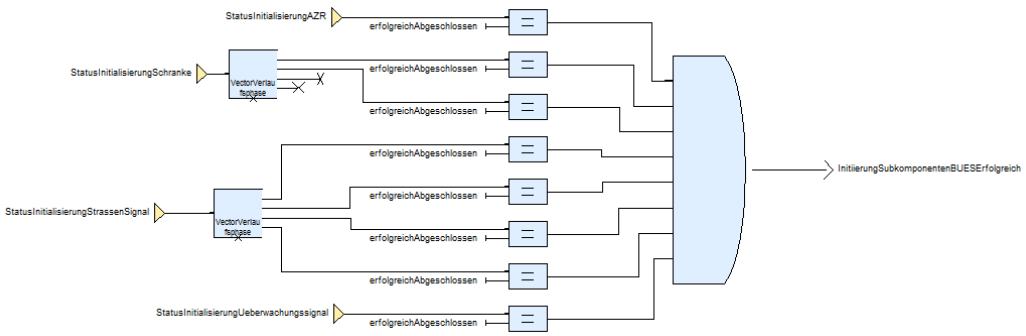

Figure 12.28: Validation of an overall successful validation of the level crossing system.

Released

If the state *NormalbetriebBahnuebergang* has been reached, the level crossing system is in the initial state *Released* (see Fig. 12.20) and the automatic control system (*BahnUebergangsSteuerung*) has to enable the cyclic execution behavior described in paragraph ***Cyclic execution behavior*** of Sec. 12.2.1. Basically, the cyclic execution behavior is realized by the implementation depicted in Fig. C.3 (appendix C) which is located inside the node *BahnUebergangssteuerungsNormalbetrieb* of state *NormalbetriebBahnuebergang* (see Fig. 12.26). The implementation model is described as a SCADE state machine and has been kept similar to the state machine description of the requirement section (Sec. 12.2.1, Fig. 12.20). The state *Release* of Fig. 12.20 corresponds to the initial state *Entsichert* of Fig. C.3. In the state *Entsichert* all subsystem-components of the level crossing maintain their states (e. g. the barriers are kept open, the lights of the monitoring signal and the traffic lights are remained switched off etc.). As required, only the build in self-tests are periodically executed respectively invoked. The corresponding implementation can be seen in Fig. 12.29. From now on, the depicted implementation of Fig. 12.29 is called the default implementation model of the corresponding submodule instance, the signal should be communicated to. For example the default implementation model for the *StrassenSignal* (traffic light) is the one which is located at the upper left corner.

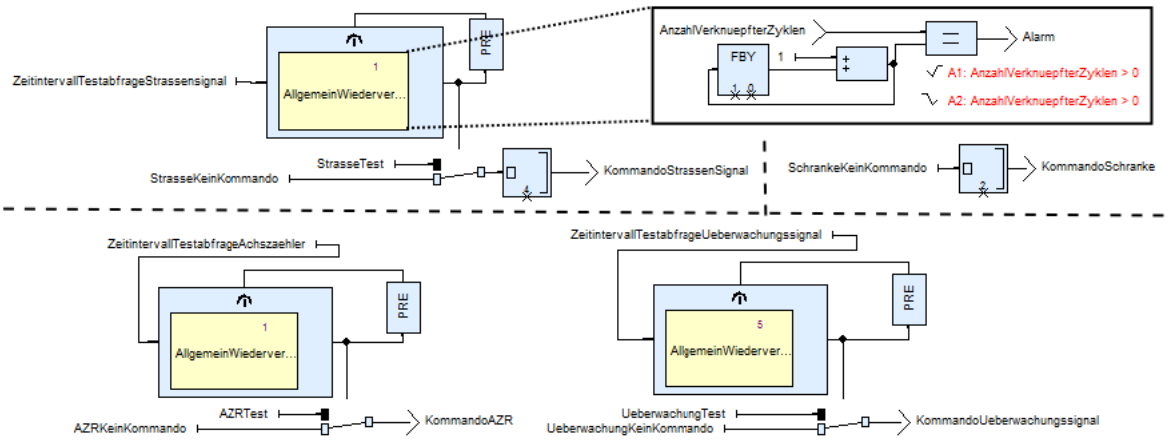

Figure 12.29: Implementation to periodically emit signal for executing build in self-test of traffic light.

As the default implementation models are implemented in equal manner, a generic explanation for the default implementation models displayed in Fig. 12.29 will be given. As long as no train is detected to enter the block section (see outgoing transition of state *Entsichert* in Fig. C.3 on page 199) of the level crossing, the default neutral placeholder signal *<identifierName>KeinKommando* is emitted to the corresponding (*<identifiereName>* : *UeberwachungsSignal*, *StrassenSignal*, *Schranke*, *AchsZaehlRechner*) subsystem-component (see also Fig. C.2 in appendix C). The placeholder signal is periodically interrupted after *ZeitintervalTestabfrage<identifierName>*[11] instants by the signal *<identifierName>Test*, invoking the build in self-test. Caused by the fact that the subsystem *Schranke* did not have any build in self-test, the default implementation model is quite different: Solely signal *SchrankeKeinKommando* is communicated to the barrier at each logical instant. Again the described implementation can be found in Fig. 12.29.

[11] An integer constant used to determine the number of logical instants. The squared snipes of Fig. 12.29 displays the implementation of the specified time interval.

Safeguarding

Thus, a change of state from *Entischert* to *Sichern1* will occur (see Fig. C.3), if the controlling unit (*BahnUebergangsSteuerung*) receives a signal from the axle counter system, indicating that a train has entered the block section of the level crossing (*ZustandAZR* = *AZRBelegt*). As required (see Sec. 12.2.1) the controlling unit commands the traffic light component *StrassenSignal* to safeguard the railway road (*StrasseSichern*), which is implemented in the node *NormalbetriebSichern1* displayed in state *Sichern1* in Fig. C.3[12]. The implementation of the node *NormalbetriebSichern1* is partly represented in Fig. 12.30 regarding the control of the traffic light instances.

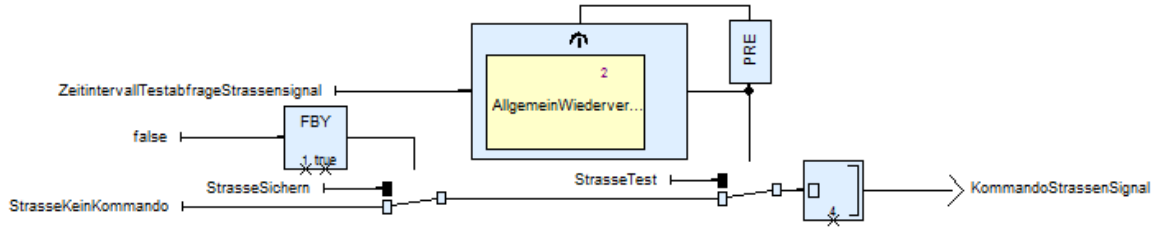

Figure 12.30: Safeguarding the level crossing by indicating traffic light(s) to switchover to a stop signal

At the first instant where the state *Sichern1* gets activated, the included node *NormalbetriebSichern1*, and therefore the implementation model of Fig. 12.30 emits the signal *StrasseSichern*. The signal is broadcast to the traffic light subsystem-component. If the signal is received by a traffic light instance, the component starts safeguarding the railroad traffic switching the correct lights (for short: the traffic lights get activated by turning on the yellow lights before switching to the red lights). After the *StrasseSichern* signal has been sent, only signal *StrasseKeinKommando* is emitted, again periodically interrupted by the emission of the signal *StrasseTest*. The implementation regarding the emission of signals to the other subsystem-components is realized by the default implementation models introduced in paragraph ***Released*** (see also Fig. 12.29).

As long as no dysfunction is reported or existing instances of *StrassenSignal* (traffic light) did not acknowledge displaying a stop signal, the state *Sichern1* is kept. This also corresponds to the user requirements introduced in Sec. 12.2.1 and displayed in Fig. 12.20 and 12.22. Disregarding a dysfunction, a change of state is performed, if and only if all instances of traffic light components of the overall level crossing system acknowledge displaying a stop signal. As a result, the state *Sichern2* will be reached (see Fig. C.3).

Up to now, regarding the requirements introduced in Sec. 12.2.1, neither the monitoring signal nor the barrier is in the right state (with respect to a safeguarded level crossing). The implementation for actuating the component monitoring signal is depicted in Fig. 12.31. As described by the use cases of Fig. 12.20 and 12.22, it must be ensured that the traffic light(s) display a stop signal before the monitoring signal gets activated. If not, train traffic and road traffic is allowed to pass the level crossing at the same time which may cause an accident. To avoid that a proceed signal is simultaneously displayed by traffic lights and the monitoring signal, the upper implementation model of Fig. 12.31 is used. The variable *AlleStrassenSignaleAufHalt* results in a Boolean value *true*, if and only if the traffic light components communicate to the automatic control system to display a stop signal and did not change their state of displaying it. Only if the latter is ensured, the monitoring signal is invoked to

[12] A detailed description of the switchover of the traffic light invoked by the signal *StrasseSichern* will be given in paragraph ***Component StrassenSignal (traffic light)***

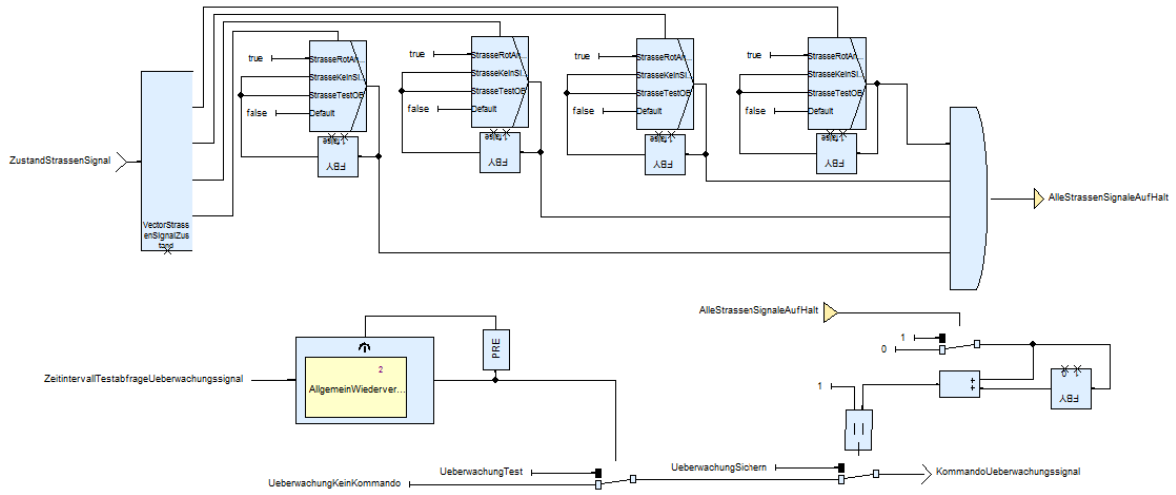

Figure 12.31: Implementation to ensure traffic light(s) displaying stop signal..

change to display a proceed signal for the train.

As mentioned before, the state of the barriers must be manipulated by the automatic control system. Twelve seconds after the traffic lights have been turned on, it is required (***Safeguarding*** in Sec. 12.2.1) that the barriers should start to close down. This is implemented in the SCADE state machine depicted in Fig. C.3. The transition between *Sichern2* and *Sichern3* implements the times operator (*ZeitintervalSchrankenschliessungLogischeEinheiten times true*) where the constant *ZeitintervalSchrankenschliessungLogischeEinheiten* defines the number of cycles of the synchronous component that are equivalent to twelve seconds in the real world (see introduction of SCADE in Sec. 2.1: Multiform of time). The implementation of the node *NormabetriebSichern3* in state *Sichern3* is again equal to the default implementation model already introduced in Fig. 12.29. The reader is referred to paragraph ***Release***. As the barrier should close down, the implementation for emitting signals to *Schranke* (barrier) differs. Before the placeholder signal *SchrankeKeinKommando* is emitted, the signal *SchrankeSchliessen* is communicated at the first instant, in which the state *Sichern3* is active. The corresponding implementation is displayed in Fig. 12.32, which should invoke the closing of the barriers.

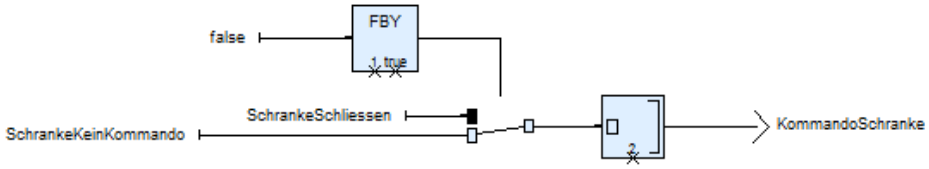

Figure 12.32: Implementation for emitting signal to close down barrier.

If the monitoring signal (the instance of component *UeberwachungsSignal*) acknowledged that it has been turned on, i. e., a proceed signal is displayed, and the barriers have been closed down[13] a state change from *Sichern3* to *Gesichert* is performed (see Fig. C.3 in appendix C). This corresponds to the state *Safeguarded* of the state machine in Fig. 12.20.

[13] Alternatively the time span between the signal emitted to close down the barriers and present is greater then six seconds.

Safeguarded

In the state *Gesichert* (see Fig. C.3) the inside located node *NormalbetriebGesichert* again includes the default implementation models, as visualized in Fig. 12.29. These implementation models are periodically executed, as long as no subsystem-component indicates an error (*ZustandAZR = AZRGestoert or ...*), the axle counter component detects and communicates that the train did exit the block section of the level crossing (*ZustandAZR = AZRFrei*), or more than 120 seconds have passed since an entering of a train in the block section of the level crossing has been detected but yet not the exiting (Fig. C.3: Transitions with guards *EntscheidungGetroffenBlockabschnitt and not ErgebnisBlockabschnitt*). If one any of the latter two conditions is fulfilled, the level crossing system must be released.

Release

For the release of the level crossing system, the controlling system unit must suspend the monitoring signal by emitting the signal *UberwachungEntsichern* to the monitoring signal subsystem-component (see Fig. 12.33). After the monitoring signal has been turned off, the traffic light instances must also be suspended, to allow the road traffic to pass the crossing. The suspension of the traffic light is realized by the automatic control system by sending the signal *StrasseEntsichern* (see also Fig. 12.33). At least, the physical obstacles must be removed by reopening the barriers. As can be inferred from the SCADE state machine from Fig. C.3, the release of the level crossing system is implemented by two succeeding states: *Entsichern1* and *Entsichern2* and their nodes located within. The implementation of node *NormalbetriebEntsichern1* located within state *Entsichern* ensures that the monitoring signal is turned off. After the acknowledgment has been received that the monitoring signal stops to display a proceed signal, the traffic lights are turned off to stop displaying a halt signal. The corresponding implementation can be regarded in Fig. 12.33

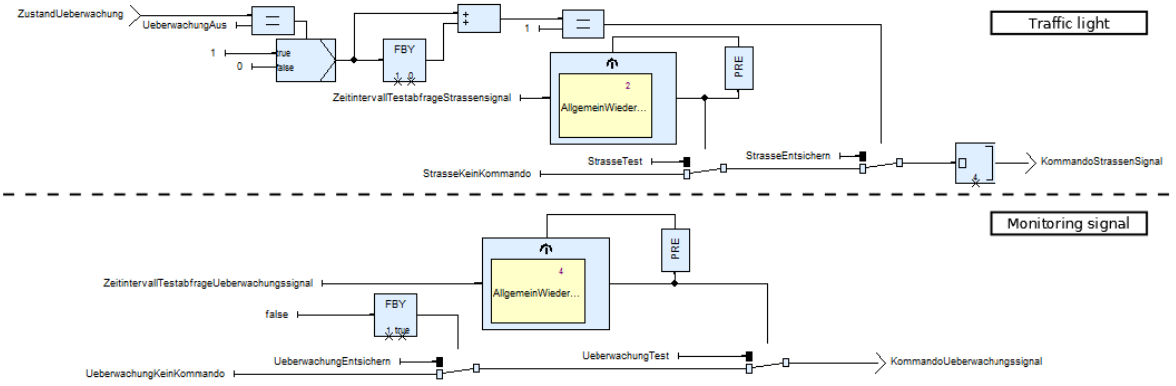

Figure 12.33: Implementation of the suspension of signals in state *Entsichern1*.

Presupposed that all instances of the traffic lights as well as the monitoring signal have been turned off, the change of state to *Entsichern2* is performed. In state *Entsichern2* the last necessary step is executed, i. e., the barriers are invoked to open. The implementation is presented in Fig. 12.34. As before, all other implementations are realized by the default implementation depicted in Fig. 12.29.

If no error is reported from any subsystem-component of the level crossing system and the complete opening of the barriers have been verified, the release phase is enclosed and the transition to the state *Entsichert* (released) is performed.

Figure 12.34: Implementation of the reopening of the barrier in state *Entsichern2*.

Dysfunction
A dysfunction of the level crossing occurs, if:

- any of the existing subsystem indicates an error to the automatic control system, or
- a timeout concerning the responds time of acknowledgment is detected.

An error may be detected by the subsystem-components, by the execution of the build in self-test. In either case, the success or the failure is forwarded to the automatic control system. The automatic control system validates the returned messages, and if necessary, establishes a safe state of the overall system (see incoming transitions of state *Gestoert* in Fig. C.3 and Fig. 12.26). The automatic control unit also checks and validates necessary timeouts. For example the barrier that must be closed or opened within a time frame of six seconds after the corresponding command (*SchrankeOeffnen* or *SchrankeSchliessen*) has been sent from the automatic control system. If timeouts are exceeded (see also incoming transitions of state *Gestoert* in Fig. C.3), the system will fall back to the defined safe state.

In the safe state the monitoring signal is turned off (halt/stop signal). As no train is allowed to pass the level crossing, if a halt signal is displayed by the monitoring signal, it is now possible to allow the road traffic participants to go across. Therefore, also all traffic lights are turned off (which is equivalent to a proceed signal) and the barriers are reopened. The level crossing is now in a safe state, where at least the road traffic is maintained.

Component AchsZaehlRechner (axle counter system)
As can be seen in Fig. 12.35, first the build in self-test must be invoked and successfully be finished, before axle counter system can be initiated from outside. After initialization, the axle counter system

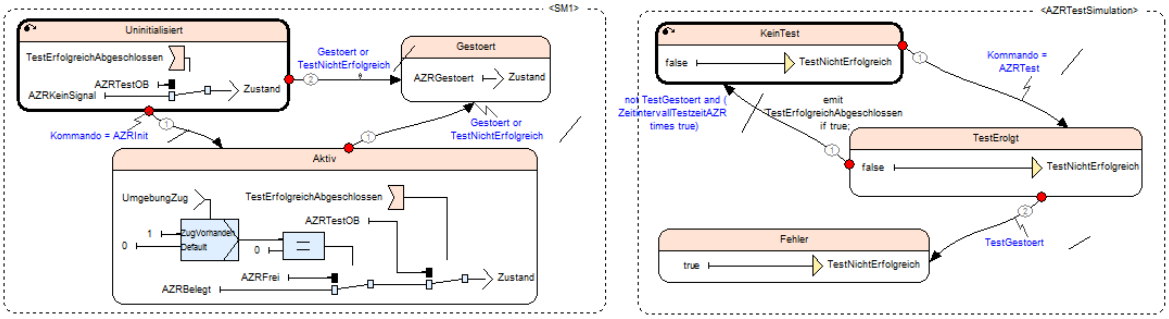

Figure 12.35: Implementation model of the axle counter system.

reacts to inputs from the detection points (interface *UmgebungZug*). If a train is detected to enter or to leave the block section of the level crossing, the information is transformed to a signal and forwarded to the automatic control system (signal *AZRFrei* or *AZRBelegt*). At any time, further build in self-tests

can be invoked from outside. If the self-test can successfully be finished, the system returns to its general behavior of forwarding occupation information of the level crossing block section. If not, the system performs a change of state from *Aktiv* to *Gestoert* and an error message (*AZRGestoert*) is communicated to the automatic control system. The same is true for the execution of the build in self-test in the initial state (*Uninitialisiert* of the axle counter system, where a change of state is performed from the state *Uninitialisiert* to the state *Gestoert*

Component StrassenSignal (traffic light)

The implementation model of the traffic light is depicted in Fig. 12.36. Again the subsystem-component must have execute a successful build in self-test before the traffic light component can be initialized. If the self-test fails, the state of the system changes from *Uninitialisiert* to *Gestoert*, an error message is communicated the automatic control system and the lights of the traffic lights are switched off (see paragraph ***Dysfunction***). The build in self-test can be executed at any time if invoked from outside, except for the case that the traffic light is in the state *Gestoert*. If the self-test was successful, the traffic lights can be initialized (*StrasseInit*), where all lights are switched off.

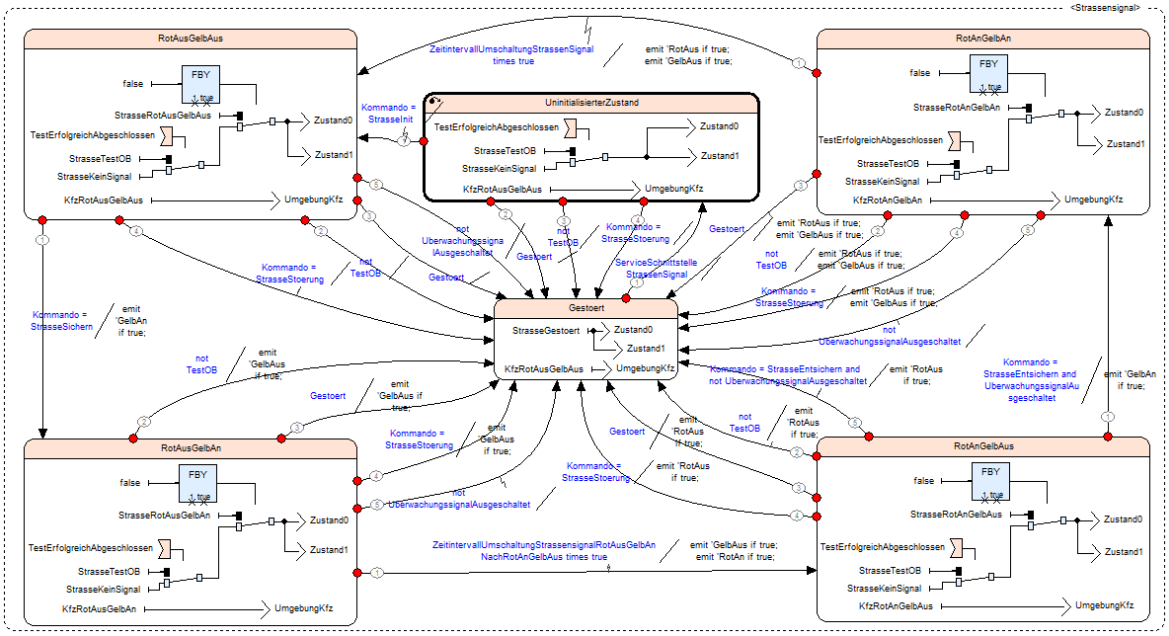

Figure 12.36: Implementation model of the traffic light.

If invoked from outside by the signal *StrasseSichern*, the traffic light can be switched on (*Kommando = StrasseSichern*), that is, first the yellow lights are switched on (state *RotAusGelbAn*), before after three seconds (implemented by the transition guard *ZeitintervallUmschaltungStrassensignal-RotAusGelbAnNachRotAnGelbAus times true*) the red light are switched on and the yellow light is switched off (state *RotAnGelbAus*). By the command *StrasseEntsichern* of the automatic control system, the traffic lights can regularly be switched off to allow the road traffic to pass the level crossing. As required in paragraph ***Release*** of Sec. 12.2.1, the monitoring signal must therefore provide a halt/stop signal. This is ensured by the label *Kommando = StrasseEntsichern and UberwachungssignalAusgeschaltet* of the incoming transition of state *RotAnGelbAn*, where *UberwachungssignalAusgeschaltet* ensures that the monitoring signal has been switched off, before the traffic lights can be

turned off to display a proceed signal.

Component UeberwachungsSignal (monitoring signal)
Again, a successful build in self-test must be invoked and performed before the monitoring signal can be initiated. As usual, a failed execution would lead to an error message, communicated to outside. As a consequence, the monitoring light is switched off to prohibit trains passing the level crossing, which is necessary to protect the road traffic participants.

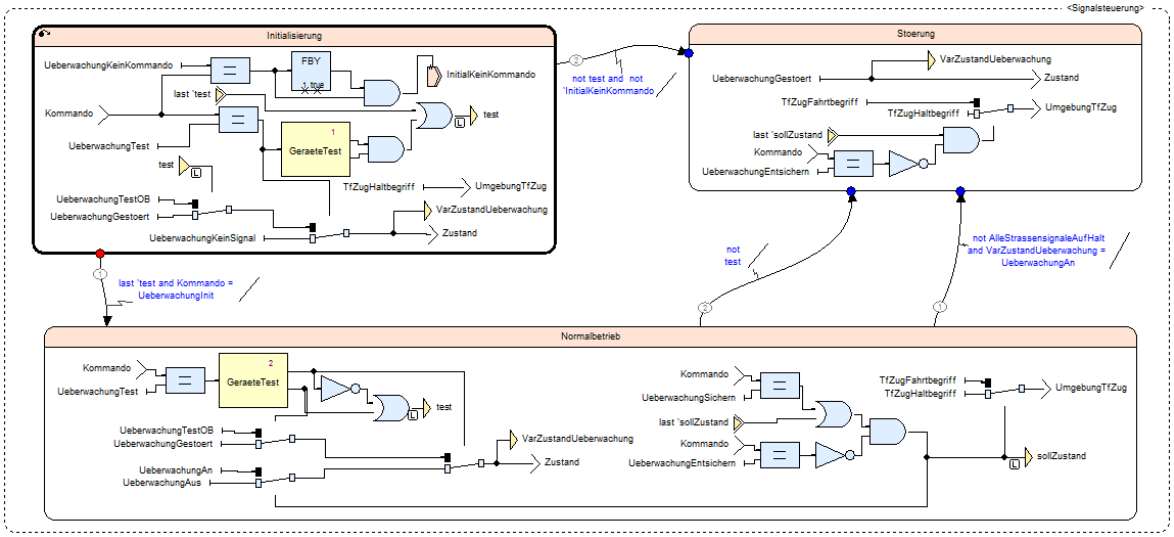

Figure 12.37: Implementation model of the monitoring signal.

A successful test and an additional initialization invoked from the automatic control system, leads to the state *Normalbetrieb* of the monitoring signal, where the reception of the signal *UeberwachungSichern* causes that the light of the monitoring signal to be switched on respectively the reception of the signal *UeberwachungEntsichern* causes the light of the monitoring signal to be switched off. The described behavior holds as long, as no error is detected, that is, the periodically invoked build in self-tests are successfully executed and no traffic light is simultaneously switched on (see outgoing transitions of state *Normalbetrieb*).

Component Schranke (barrier)
The implementation model of the barrier is displayed in Fig. 12.38. A barrier can either be not initialized (*Uninitialisiert*), opened (*Geoeffnet*), closed (*Geschlossen*), faulty (*Gestoert*) or undefined (*ZustandUnbekannt*). The latter is used to describe the state of the barrier when invoked to open or close but not yet finished. The initial state is as usual the state *Uninitialisiert*. Contrary to the descriptions of the subsystem-components above, no build in self-test is implemented in the subsystem-component and can therefore be executed. It is sufficient that the barrier is brought to an initial (open) position, by communicating the signal *SchrankeInit*. As depicted in Sec. 12.2.1, the opening of the barrier must take place in a time frame of six seconds. The time frame to open the barrier is validated by a timer implemented inside the state *ZustandUndefniert*. If the elapsed time is greater than the defined time frame, a change of state to *Gestoert* is enforced (transition labeled with *NichtImZeitfenster*). Otherwise, a sensor (implemented as a hidden input interface) indicates the complete opening of the barrier,

enforcing a change of state to *Geoeffnet*. The barrier can be controlled from outside by communicating the signals *SchrankeSchliessen* to close the barrier, and the signal *SchrankeOeffnen* to open it.

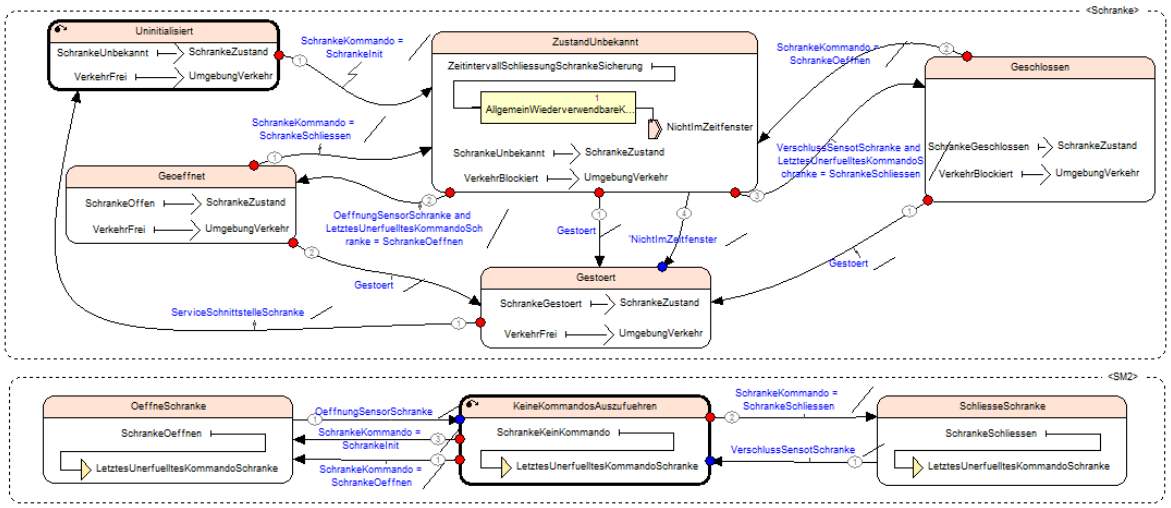

Figure 12.38: Implementation model of the barrier.

Every time the barrier is invoked to be closed or to be opened, a state change to *ZustandUndefiniert* is performed and the implemented timer to observe the defined time frame is restarted. At the end, either the sensors of the barrier acknowledge the opened or closed position inside the defined time frame and perform a corresponding change of state to *Geoeffnet* or *Geschlossen*. Or, due to elapsed time, a change of state to *Gestoert* is triggered, which results in emitting an error message *SchrankeGestoert* to outside.

The SCADE state machine <*SM2*> has been implemented to ensure that dates of the right sensor are evaluated. If the barrier has been prompted to be closed, it has to be ensured that the sensor validating the closure is validated and not the sensor for the opening of the barrier.

12.2.4 Model-checking GALS Systems on the Source Code Level

The top-goal of project VerSyKo is to make model-checking feasible and applicable for industrial projects using GALS architectures. The VerSyKo approach is to make abstractions of synchronous components and the asynchronous system architecture in order to manage the problem of state space explosion.

Control applications in the railway domain can often be separated in a system control function and a safety monitoring function. Those safety monitoring functions usually consist of low-complexity algorithms, i. e. only few states. The level-crossing systems is an example of such a rather low-complexity system. In order to evaluate the handling of GALS systems, we wanted to test whether for such low-complexity systems it might be feasible to perform model-checking directly using the SCADE models. Hence, industrial practioners could immediately benefit from using a model-checker for low-complexity verification problems without further time- and cost-intensive modification or search for suitable abstractions.

The following section reports on experience we have gained by model-checking GALS systems and directly using the source code generated from SCADE models. The experiments have been conducted partially in parallel to the conceptual work of sections 5 to 6 in order to understand the practical

implications of the GALS problems in an industrial case study. The experiments served as an early feed-back and experience and results contributed to the conceptual work of VerSyKo.

At first we will give a description of our approach for model-checking on the source code level using SPIN and SCADE. Then follows a short introduction of how to integrate C-code into PROMELA models for the SPIN model checker. We will discuss modeling of synchronous components, the asynchronous environment, communication and scheduling. Thereafter, we will describe the technical details of integrating SCADE models into SPIN verification. The next part deals with experiments using the level crossing or parts thereof. The last part of this section summarizes the outcome of the experiments and explores possible problems of this direct source-code based approach.

Code-level Verification of GALS Systems

Code-level verification means to apply model-checking directly on the source-code which will be used for the production system rather than to take an abstraction of the system's behavior. This approach implicates several main advantages: (1) the code can be taken as is and does not require any modification, (2) which guarantees the relevance of the verification results for the target system, and (3) therefore raises the value for the assurance and certification of systems, especially for safety relevant applications. Several approaches particularly for model-checking of SCADE-models integrated with SPIN or other model-checkers have been discussed in the Related Work section 1.3.

For the following experiments we have chosen to follow a procedure similar as in section 7.2.1:

1. The local synchronous components are defined as SCADE models. The SCADE code generator is used to produce C-code from the components. The model of the level crossing control serves as a running example.

2. SPIN allows to integrate C-code directly in PROMELA models which is automatically woven into the generated model-checker. For each generated component a wrapper component in PROMELA has to be provided which calls the generated C-code.

3. All components are integrated to a GALS architecture which includes a model of communication, scheduling of components and also a model of the environment.

4. The verification goals for the integrated GALS system are stated as LTL formulas. The LTL formulas are derived from the User Requirements Specification which covers the required behavior as seen from the external system interfaces. Additionally, scenarios can be formulated to check whether the system behaves as expected.

PROMELA-Wrapper for Scade-Models

The purpose of the PROMELA wrapper is to provide a pattern for systematical integration of C-code generated from SCADE models. For each instance of a component in the system architecture, an instance of PROMELA's `proctype` has to be defined (similar to Section 7.2.1).

The code generator from SCADE produces several artifacts from the model. One artifact is a structure describing the state variables of a component which also includes the outputs of a component. The structure's name carries the prefix "outC_" followed by the operator name (here 'outC_UeberwachungsSignal').

At first, the wrapper includes the declaration of component. This is realized using the `c_decl` statement in PROMELA.

```
c_decl {
  #include <UeberwachungsSignal.h>
}
```

Listing 12.6: Inclusion of Scade operator declations

Next, the state variables of the generated component have to be declared. The `c_state` statement includes a variable of type "outC_UeberwachungsSignal" in the state vector of the generated model-checker. The variable is declared to be a global variable of the PROMELA model. In C-code it can be accessed via `now.UeberwachungsSignal_ctx` where `now` allows to access all global variables of the current state of the model-checker.

```
c_state "outC_UeberwachungsSignal UeberwachungsSignal_ctx"
```

Listing 12.7: Declaration of state variables

The second and third artifacts produced by the SCADE code generator are C-functions which implement the dataflow processing of the SCADE operator representing the synchronous component.

The first function is named like the operator and is prefixed with "_reset" ('UeberwachungsSignal_reset'). The "Reset" function expects the state variable as an argument and sets the variables and state of the operator to the initial dataflow values.

The second function is named like the operator (here 'Ueberwachungssignal') and implements the synchronous dataflow behavior, respectively the state transition function. It expects the list of input variables of the operator as first arguments and the state variable as its last argument. The generated function only operates on its state variables. Hence, it can be integrated without modification into PROMELA models using the `c_code` expression.

```
proctype PUeberwachungsSignal()
{
  c_code{
    UeberwachungsSignal_reset(&now.uebSig_ctx);
    UeberwachungsSignal(
                    now.strSig1.Zustand1,
                    now.strSig1.Zustand1,
                    now.strSig1.Zustand1,
                    now.strSig1.Zustand1,
                    now.uebSigKommando,
                    &now.uebSig_ctx);
  };
  do
```

Listing 12.8: Initialization of the state variables

During the first execution step of the wrapper process, the state variable is reset to the initial values defined in the SCADE operator. Then the cyclic function of the operator is called once. After that the wrapper cyclically calls the operator's functions. The `c_code` statement guarantees uninterrupted, atomic execution of the C-code inside.

```
  do
  ::c_code{
      UeberwachungsSignal(
                    now.strSig1.Zustand1,
                    now.strSig1.Zustand2,
                    now.strSig1.Zustand3,
```

```
7                       now.strSig1.Zustand4,
8                       now.uebSigKommando,
9                       &now.uebSig_ctx);
10      };
11    od;
12  }
```

Listing 12.9: Cyclic execution of the synchronous model

We have seen that the variables used here are defined in the C-code. PROMELA allows to access and evaluate variables defined in the C-source for use whithin PROMELA-Models via the `c_expr{ c-code }` operator.

Prerequisites

After modeling the synchronous components, one has to specify the asynchronous part of the system architecture. In the beginning of project VerSyKo it was quite unclear what kind of asynchronicity will be regarded and to what extent. During the experiments we found out that we have to take into account and explicitly define the following aspects to model the asynchronous environment of the synchronous components (cf. Sections 5 and 7):

Scheduling policy The scheduling policy defines the order of execution of each component and specifies how to handle the execution of parallel components. Hence, the results of the model-checking strongly depend on the scheduling policy. The least restrictive scheduling policy allows arbitrary execution of processes. SPIN implements an interleaving semantic, i.e. parallelism is simulated via interleaving atomic processing steps of all processes. Without any scheduling policy or fairness constraints SPIN's model-checker will evaluate executions of the system where probably processes never get scheduled. If a property holds for a system, i.e., the corresponding requirement holds, with the least restrictive scheduling policy it will also hold for stronger policies, but little restrictions of scheduling usually implicate a larger state space required to explore.

Time and Timing In order to verify timing properties, it is necessary to implement a notion of time. Regarding the verification goals it has to be decided whether time is specified and checked as dense time or discrete time. Dense time allows to specify timing values in a continous domain. Discrete time has an underlying model of integral time steps. For verification of GALS systems we usually know values like worst-case execution time of the synchronous components. Furthermore project VerSyKo aims at synchronous systems with a fixed cycle time. Therefore, timing for locally synchronous components can be sufficiently modelled with discrete time steps. On the system level, it depends on the asynchronous architecture and on the time dependent properties to be verified.

Communication Within a GALS architecture the communication defines how the synchronous components interact with each other and the environment of the system. In a real, physical system the components are connected via physical media which are often controlled by a logical communication protocol. The more details we model of the communication system the better will be the results of the GALS verification. But more details also imply a larger state space of the overall system which has to be handled.

System environment The system environment comprises a model of the real world beyond the boundaries of the system. It is sufficient to model the aspects of the real world which will

be recognised at the systems interfaces. A good model of the environment can help to reduce the complexity of the verification problem. Often it makes sense to create several models of the environment which address the context of certain requirements to be verified. The assumptions of the environment can also be modeled in terms of the formal requirements, which resembles the assume-guarantee approach for formal verification.

All aspects discussed above influence the verification process, the size of the systems which can be handled and of course the robustness and validity and universality of the verification results obtained during model-checking. The paramaters related to the above mentioned aspects can and have to be adopted to the needs of a specific verification problem.

Model of the Asynchronous Environment

As mentionened in the introduction of this section, our goal was to initially gain some practical experience and evaluating the parameters and limitations of GALS verification. Therefore, we have decided for a minimum realistic asynchronous environment.

Scheduling: Although properties sucessfully verified with a least restrictive environment, the no-scheduling assumption seems to be too unrealistic. Furthermore, our own experiments showed that too little assumptions can produce lots of false positives resulting from the fact that there exist schedulings in which a component will never or extremely rarely be scheduled.

Because of the asynchronous nature of an asynchronous architecture, the correctness of a GALS system could depend on the order of execution respectively order of incoming or outgoing messages transmitted between components. A "non-deterministic round-robin scheduling" fulfills our needs to model the uncertainty of when components read their inputs or set their outputs. Non-deterministic round-robin scheduling means that there exists a cycle where all synchronous components will be scheduled and the order of execution of each model is determined non-deterministically.

```
1  proctype PScheduler(){
2    do
3    ::SCH_PUmgebung=1;
4      (SCH_PUmgebung==0);
5      (SCH_PUeberwachungsSignal==0
6        && SCH_PStrassenSignal_1==0
7        && SCH_PSimBUESt==0
8        );
9      d_step{
10       SCH_PUeberwachungsSignal=1;
11       SCH_PStrassenSignal_1=1;
12       SCH_PSimBUESt=1;
13     };
14   od;
15 }
16 proctype PUeberwachungssignal(){
17   do
18   ::SCH_PUeberwachungsSignal==1;
19     /* execute Scade model */
20     SCH_PUeberwachungsSignal=0;
21   od;
22 }
```

Listing 12.10: A non-deterministic round robin scheduler

The listing above shows the PROMELA implementation of the non-deterministic round-robin scheduler. For each process exists a flag—prefixed with "SCH_". The scheduler synchronizes on

all flags and then atomically schedules all processes for execution. The order of execution of the processes is determined by the interleaving semantic of SPIN. A special process is the model of the environment "PUmgebung" which is always executed in the beginning of a cycle.

Time: For our first experiments we did not explicitely model a concept of time, since there was no need to verify time-dependent requirements. Nevertheless, the round-robin scheduling implies a rather synchronous model of discrete time which would allow to specify timing properties on the base of a scheduling cycle.

Communication: Connections and communication among the synchronous components whithin the GALS architecture are modeled via shared variables. Shared variables provide a minimum model of communication infrastructure compared to real world communication which often implies bus systems (e.g. CAN) with dedicated arbitration strategies.

A process accesses the shared variables in the beginning and at the end of its cycle. The read and write accesses will always be performed atomically. Situations, where one process reads an inconsistent datum from a second component, represented by two or more shared variables are not modeled here. Because of the selected scheduling policy consistency of input values resulting from more than one component is not guaranteed, where consistency means that all input values of a component result from the same scheduling cycle.

System environment: The basic usage scenario of the level crossing is where a train passes the level crossing. Arrival, passage of the crossing area and leaving the level crossing are modeled to consume some time because a realistic passage also does. This consumption of time has been introduced to give the synchronous components some time (scheduling cycles) to react.

```
1  proctype PUmgebung (){
2    do
3      :: SCH_PUmgebung; SCH_PUmgebung--;
4      :: break;
5    od;
6    SCH_PUmgebung;
7    umgebungZug=cZugAbwesend;
8    SCH_PUmgebung--;
9    do
10     :: SCH_PUmgebung; SCH_PUmgebung--;
11     :: break;
12   od;
13   SCH_PUmgebung;
14   umgebungZug=cZugVorhanden;
15   SCH_PUmgebung--;
16   do
17     :: SCH_PUmgebung; SCH_PUmgebung--;
18     :: break;
19   od;
20   SCH_PUmgebung;
21   umgebungZug=cZugAbwesend;
22   SCH_PUmgebung--;
23   do
24     :: SCH_PUmgebung; SCH_PUmgebung--;
25   od;
26 }
```

Listing 12.11: System environment: a train passes the level crossing

Verification goals

The verification goals for the level crossing case study are stated as LTL-formulas. We have divided the verification goals in two sets: (1) LTL-formulas from User Requirements and (2) usage scenarios to query additional properties of the system.

The first source of verification goals is the User Requirements Specification which describes the intended behavior of the system. The given URS specifies the behavior as it can be observed at the system boundaries.

Most of the requirements of the level crossing system are formulated as a scheme of conditions and desired reactions of the system. Therefore, most of the requirements can be modeled similar to the listing of UR03 below: *Three seconds*[14] *after the set of traffic lights shows the yellow light the systems shall turn on the red light and turn off the yellow light.*

```
1 ltl UR03_Stoppen_des_Strassenverkehrs {
2   always((umgebungKfzX==cKfzRotAusGelbAn) ->
3     ((umgebungKfzX==cKfzRotAusGelbAn)
4     until (umgebungKfzX==cKfzRotAnGelbAus) ) )
5 }
```

Listing 12.12: Stopping the road traffic

In order to get a coarse validation that the systems works as intended, we have also specified some queries which represent typical scenarios, i.e. use cases of the system. The following listing shows a LTL-formula which shall prove that the train will eventually be signalled permission to approach the level crossing.

```
1 ltl Scenario_Freigabe {
2   always ((umgebungZug==cZugVorhanden) ->
3     ((umgebungZug==cZugVorhanden && umgebungTfZug==cTfZugHaltbegriff)
4     until (umgebungZug==cZugVorhanden && umgebungTfZug==cTfZugFahrtbegriff)))
5 }
```

Listing 12.13: Scenario: train gets permission to approach

The goal for the experiments was to use LTL-formulas of the first User Requirements UR01-UR03 and several scenarios which were created in an ad-hoc manner including the above scenario.

First experiment: Verifiying the complete model

In a first attempt we wanted to apply the verification to the complete GALS model of the level crossing case study. At first the system architecture was modeled in PROMELA according to the procedure described above. The resulting PROMELA model consists of 13 processes: 9 SCADE components, the environment, the scheduler, the LTL-Büchi-automaton, and SPIN's init process. Fig. 12.39 shows the automaton representation of the PROMELA process of the level crossing core controller.

The model-checker was generated with options: partial order reduction, graph encoding set to size of state vector, state compression. The Version of SPIN used was `6.1.0 -- 4 May 2011`. The verfication was executed on an `Intel Core i5 CPU M 560` with 2 cores running at 2.67GHz with 4GByte main memory under MS Windows 7 (32-bit). The process of the generated pan model-checker occupied 100% computation time on one of the cores.

Results: The size of the state vector was 1460 bytes. The average memory consumption was about 30 MBytes. After 3 days of continuous operation the experiment was aborted. Until that day a verification depth of 75 was reached. The model-checker did not yield any error-trace.

[14] Timing is not considered here; only the order of execution.

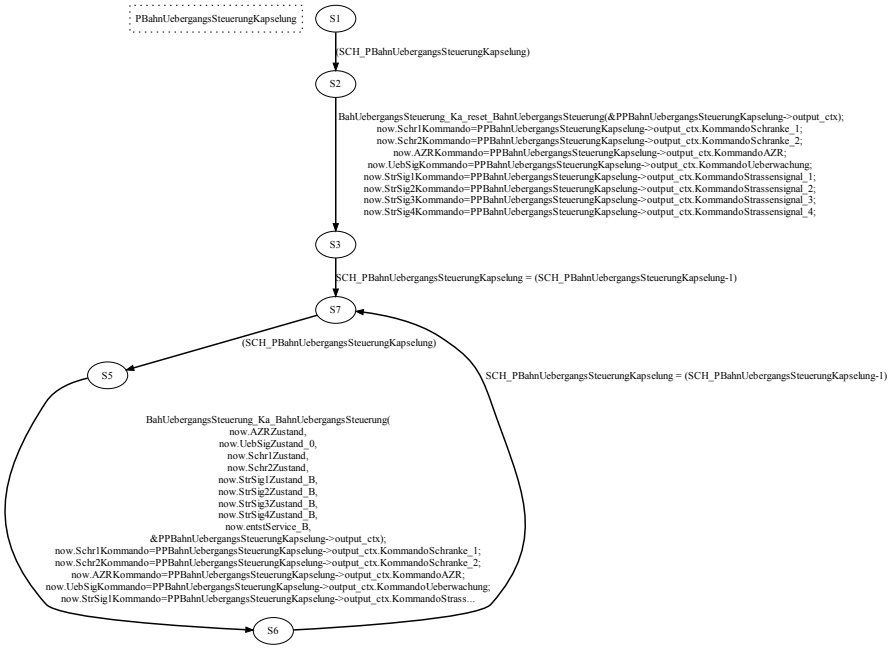

Figure 12.39: Automaton representation of the core controller generated by PAN

Analysis: Although, the average memory consumption was rather low it seems quite clear that the size of the state vector (1460 Bytes) and the reachable state space which has to be explored is too large. One reason might be the size of the original model which implements the User Requirements. A closer look on the code generated by SCADE's KCG reveales the resulting C-code is not optimized for memory size and, of course, not optimized to serve as input for model-checking:

- SPIN represents all boolean and bit variables with exactly one bit in the state vector. SCADE's code generator generates `int` variables[15] (32 bits) in C to represent boolean values.
- The KCG code generator produces the C-functions for each type of operator instantiated in the SCADE model. If an operator T is composed of nested operator instances S_i, the generated state structure of operator T consequently contains variables of the state structures of the sub-operators S_i. If a nested operator produces an output which is directly routed to the output of the top-level, KCG produces a copy of the output variable for each nesting-level.

On the one hand, the first experiment could not be sufficiently finished, but on the other hand it helped to understand more of the difficulties in the application of formal methods on industrial problems.

Second experiment: A minimal model

After the negative results of the first experiment, we decided to make a second attempt of source-level model-checking. This time the size of the state vector should be reduced. Therefore, a minimal sub-model of the level crossing case study should be used for model-checking.

[15] could be reduced to uint8 (8 bits)

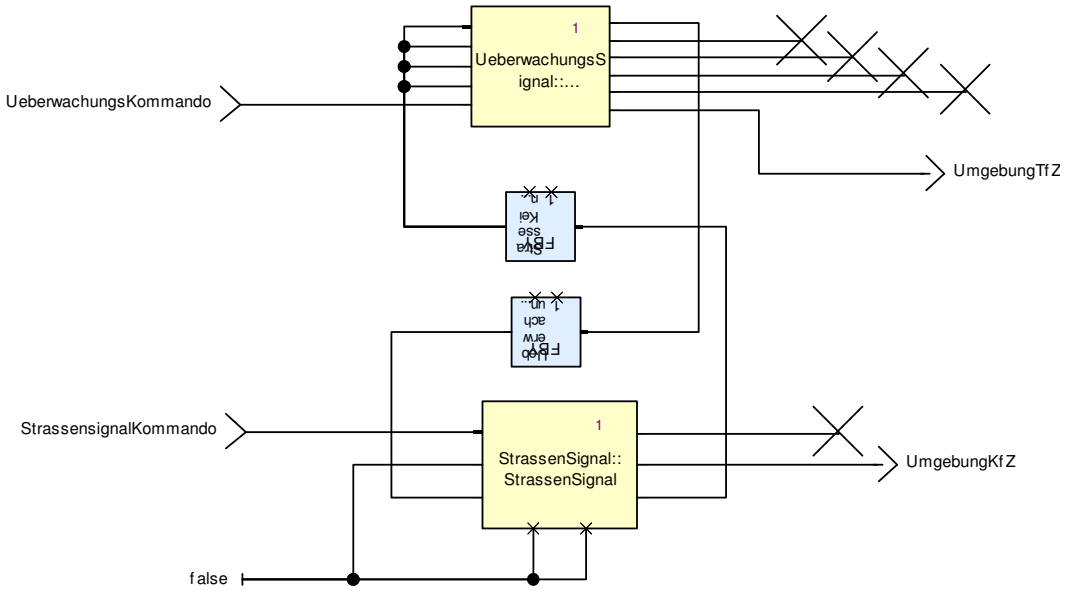

Figure 12.40: A minimal sub-model of the level crossing

The chosen sub-model consists of the traffic lights (‘StrassenSignal’) and the supervision signal (‘UeberwachungsSignal’). Both components implement only a limited stateful behavior and they interact with each other. That means that the size of the state vector should get smaller and the minimal example also represents a small part of the asynchronous architecture (see Fig. 12.40).

Since the model was changed, another verification goal had to be chosen. The User Requirements for both components require the traffic lights and the supervision signal to observe each other such that not both road traffic and trains may be signalled permission to approach the level crossing. One direction of the requirement can be formalized as follows:

```
1 ltl Scenario_Niemals_Ueberwachungssignal_und_nicht_KfzRot {
2   always (umgebungTfZug==cTfZugFahrtbegriff ->
3     umgebungKfzABr==cKfzRotAnGelbAus)
4 }
```

Listing 12.14: LTL-formula: train permission to approach implies road vehicles must stop

Both components ‘StrassenSignal’ and ‘UeberwachungsSignal’ do not receive inputs from the environment but are controlled by the internal command inputs ‘UeberwachungsKommando’ and ‘StrassensignalKommando’. Hence, another environment for model-checking had to be chosen. Since the above mentioned verification goal is a safety requirement, the property should hold in all cases independent of the issued commands to the components. The command inputs are stimulated with a non-deterministic choice of commands:

```
1 proctype PSimBUESt(){
2 do
3   ::SCH_PSimBUESt; atomic{
4     if
5       ::uebSigKommando=1;
6       ::uebSigKommando=2;
7       ::uebSigKommando=3;
```

```
8           :: uebSigKommando=4;
9       fi;
10      if
11          :: StrSig1Kommando=0;
12          :: StrSig1Kommando=1;
13          :: StrSig1Kommando=3;
14          :: StrSig1Kommando=4;
15      fi;
16    }; SCH_PSimBUESt=0;
17  od
18  }
```

Listing 12.15: Non-determinstic stimulation of the command inputs

Results: The model-checker was generated with the same options as above. The size of the resulting state vector is 148 bytes. The model-checking took only a few seconds and terminated with 0 errors.

Conclusion

Verification of GALS systems directly on the source code level remains an attractive option for industrial application. Today, tools like SPIN for the asynchronous world and SCADE for synchronous systems that implement formal methods are available and in a mature state. During the last section we have evaluated model-checking of GALS systems. Source code generated directly from synchronous SCADE component models (without any abstraction) are combined with an asynchronous architecture in PROMELA (SPIN) and verified with SPIN against requirements stated as LTL-formulas.

However, the experiments have shown that the source level approach without abstraction is not yet feasible even for low-complexity GALS systems as the level crossing case study. Only a minimized sub-model of the case study could be checked against its requirements. The main reason was that the generated code from SCADE is not optimized for model checking:

- code generation often inefficient
- state space explosion: e.g. boolean variables as int, etc
- inefficient state representation

Although, it was not possible to verify the integrated level crossing system, the outcome of the previous experiments was not disappointing. (1) The experiments allowed to gain experience with all aspects of GALS verification (synchronous and asynchronous modeling, scheduling, communication, time, verification goals) and provides us with important questions to be answerded within project VerSyKo. (2) That source code model-checking is not yet feasible for our case study, gives additional motivation for the VerSyKo approach to use contracts as abstractions in order to handle industrial size model-checking problems.

12.3 Turn indicator

12.4 Engine-Start-Stop-Automatic

Chapter 13

Validation of developed methods and comparison of model checking methods

13.1 CDLS Descriptions of Level Crossing Case Study

13.2 Contracts and Verification of the Level Crosssing System

13.3 Benchmark of SPIN and UPPAAL Targets

This section reports on benchmark measurements performed for the alternative model-checking backends of the contract specification language GTL.

The considered backends are the model-checking tools SPIN [59] and UPPAAL [67]. The objective of this document is to compare their performance with respect to time and memory consumption for equivalent inputs.

More details on measurement process and input data can be found in the whitepaper [75].

Tool Version Information.
For the GTL tool, version 0.1 (2012-01-09) is used.

For UPPAAL, the 64-bit version 4.1.7 is used.

For SPIN, version 6.1.0 is used.

Test Platform Information.
All run-time tests have been executed on a 2.80GHz Intel Xeon CPU with 24GB of main memory and 12288 KB of cache. The machine (bull) provides multiple CPUs (24), but the tool make use of only one CPU (with 100% load).

Time and memory data has been retrieved with the system utility

```
/usr/bin/time
```

`time` version: GNU time 1.7.

13.3.1 The Benchmark Example: Client-Server Mutex

SPIN is a LTL model checker, while UPPAAL treats a (small) subset of timed CTL (TCTL). Therefore

the tools are incomparable in general. However, the common subset of these concepts includes *safety conditions* (also known as "*never claims*"). We will use this subset for our analysis.

Criteria for Selecting the Example

When selecting a benchmark example, the following criteria should be fulfilled.

(A) *The example should be easy to comprehend.*

Rationale: It should be possible for a human inspector to determine whether a given input represents a valid instance of the example. Otherwise, the generated input data could not be trusted.

(B) *The example should be scalable with respect to size, i.e. provide parametrized instances with increasing state space.*

Rationale: For model-checking, it is common that "small" inputs can be processed without meaningful time/memory consumption and "large" inputs yield out-of-memory conditions. To allow for a meaningful comparison, we would like to gradually increase the input size (i.e. size of the state space), until we reach the limits of feasibility or user patience.

(C) *The example should provide a non-trivial safety property (that holds).*

Rationale: A meaningful comparison of tools requires comparable tasks they perform. Model-checking a safety-condition (successfully) corresponds to a statement with respect to the complete (reachable) state space; since the input data is equivalent, so is their state space.

Client-Server Mutex Protocol

We found an example that meets all the above criteria with a Client-Server Mutex protocol.

In general, Mutex (or *mutual exclusion*) is a property of parallel processes that compete for a shared resource. The resource can be allocated to at most one process, thus a protocol has to be established that (I) allows in principle every process to get the resource and (II) prevents two (or more) processes from using the resource at the same time.

A process that got hold of the resource is said to be in the *critical section*. (I) is a *fairness condition* ("something good will eventually happen to everyone") on the protocol, while (II) is a *safety condition* ("nothing bad will happen").

There are various mutex protocols established in the literature (see, e.g.,[70]). The *Client-Server Mutex* is a variation, where N clients may request access to the critical section from a single server. A client only enters the critical section, if the request is granted. After leaving the critical section, the client notifies the server.

The server

- keeps track of who got granted access to the critical section
- and
  - (i) if a client is in the critical section, nobody else is granted access
  - (ii) otherwise, the server grants access to exactly one client (selected from the list of applicants for the critical section)

This protocol is simple (*criterion (A)*) and allows for an arbitrary (fixed) number of clients (*criterion (B)*).

Moreover, the safety condition "at most one client in the critical section" is a purely combinatorial property, i.e., no clock is involved. Thus it can be expressed equivalently both in SPIN and in UPPAAL. Since the property is true (by inspection), the complete state space has to be analyzed by the corresponding tool (*criterion (C)*).

13.3.2 Generation of Input Data

This chapter explains how the input data for the model-checking tools have been generated. First a GTL formulation for sequence of clients N has been generated. This has been transformed by the GTL utility to the equivalent SPIN or UPPAAL representation.

Generation of GTL formulation

The Client-Server Mutex sketched in Section 13.3.1 can be formulated very systematically in GTL.

The generation here has been done via an AWK-Script. The script takes the number N of clients as command line input and prints the GTL-Syntax to `<stdout>`.

Proper operation has been validated by (textually) comparing the output of $N = 3$ against the manual formulation, as it has been developed by Technical University Braunschweig.

Generation of SPIN/PROMELA input data

The input language for the SPIN tool is PROMELA, see [59].

The PROMELA-files were generated by the GTL tool via processing of `mutex_<N>.glt`

```
gtl  mutex_<N>.gtl
```

This yields a `mutex_<N>.pr` file, which contains a LTL-formulation of the safety condition.

Note. The above call to GTL-0.1 (2012-01-09) does not only generate the `*.pr` file, but also processes it with SPIN and runs the corresponding `verifier`. For the measurements, the `verifier` were aborted and re-compiled on the measurement machine (based on the corresponding `*.pr` file).

Generation of UPPAAL input data

The UPPAAL tool operates on `*.xml` files (following a UPPAAL specific document-type definition), see [67].

The `*.xml` files were generated by the GTL tool via processing of `mutex_<N>.glt`:

```
gtl  -m uppaal  mutex_<N>.gtl
```

This yields a `mutex_<N>.xml` file.

UPPAAL expects model-checking queries (here: the safety condition) to be stored in a separate *query* file, `*.q`.

In version 0.1 (2012-01-09), the GTL tool is not capable of generating this file automatically, since in general not every LTL formula *can* be expressed in the UPPAAL query language.[1]

Therefore, the query files `mutex_<N>.q` have been generated together with the `*.gtl` files by the AWK-Script.

[1] A subset of TCTL.

13.3.3 The Measurement Process

SPIN-6.1.0 and UPPAAL-4.1.7 have been installed on a high-performance machine (`bull`), which provides 2.80GHz Intel Xeon CPU with 24GB of main memory and 12288 KB of cache. The operating system is Linux CentOS release 5.7.

The machine has *not* been reserved exclusively for the benchmark testing; however, there were no processes with substantial memory consumption present during the measurement. The time measurement refers to user time, i.e. the total number of CPU-seconds that the process spent in user mode.[2] The memory measured is the *optimistic* allocation that the Linux kernel allows, i.e. the memory the process *might* make use of at the point of highest memory load.[3]

The UPPAAL tool can directly operate on the `*.xml` and `*.q` files. The command line invocation looks like this:

```
verifyta   mutex_<N>.xml mutex_<N>.q
```

SPIN/PROMELA, the `*.pr` files were transformed to a `verifier` executable, before the measurement was started; this happened in time well below a second and can be neglected. The command line invocation then looks like this:

```
./mutex_<N>-verifier
```

The time and memory information have been recorded by means of the script `measure.bash`, which makes use of the system utility `/usr/bin/time`.

All verification processes were run in sequence (to exclude interaction). Every verification process recorded its options and results (time, memory, verification outcome) to a file. The graphical representation has been derived from these files by means of of the `gnuplot`.

Selection of Compile-Time / Run-Time Options

The tools SPIN and UPPAAL allow for various user options to modify (and hopefully speed-up) the verification of a given model. SPIN also allows *compile-time options*, since the verifier is generated by compiling C-file `pan.c` (here: with the gcc).

For UPPAAL, there are no compile-time options available (to the user); it should be noted, however, that a 64-bit executable of UPPAAL is used, since this allows addressing of more than 4GB of memory.

The following selection of (combinations of) tool options have been made.

[2] Comparison of user time/real time shows, that effectively one CPU exclusively executed the model-checking process; in presence of 24 CPUs, this is not surprising.

[3] This explains why some numbers exceed the available 24GB of main memory without using swap.

SPIN option combinations used:

Compile Time	Run Time
	`-a`
	`-m999k`
	`-m9999999k -a`
`-DSAFETY`	`-m9999999k`
`-O2`	`-m9999999k -a`
`-O2 -DSAFETY`	`-m9999999k`
`-O3`	`-m9999999k -a`
`-O3 -DSAFETY`	`-m9999999k`

UPPAAL option combinations used:

Compile Time	Run Time
(N/A)	`-S 0`
(N/A)	`-S 1`
(N/A)	`-S 2`
(N/A)	`-S 2 -A`
(N/A)	`-S 0 -C`
(N/A)	`-S 1 -C`
(N/A)	`-S 2 -C`
(N/A)	`-S 2 -Z`
(N/A)	`-S 2 -n 0`
(N/A)	`-S 2 -n 1`
(N/A)	`-S 2 -n 2`
(N/A)	`-S 2 -n 3`
(N/A)	`-S 2 -n 4`

Tabular Presentation of Results

This section lists the measurements in a tabular manner. The digits after the second fractional digit (i.e. everything after 0.00) have been truncated to meet width conditions.

Notes on unreliable executions.

(*) Here, the SPIN results are unreliable (i.e., do not give proof), due to option `-m999k`:

The `mutex_*-verifier` output shows the line

```
error:  max search depth too small
```

(‡) Here, the UPPAAL results are unreliable (i.e., do not give proof), due to option `-Z` (bit state hashing):

The `verifyta` output shows the line

```
-- Property MAY be satisfied.
```

These are *not* tool defects. For some run-time options the results are inexact in one direction; if a violation would have been found, this would have been reliable.

For a description of the tool options refer to the Appendix in [75].

Compile	Run	2	3	4	5	6	7	8	9	10	11	12
		0.00	0.02	0.15	1.65	9.77	46.82	304.10				(*)
	`-a`	0.00	0.03	0.32	2.06	10.88	53.07	344.71				(*)
	`-m999k`	0.00	0.02	0.20	1.34	5.09	37.79	276.30	1307.00			(*)
	`-m9999999k -a`	0.08	0.11	0.36	1.85	11.22	87.68	606.76				
`-DSAFETY`	`-m9999999k`	0.10	0.10	0.19	1.65	11.01	79.59	532.94				
`-O2`	`-m9999999k -a`	0.17	0.18	0.27	1.20	5.11	38.04	255.48				
`-O2 -DSAFETY`	`-m9999999k`	0.15	0.17	0.26	1.24	4.60	33.45	225.12				
`-O3`	`-m9999999k -a`	0.16	0.17	0.26	1.23	5.20	37.83	256.15				
`-O3 -DSAFETY`	`-m9999999k`	0.13	0.18	0.25	1.19	4.22	33.68	229.37				

Figure 13.1: Time Consumption [s] of SPIN with Various Compile-time and Run-time Options.

Run	2	3	4	5	6	7	8	9	10	11	12
`-S 0`	0.00	0.00	0.02	0.10	0.54	1.82	9.08	47.43	247.09	1261.18	
`-S 1`	0.00	0.00	0.02	0.10	0.55	1.89	8.90	48.59	252.34	1296.53	
`-S 2`	0.00	0.00	0.02	0.11	0.63	2.07	10.31	56.68	295.80	1483.89	
`-S 2 -A`	0.00	0.01	0.02	0.10	0.61	2.06	10.37	54.98	289.80	1490.53	
`-S 0 -C`	0.00	0.00	0.02	0.10	0.52	1.82	8.77	46.51	242.52	1259.12	
`-S 1 -C`	0.00	0.00	0.02	0.10	0.53	1.92	8.97	47.16	245.69	1276.53	
`-S 2 -C`	0.00	0.00	0.02	0.11	0.58	1.76	10.31	54.71	286.98	1470.58	
`-S 2 -Z`	0.01	0.02	0.03	0.13	0.67	2.23	11.53	60.11	302.63	1316.19	3806.93 (‡)
`-S 2 -n 0`	0.00	0.00	0.02	0.05	0.62	2.07	10.26	55.93	292.92	1476.42	
`-S 2 -n 1`	0.00	0.00	0.02	0.11	0.59	2.07	9.93	53.35	283.26	1418.98	
`-S 2 -n 2`	0.00	0.01	0.02	0.12	0.66	2.01	10.91	59.51	310.42	1577.45	
`-S 2 -n 3`	0.00	0.00	0.02	0.11	0.56	1.83	10.74	56.23	296.44	1492.76	
`-S 2 -n 4`	0.00	0.00	0.02	0.11	0.63	2.16	10.75	56.04	295.61	1485.28	

Figure 13.2: Time Consumption [s] of UPPAAL with Various Run-time Options.

Compile	Run	2	3	4	5	6	7	8	9	10	11	12
		20.90	22.25	32.18	100.62	566.45	2482.18	15477.00				(*)
	`-a`	20.90	22.25	32.18	100.62	566.46	2482.18	15476.98				(*)
	`-m999k`	19.01	20.32	27.90	71.93	321.59	2061.00	16182.60	64105.17			(*)
	`-m9999999k -a`	2155.03	2156.37	2166.32	2234.75	2718.60	6189.29	27875.21				
`-DSAFETY`	`-m9999999k`	2155.01	2156.34	2165.48	2234.65	2718.57	6188.85	27875.20				
`-O2`	`-m9999999k -a`	2154.93	2156.25	2166.17	2234.57	2718.42	6189.06	27874.93				
`-O2 -DSAFETY`	`-m9999999k`	2154.92	2156.23	2165.37	2234.50	2718.39	6188.62	27874.92				
`-O3`	`-m9999999k -a`	2155.01	2156.34	2166.28	2234.70	2718.56	6189.20	27875.10				
`-O3 -DSAFETY`	`-m9999999k`	2155.00	2156.32	2165.48	2234.62	2718.56	6188.78	27875.09				

Figure 13.3: Memory Allocation [MB] of SPIN with Various Compile-time and Run-time Options.

Run	2	3	4	5	6	7	8	9	10	11	12
`-S 0`	15.60	16.20	17.28	20.10	31.85	81.98	307.95	1287.21	5489.53	23505.53	
`-S 1`	15.56	16.15	17.21	19.96	31.78	81.93	307.92	1287.12	5489.50	23505.54	
`-S 2`	15.56	16.12	17.23	19.96	31.84	82.21	309.35	1292.79	5515.70	23612.65	
`-S 0 -A`	15.57	16.20	17.43	20.87	34.81	95.32	358.09	1499.48	6359.03	27396.53	
`-S 1 -A`	15.53	16.14	17.39	20.85	34.79	95.29	358.09	1499.50	6358.95	27396.57	
`-S 2 -A`	15.50	16.17	17.40	20.78	34.82	95.60	359.43	1505.17	6385.17	27503.62	
`-S 2 -C`	15.50	16.18	17.40	20.78	34.84	95.56	359.48	1505.15	6385.09	27503.65	
`-S 2 -Z`	143.53	144.18	145.31	148.50	161.70	217.90	460.73	1509.68	5939.50	24886.25	89436.68 (‡)
`-S 2 -n 0`	15.59	16.15	17.21	20.09	31.95	82.26	309.29	1292.82	5515.71	23612.68	
`-S 2 -n 1`	15.46	16.06	17.12	19.92	31.75	82.14	309.17	1292.78	5515.57	23612.48	
`-S 2 -n 2`	15.56	16.20	17.25	20.04	31.92	82.25	309.40	1292.89	5515.70	23612.59	
`-S 2 -n 3`	15.57	16.14	17.25	19.96	31.90	82.26	309.37	1292.87	5515.75	23612.68	
`-S 2 -n 4`	15.51	16.15	17.23	19.95	31.84	82.25	309.32	1292.89	5515.70	23612.62	

Figure 13.4: Memory Allocation [MB] of UPPAAL with Various Run-time Options.

Measurement Plots

Figures 13.5,13.6 display the time and memory consumption with increasing number N of clients. Unmapped N correspond to out-of-memory situations.

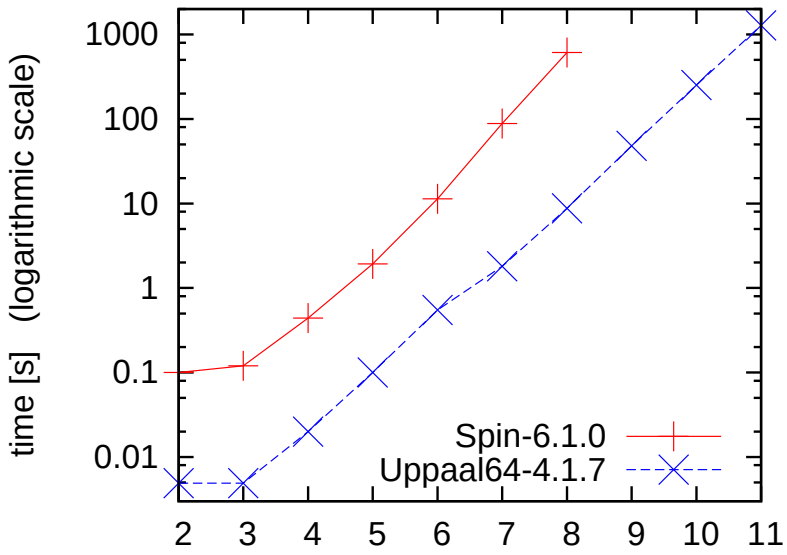

Figure 13.5: Time consumption for exhaustive search for N clients; measured on a 2.80GHz Intel® Xeon® CPU with 24GB of main memory

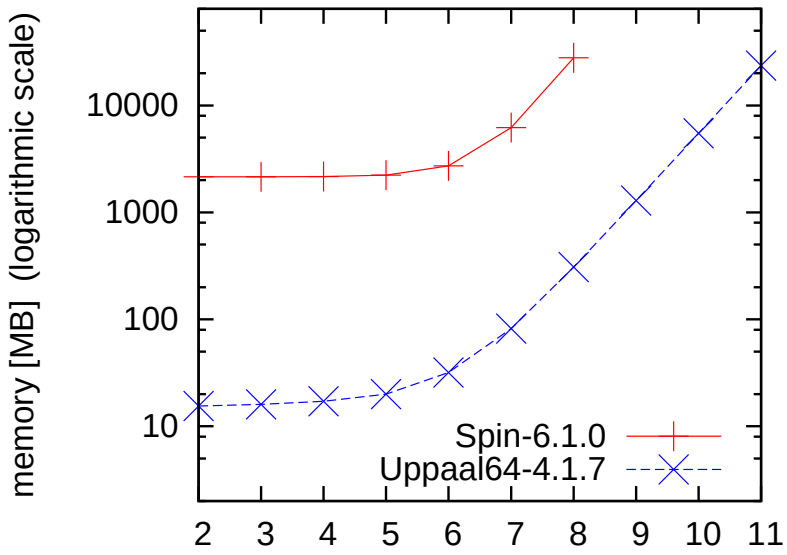

Figure 13.6: Memory consumption for exhaustive search for N clients; measured on a 2.80GHz Intel® Xeon® CPU with 24GB of main memory

13.3.4 Evaluation and Conclusion

In all the samples, time-and memory consumption follow an exponential slope (after some offset). This follows to state-space increment and is hardly surprising.

Comparing the performance of UPPAAL and SPIN as used as a backend *for this specific example*, the following observations can be made.

1. **UPPAAL does perform significantly better than SPIN (here).**

   With respect to run-time, UPPAAL is ≈ 24 times faster when comparing best-to-best, without use of compiler optimizations (`-O2, -O3`) for SPIN the factor would be ≈ 53.[4]

   With respect to memory allocation, UPPAAL uses only $\approx 1/87$.[5]

   With all combinations, UPPAAL was able to process $N = 11$ clients, while SPIN ran out of memory after at most 9 clients.[6]

2. **Compile-Time optimization gives some time-improvements for SPIN.**

   The SPIN verifier is ≈ 2.4 times faster when compiled with optimization. It is mildly surprising that `-O2` actually performs a bit *better* than `-O3`. This is—of course—dependent on the used `C` compiler.

   Note that the optimizations do essentially not affect the *memory* consumption. Memory remains the limiting factor.

3. **No Run-Time option gives significant improvements (here).**

   The best and worst were all within a factor of 2.

   Apparently the symmetry of the N client machines cannot be exploited in a significant way (by the tried options).

A positive observation on the side is that all successful runs completed within the hour.[7] This means that (for 24GB of main memory) there is an acceptable time, after which we can stop waiting.

[4] The factor is derived as

$$\frac{\Sigma_{N=2..8}(\text{time of successful SPIN runs})/(\text{number of successful SPIN runs})}{\Sigma_{N=2..8}(\text{time of successful UPPAAL runs})/(\text{number of successful UPPAAL runs})}$$

[5] This comparison is somewhat unfair, since run-time option `-m9999999k` forces SPIN to allocate a big hash-table even for small value of N. Adjusting this option to "just fit" the model size would yield better result for SPIN here.

[6] $N = 9$ completes for the SPIN execution with run-time option `-m999k`; the outcome is unreliable due to limited search depth. Reliable computations are possible up to 8 clients.

[7] The unsuccessful (aborted) runs are not displayed in the plots or tables. The longest observed run lasted for 4.4 hours, before it got killed due to out-of-memory condition.

13.4 Model Based Testing

Chapter 14

Lessons Learned and Summary

Chapter 15

Summary and Future Work

In this section we will summarize the current status of the VerSyKo project and outline the further steps to be taken within this project and beyond.

The results planned for AP 1 and AP 7 have all been reached in the allocated time:

- The definition of one user level domain specific language for specifying GALS systems—the CDSL (see Section 4). This provides in particular a user friendly specification environment supporting intuitive specifications of component behavior using state machines and descriptive black box specifications using LTL. It also integrates guaranteed behavior of components as a way of annotating component specifications with previously verified properties or behavior.
- Prototypical tool support for specification in CDSL; indeed, CDSL is realized as a UML-profile, and modeling is supported by the tool Enterprise Architect [69].
- A stable textual core specification language—the GALS translation language GTL (see Section 5). It allows for the specification of GALS systems as a network of synchronous components each of which is abstractly described by one or several contracts in the form of LTL formulas or automata. Again, component models are annotated by guaranteed behavior.
- Concepts for transformations from CDSL to GTL (see Section 4.5 and from there to analysis tools for model checking (see Section 7).
- Two industrial case studies; the smoke detection system described in Section 12.1 and the level crossing system described in Section 12.2. The first of these case studies now provides a GALS system with specified contracts of all components, where as the second case study currently provides the (implementation level) SCADE models of all synchronous components.

In addition to these planned works we also provided a first prototype of the translation tool GTL $\rightarrow$ PROMELA. That the first verification experiments with this prototype still show performance problems comes as no surprise. The implemented algorithm for translation is not optimized yet. However, first experiments we currently are performing with an optimized version look very promising.

15.1 Further work

The next tasks scheduled according to the project plan have already been started or are starting. Other tasks also follow as planned. The next steps are as follows:

- a model transformation GTL → PROMELA (AP 2); here and in the next point it will be particularly important to employ optimization techniques such as removal of redundant transitions/states or symmetry reduction (in the presence of many instances of the same component) to fight complexity issues
- a model transformation to GTL → UPPAAL (AP 3; from Dec. 2011)
- local verification of contracts for components using bounded model checking (AP 4)

In addition, the two case studies will be expanded as needed for experiments evaluating the work of the above work packages. In particular, as soon as first optimizations for the model transformation GTL → PROMELA have been implemented experiments will be performed with the manually coded GTL specification for the cabin smoke detection system (see Listing A.1) and with a GTL specification of the level crossing system. Later we will use the GTL specifications generated from a GALS model specified with the CDSL in Enterprise Architect. The following two tasks will expand the two case studies:

- For the level crossing system a user level DSL specification (e. g. in CDSL) will be specified during the course of the project, and this specification will be transformed into GTL.
- For evaluation of the work on local model checking the implementations as synchronous models of the components from the smoke cabin case study will be needed. These models will be produced, and the local verification of the contracts of each component on the corresponding implementation model will be performed.

A new idea within VerSyKo is to use SMT ("Satisfiability Modulo Theory") solving and bounded model checking not only for local verification but also globally, at least to produce counterexamples in the formal verification. So besides the model transformations GTL → PROMELA and GTL → UPPAAL we envision a third transformation from GTL to the input language of an SMT-solver. This will further help to mitigate the project risk that formal verification using one of the model transformations does not scale to case studies of a size relevant for future industrial application.

Verified will provide their SMT-solver SONOLAR, which is developed in cooperation with the University of Bremen and which performed very well in recent SMT-COMP'11 [1]

Later we intend to study how to prove *completeness* using the SMT-solver, i. e., to show that a GALS model satisfies a certain verification goal. This will combine SMT-solving with induction and involves to establish an upper bound for the step width of the induction step.

[1] see `http://www.smtcomp.org/2011/`.

Bibliography

[1] Automated verification for train control systems. In: Schnieder, E., Tarnai, G. (eds.) Proceedings of the FORMS/FORMAT 2004 - Formal Methods for Automation and Safety in Railway and Automotive Systems. pp. 252–265. Technical University of Braunschweig (December 2004), iSBN 3-9803363-8-7

[2] Abdulla, P.A., Deneux, J., Stålmarck, G., Ågren, H., Åkerlund, O.: Designing Safe, Reliable Systems Using Scade. In: Margaria, T., Steffen, B. (eds.) ISoLA. Lecture Notes Comput. Sci., vol. 4313, pp. 115–129. Springer (2004), http://dx.doi.org/10.1007/11925040_8

[3] Airbus: A350 ata26 can application layer protocol – system interface document, reference V26D08019955, Issue 2, Date 12.10.2009

[4] Airbus: A350_cbc_srd (tbcke-4033/08) – application can bus control (cbc), baseline: 6.2 A350_DIR_CBC_SRD_CHK_1.0.0_05

[5] Airbus: Technical specification - chapter 2.1.5.1.26 smoke detection function (ata26) (Dec 2009), reference 4411 M1S 0031 01, Issue 3.0

[6] de Alfaro, L., Henzinger, T.A.: Interface automata. SIGSOFT Softw. Eng. Notes 26, 109–120 (September 2001), http://doi.acm.org/10.1145/503271.503226

[7] Alur, R., Dill, D.: A Theory of Timed Automata. Theoretical Computer Science (126), 183–235 (1994)

[8] Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking in dense real-time. Information and Computation 104(1), 2–34 (1993)

[9] André, C.: Semantics of S.S.M (safe state machine). Tech. Rep. UMR 6070, I3S Laboratory, University of Nice-Sophia Antipolis (2003)

[10] André, C.: Semantics of s.s.m. (safe state machine). Tech. rep., I3S Laboratory – UMR 6070, University of Nice-Sophia Antipolis / CNRS, BP 121, F 06903 Sophia Antipolis cédex (April 2003)

[11] Badban, B., Fränzle, M., Peleska, J., Teige, T.: Test automation for hybrid systems. In: Proceedings of the Third International Workshop on SOFTWARE QUALITY ASSURANCE (SOQUA 2006). Portland Oregon, USA (November 2006)

[12] Baker, P., Dai, Z.R., Grabowski, J., Haugen, O., Schieferdecker, I., Williams, C.: Model Driven Testing – Using the UML Testing Profile. Springer, Berlin, Heidelberg, New York (2008)

[13] Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.L.: Metropolis: An integrated electronic system design environment. In: Proceedings of the IEEE Computer Society. vol. 36.4, pp. 45–52 (April 2003)

[14] Barrett, C., Stump, A., Tinelli, C., Boehme, S., Cok, D., Deharbe, D., Dutertre, B., Fontaine, P., Ganesh, V., Griggio, A., Grundy, J., Jackson, P., Oliveras, A., Krstić, S., Moskal, M., Moura, L.D., Sebastiani, R., Cok, T.D., Hoenicke, J.: The SMT-LIB standard: Version 2.0. Tech. rep. (2010)

[15] Baufreton, P.: Sacres: A step ahead in the development of critical avoinics applications (abstract). In: Proceedings of the Second International Workshop on Hybrid Systems: Computation and Control. Lecture Notes Comput. Sci., vol. 1569. Springer-Verlag, London, UK (1999), http://portal.acm.org/citation.cfm?id=646879.710316

[16] Baufreton, P.: Visual notations based on synchronous languages for dynamic validation of gals systems. In: CCCT'04 Computing, Communications and Control Technologies. Austin (Texas) (August 2004)

[17] Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. pp. 200–236. No. 3185 in LNCS, Springer–Verlag (September 2004)

[18] Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Guernic, P.L., de Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE 91(1), 64–83 (jan 2003)

[19] Berry, G., Sentovich, E.: Embedding synchronous circuits in gals-based systems. In: Sophia-Antipolis conference on Micro-Electronics (SAME 98) (October 1998)

[20] Biallas, S., Brauer, J., Kowalewski, S.: Counterexample-guided abstraction refinement for plcs. In: Proceedings of the 5th International Workshop on Systems Software Verification (SSV 2010),Vancouver, Canada. pp. 2–9. USENIX Association, Berkeley, CA, USA (2010)

[21] Biere, A., Heljanko, K., Junttila, T.A., Latvala, T., Schuppan, V.: Linear encodings of bounded ltl model checking. Logical Methods in Computer Science 2(5) (2006)

[22] Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. Electronic Notes in Theoretical Computer Science 66(2), 160 – 177 (2002), http://www.sciencedirect.com/science/article/pii/S1571066104804109, FMICS'02, 7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems (ICALP 2002 Satellite Workshop)

[23] Biere, A., Heljanko, K., Junttila, T.A., Latvala, T., Schuppan, V.: Linear encodings of bounded ltl model checking. Logical Methods in Computer Science 2(5) (2006)

[24] Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley (2000)

[25] Bozga, M., Graf, S., Ober, I., Ober, I., Sifakis, J.: The if toolset. In: Bernardo, M., Corradini, F. (eds.) Formal Methods for the Design of Real-Time Systems, Lecture Notes in Computer Science, vol. 3185, pp. 131–132. Springer Berlin / Heidelberg (2004)

[26] Büchi, J.R.: Symposium on decision problems: On a decision method in restricted second order arithmetic. In: Ernest Nagel, P.S., Tarski, A. (eds.) Logic, Methodology and Philosophy of Science, Proceeding of the 1960 International Congress, Studies in Logic and the Foundations of Mathematics, vol. 44, pp. 1 – 11. Elsevier (1966), http://www.sciencedirect.com/science/article/pii/S0049237X09705646

[27] CENELEC: EN 50128 – Railway Applications – Software for Railway Control and Protection Systems. European Standard. (2001)

[28] CENELEC: EN 50159-1. Railway applications -Communication, signalling and processing systems Part 1: Safety-related communication in closed transmission systems (2001)

[29] CENELEC: EN 50159-2. Railway applications -Communication, signalling and processing systems Part 2: Safety related communication in open transmission systems (2001)

[30] Chapiro, D.M.: Globally-asynchronous locally-synchronous systems. Ph.D. thesis, Stanford University (1984)

[31] Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering SE-4(3), 178–186 (Mar 1978)

[32] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Berlin Heidelberg New York (2000)

[33] Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM 50(5), 752–794 (September 2003)

[34] Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 5, 752–794 (September 2003)

[35] Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge, Massachusetts (1999)

[36] Dajani-Brown, S., Cofer, D., Bouali, A.: Formal verification of an avionics sensor using SCADE. In: Lakhnech, Y., Yovine, S. (eds.) Proc. FORMATS/FTRTFT 2004. vol. 3253, pp. 5–20. Springer-Verlag (2004)

[37] Software considerations in airborne systems and equipment certifica- tion (December 1992)

[38] Doucet, F., Menarini, M., Krüger, I.H., Gupta, R., Talpin, J.P.: A verification approach for gals integration of synchronous components. Electron. Notes Theor. Comput. Sci. 146, 105–131 (January 2006)

[39] Duflot, M., Fribourg, L., Hérault, T., Lassaigne, R., Magniette, F., Messika, S., Peyronnet, S., Picaronny, C.: Probabilistic model checking of the CSMA/CD protocol using PRISM and APMC. In: Proc. 4th Workshop on Automated Verification of Critical Systems (AVoCS'04). Electronic Notes in Theoretical Computer Science, vol. 128(6), pp. 195–214. Elsevier Science (2004)

[40] Esposito, R., Sanseviero, A., Lazzaro, A., Marmo, P.: Formal verification of ertms euroradio safety critical protocol. In: Proceedings of FORMS 2003, May 15-16, 2003, Budapest, Hungary (2003)

[41] Garavel, H., Thivolle, D.: Verification of gals systems by combining synchronous languages and process calculi. In: Proceedings of the 16th International SPIN Workshop on Model Checking Software. pp. 241–260. Springer-Verlag, Berlin, Heidelberg (2009)

[42] Gastin, P., Oddoux, D.: Fast ltl to büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) Computer Aided Verification. Lecture Notes in Computer Science, vol. 2102, pp. 53–65. Springer-Verlag, London, UK (2001)

[43] Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Dembinski, P., Sredniawa, M. (eds.) Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification. IFIP Conference Proceedings, vol. 38, pp. 3–18. Chapman & Hall, Ltd. (1996)

[44] Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the compositional verification of real-time uml designs. SIGSOFT Softw. Eng. Notes 28, 38–47 (September 2003), http://doi.acm.org/10.1145/949952.940078

[45] Giese, H., Vilbig, A.: Separation of non-orthogonal concerns in software architecture and design. Software and System Modeling 5(2), 136–169 (2006)

[46] Gnesi, S., Latella, D., Massink, M.: Formal test-case generation for uml statecharts. In: Ninth IEEE International Conference on Engineering Complex Computer Systems (ICECCS'04). pp. 75–84. iceccs (2004)

[47] Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. Ph.D. thesis, Universite de Liege (1995)

[48] Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating Finite State Machines from Abstract State Machines. ACM SIGSOFT Software Engineering Notes 27(4), 112–122 (July 2002)

[49] Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating finite state machines from abstract state machines. ACM SIGSOFT Software Engineering Notes 27(4), 112–122 (July 2002)

[50] Güdemann, M., Ortmeier, F., Reif, W.: Using deductive cause-consequence analysis (DCCA) with SCADE. In: Computer Safety, Reliability, and Security, 26th International Conference, SAFECOMP 2007, Nuremberg, Germany, September 18-21, 2007. Lecture Notes in Computer Science, vol. 4680, pp. 465–478. Springer (2007), http://dx.doi.org/10.1007/978-3-540-75101-4_44

[51] Günther, H.: Verifikation von GALS Systemen. Master's thesis, TU Braunschweig (Jun 2011)

[52] Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language LUSTRE. In: Proceedings of the IEEE. vol. 79:9, pp. 1305–1320 (1991)

[53] Halbwachs, N., Lagnier, F., Raymond, P.: Synchronous observers and the verification of reactive systems. In: Proceedings of the Third International Conference on Methodology and Software Technology: Algebraic Methodology and Software Technology. pp. 83–96. Springer-Verlag, London, UK (1994), http://portal.acm.org/citation.cfm?id=646055.677894

[54] Halbwachs, N., Mandel, L.: Simulation and verification of asynchronous systems by means of a synchronous model. In: Proceedings of the Sixth International Conference on Application of Concurrency to System Design. pp. 3–14. IEEE Computer Society, Washington, DC, USA (2006), http://portal.acm.org/citation.cfm?id=1152965.1152975

[55] Halbwachs, N., Raymond, P.: Validation of synchronous reactive systems: from formal verification to automatic testing. In: Asian Computer Science Conference (ASIAN'99) (December 1999)

[56] Harel, D.: Statecharts: A visual formalism for complex systems (1987)

[57] Henry, J.: Integration of SCADE models generated code. Engineering note ESEG-EN-003, Esterel Technologies (5 2009)

[58] Holzmann, G.J.: State compression in spin: Recursive indexing and compression training runs. In: Proceedings of the third international SPIN workshop (1997)

[59] Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (Sep 2003)

[60] Holzmann, G.J.: A stack-slicing algorithm for multi-core model checking. Electron. Notes Theor. Comput. Sci. 198, 3–16 (February 2008)

[61] IEC: Iec 60050-191-am1 ed1.0 amendment 1 - international electrotechnical vocabulary. chapter 191: Dependability and quality of service. (1999)

[62] ISO/DIS 26262-4: Road vehicles – functional safety – part 4: Product development: system level. Tech. rep., International Organization for Standardization (2009)

[63] Jahier, E., Halbwachs, N., Raymond, P., Nicollin, X., Lesens, D.: Virtual execution of aadl models via a translation into synchronous programs. In: Proceedings of the 7th ACM & IEEE international conference on Embedded software. pp. 134–143. EMSOFT '07, ACM, New York, NY, USA (2007)

[64] Jones, C.B.: Specification and design of (parallel) programs. In: Proc. IFIP Congress. pp. 321–332 (1983)

[65] Kähloer, M.: The european train control system in thales signalling solutions. Mechanics Transport Communications 3, VIII–8–VIII–12 (2008)

[66] Kelly, S., Lyytinen, K., Rossi, M.: Metaedit+ a fully configurable multi-user and multi-tool case and came environment. Advanced Information Systems Engineering 1080, 1–21 (1996)

[67] Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell (1997), http://www.it.uu.se/research/group/darts/papers/texts/lpw-sttt97.pdf

[68] Le Guernic, P., J.-P., T., J.-L., L.L.: Polychrony for system design. Journal of Circuits, Systems and Computers (2002), special Issue on Application-Specific Hardware Design. World Scientific

[69] Ltd, S.S.P.: Enterprise architect professional 9.0 (2011), http://www.sparxsystems.com/products/ea/index.html

[70] Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)

[71] Maxemchuk, N.F., Sabnani, K.K.: Probabilistic verification of communication protocols. In: PSTV. pp. 307–320 (1987)

[72] metacase.com: Metaedit+ workbench (2009), http://www.metacase.com/mwb/

[73] Milner, R.: On relating synchrony and asynchrony. Tech. Rep. CSR-75-80, Computer Science Dept., Edinburgh Univ. (1981)

[74] Milner, R.: Calculi for synchrony and asynchrony. Theoret. Comput. Sci. 25(3) (July 1983)

[75] Möller, M.O.: Benchmark Analysis of GTL-Backends using Client-Server Mutex (2012), http://www.verified.de/en/publications/, Verified Systems International GmbH, Doc.Id.: Verified-WHITEPAPER-001-2012, Issue 1.2

[76] Mousavi, M.R., Le Guernic, P., Talpin, J.P., Shukla, S.K., Basten, T.: Modeling and validating globally asynchronous design in synchronous frameworks. In: Proceedings of the conference on Design, automation and test in Europe - Volume 1. pp. 10384–. DATE '04, IEEE Computer Society, Washington, DC, USA (2004), http://portal.acm.org/citation.cfm?id=968878.969070

[77] Object Management Group: OMG Systems Modeling Language (OMG SysML^{TM}). Tech. rep., Object Management Group (2010), OMG Document Number: formal/2010-06-02

[78] Peleska, J., Siegel, M.: Test automation of safety-critical reactive systems. South African Computer Jounal 19, 53–77 (1997)

[79] Peleska, J., Vorobev, E., Lapschies, F.: Automated Test Case Generation with SMT-Solving and Abstract Interpretation. In: NFM. vol. 6617, pp. 298–312 (2011)

[80] Peleska, J., Honisch, A., Lapschies, F., Löding, H., Schmid, H., Smuda, P., Vorobev, E., Zahlten, C.: Embedded systems testing benchmark (2011), http://www.mbt-benchmarks.org

[81] Peleska, J., Honisch, A., Lapschies, F., Löding, H., Schmid, H., Smuda, P., Vorobev, E., Zahlten, C.: A real-world benchmark model for testing concurrent real-time systems in the automotive domain. In: Proceedings of the ICTSS11, Paris, November 2011. Lecture Notes in Computer Science, Springer-Verlag (2011), to appear

[82] Peleska, J., Vorobev, E., Lapschies, F., Zahlten, C.: Automated model-based testing with RT-Tester. Tech. rep. (2011), http://www.informatik.uni-bremen.de/agbs/testingbenchmarks/turn_indicator/tool/rtt-mbt.pdf

[83] Potop-Butucaru, D., Caillaud, B.: Correct-by-construction asynchronous implementation of modular synchronous specifications. Fundam. Inf. 78, 131–159 (January 2007), http://portal.acm.org/citation.cfm?id=1366007.1366013

[84] Press, O.U.: Oxford dictionary of computing. Oxford paperback reference, Oxford University Press, 4 edn. (1997), http://books.google.co.uk/books?id=Hay6vTsGFAsC

[85] Rajan, B., Shyamasundar, R.: Multiclock esterel: a reactive framework for asynchronous design. In: Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International. pp. 201 –209 (2000)

[86] Ramesh, S., Sonalkar, S., D'silva, V., Chandra R., N., Vijayalakshmi, B.: A toolset for modelling and verification of gals systems. In: Alur, R., Peled, D. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 3114, pp. 385–387. Springer Berlin / Heidelberg (2004)

[87] Roggenbach, M.: Determinization of büchi-automata. In: Grädel, E., Thomas, W., Wilke, T. (eds.) Automata Logics, and Infinite Games, Lecture Notes in Computer Science, vol. 2500, pp. 43–60. Springer Berlin Heidelberg (2002), http://dx.doi.org/10.1007/3-540-36387-4_3

[88] RTCA,SC-167: Software Considerations in Airborne Systems and Equipment Certification, RTCA/DO-178B. RTCA (1992)

[89] Schlingloff, F., Barthel: Verifikation und test des profisafe-sicherheitsprofils (2007)

[90] Sistla, A.P.: Liveness and fairness in temporal logic. Formal Aspects of Computing 6(5), 495–512 (1994)

[91] Spillner, A., Linz, T., Schaefer, H.: Software Testing Foundations. dpunkt.verlag, Heidelberg (2006)

[92] Springintveld, J., Vaandrager, F., D'Argenio, P.: Testing timed automata. Theoretical Computer Science 254(1-2), 225–257 (March 2001)

[93] Sulzmann, D.M., Zechner, A., Hedayati, R.: Anforderungsdokument für die Fallstudie Bahnübergangssicherungsanlage. Tech. rep., ICS AG (2011)

[94] Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software-Concepts and Tools 17, 103–120 (1996)

[95] UNISIG: ERTMS/ETCS SystemRequirements Specification, Chapter 3, Principles, vol. Subset-026-3 (February 2012), issue 3.3.0

[96] Vasilevskii, M.P.: Failure diagnosis of automata. Kibernetika (Transl.) 4, 98–108 (July-August 1973)

[97] Verified Systems International GmbH, Bremen: RT-Tester 6.2 – User Manual (2007)

[98] Weißleder, S.: Test Models and Coverage Criteria for Automatic Model-Based Test Generation with UML State Machines. Doctoral thesis, Humboldt-University Berlin, Germany (2010)

Appendix A

Manual Smoke Detection Model in GTL (2 Smoke Sensors) - Full Listing

```
1  // ------------------------------------------------------------------------
2  // Manual definition of the verification constracts
3  // ------------------------------------------------------------------------
4  // @TABLE OF CONTENTS:                   [TOCD: 19:44 09 Aug 2011]
5  //
6  //  [1] STATE MACHINE MODELS
7  //      [1.1] STATE MACHINE CORRESPONDING TO CIDS()
8  //      [1.2] STATE MACHINE CORRESPONDING TO CAN BUS()
9  //      [1.3] STATE MACHINE CORRESPONDING TO SMOKESENSOR()
10 //      [1.4] STATE MACHINES FOR SMOKE DETECTOR (split-up)
11 //          [1.4.1] Statemachine SDIdentRequestWatchdog
12 //          [1.4.2] Statemachine SDLOGIC
13 //          [1.4.3] Statemachine SDSEND (see EA: UPSTREAMSD2)
14 //      [1.5] AUXILIAR: GLOBAL CLOCK and dependent limit-range int (WRAPCLOCK)
15 //      [1.6] AUXILIAR: generating CONSTANTS (as parameters for instances)
16 //  [2] INSTANCES AND CONNECTIONS
17 //  [3] VERIFICATION GOALS
18 // ------------------------------------------------------------------------
19
20 // ////////////////////////////////////////////////////////////////////
21 // [1] STATE MACHINE MODELS
22 // ////////////////////////////////////////////////////////////////////
23
24
25 // ////////////////////////////////////////////
26 // [1.1] STATE MACHINE CORRESPONDING TO CIDS()
27 // ////////////////////////////////////////////
28
29 model[none] CIDS("") {
30
31   input int t;
32   input int IdentRequestRepeater;
33   output bool IdentRequestDone;
34   output int inferTopologyCanFail;
35   output int smokeAlarm; // number of smoke detector reporting last alarm
36                          // 0 for none
37   input int OUTCAN0Identifier;
38   input int OUTCAN2Identifier;
```

```
39  output int TOSENDUPSTREAM;
40  automaton {
41    // -- CIDSLOGIC STATE: IDENTREQUESTMODE ----------------------------------
42    // -- IDENTREQUESTMODE STATES --------------------------------
43    init state IDENTREQUESTMODEInitial {
44
45      transition[(IdentRequestDone = false) and
46                 (0 = t) and
47                 (TOSENDUPSTREAM = 10)]
48        IDENTREQUESTMODETRANSMITIDENTREQUEST;
49    }
50    state IDENTREQUESTMODETRANSMITIDENTREQUEST {
51      (IdentRequestDone = false) and
52        (TOSENDUPSTREAM = 10);
53
54      transition[(t >= 50) and
55                 (IdentRequestDone = true) and
56                 (inferTopologyCanFail = 7)]
57        IDENTREQUESTMODECANFAILNOACK;
58
59      transition[(OUTCAN0Identifier + 1) = 12]
60        IDENTREQUESTMODEACKRECEIVED;
61
62      transition[(IdentRequestDone = true) and
63                 (0 = IdentRequestRepeater)]
64        SDPOLLINGMODE;
65    }
66    state IDENTREQUESTMODECANFAILNOACK {
67      (IdentRequestDone = true) and
68        (TOSENDUPSTREAM = 10);
69
70      transition[(IdentRequestDone = true) and
71                 (0 = IdentRequestRepeater)]
72        SDPOLLINGMODE;
73    }
74    state IDENTREQUESTMODEACKRECEIVED {
75      (IdentRequestDone = false) and
76        (TOSENDUPSTREAM = 10);
77
78      transition[((OUTCAN0Identifier + 1) = 13) and
79                 (IdentRequestDone = true) and
80                 (inferTopologyCanFail = 7)]
81        IDENTREQUESTMODECANFAILREPLYRECEIVED;
82
83      transition[t >= 500]
84        IDENTREQUESTMODEPROTOCOLFAIL;
85
86      transition[OUTCAN2Identifier = 10]
87        IDENTREQUESTMODEREQUESTROUNDTRIPOK;
88
89      transition[(IdentRequestDone = true) and
90                 (0 = IdentRequestRepeater)]
91        SDPOLLINGMODE;
92    }
93    state IDENTREQUESTMODECANFAILREPLYRECEIVED {
94      (IdentRequestDone = true) and
95        (TOSENDUPSTREAM = 10);
```

```
96
97          transition[(IdentRequestDone = true) and
98                     (0 = IdentRequestRepeater)]
99            SDPOLLINGMODE;
100       }
101       state IDENTREQUESTMODEREQUESTROUNDTRIPOK {
102         (IdentRequestDone = true) and
103           (TOSENDUPSTREAM = 10);
104
105         transition[(IdentRequestDone = true) and
106                    (0 = IdentRequestRepeater)]
107           SDPOLLINGMODE;
108       }
109       state IDENTREQUESTMODEPROTOCOLFAIL {
110         TOSENDUPSTREAM = 10;
111         transition
112           IDENTREQUESTMODEPROTOCOLFAIL;
113       }
114       // -- CIDSLOGIC STATE: SDPOLLINGMODE ------------------------------------
115       state SDPOLLINGMODE {
116         IdentRequestRepeater <= 3600;
117         transition[(IdentRequestRepeater >= 3600) and
118                    (IdentRequestDone = false)]
119           IDENTREQUESTMODEInitial;
120       }
121     };
122 }
123
124
125 // ////////////////////////////////////////////
126 // [1.2] STATE MACHINE CORRESPONDING TO CAN BUS()
127 // ////////////////////////////////////////////
128
129 model[none] CAN("") {
130   input int UPINCANIdentifier;
131   input int DOWNINCANIdentifier;
132   output int OUTCANIdentifier;
133   output bool BusFault;
134   automaton {
135     init state CANNormalOperation0 {
136       BusFault = false;
137       transition[UPINCANIdentifier <= DOWNINCANIdentifier]
138         CANNormalOperationA;
139       transition[UPINCANIdentifier > DOWNINCANIdentifier]
140         CANNormalOperationB;
141       transition
142         CANBusFailure;
143     }
144     state CANNormalOperationA {
145       BusFault = false;
146       OUTCANIdentifier = UPINCANIdentifier;
147       transition
148         CANNormalOperation0;
149     }
150     state CANNormalOperationB {
151       BusFault = false;
152       OUTCANIdentifier = DOWNINCANIdentifier;
```

```
153         transition
154           CANNormalOperation0;
155       }
156       state CANBusFailure {
157         BusFault = true;
158         OUTCANIdentifier = 0;
159         transition
160           CANNormalOperation0;
161       }
162     };
163   }
164
165   // ////////////////////////////////////////////
166   // [1.3] STATE MACHINE CORRESPONDING TO SMOKESENSOR()
167   // ////////////////////////////////////////////
168
169   model[none] SMOKESENSOR("") {
170     output bool smoke;
171     automaton {
172       init state IDLE {
173         smoke = false;
174         transition
175           DETECTINGSMOKE;
176       }
177       state DETECTINGSMOKE {
178         smoke = true;
179         transition
180           IDLE;
181       }
182     };
183   }
184
185
186   // ////////////////////////////////////////////
187   // [1.4] STATE MACHINES FOR SMOKE DETECTOR (split-up)
188   // ////////////////////////////////////////////
189
190
191   // ////////////////////////////////////
192   // [1.4.1] Statemachine SDIdentRequestWatchdog
193   // ////////////////////////////////////
194
195   model[none] SDIdentRequestWatchdog("") {
196     input int OUTCANIdentifier;
197     input bool elapsed;
198     output int TOSENDDOWNSTREAM;
199     output bool reset;
200     automaton {
201       init state WaitforIdentRequest {
202         ((not (OUTCANIdentifier = 10)) and
203          (reset = false));
204         transition[OUTCANIdentifier = 10]
205           WaitforIdentAckENTRY;
206       }
207       state WaitforIdentAckENTRY {
208         reset = true;
209         transition[true]
```

```
210           WaitforIdentAck;
211       }
212       state WaitforIdentAck {
213         reset = false;
214         transition[OUTCANIdentifier = 12]
215           WaitforIdentRequest;
216
217         transition[(not (OUTCANIdentifier = 12)) and
218                    (elapsed = true) and
219                    (TOSENDDOWNSTREAM = 13)]
220           WaitforIdentRequest;
221       }
222     };
223 }
224
225 // ////////////////////////////////////
226 // [1.4.2] Statemachine SDLOGIC
227 // ////////////////////////////////////
228
229
230 model[none] SDLOGIC("") {
231   input bool smoke;
232   input int myAddress;
233   input int OUTCANIdentifierDOWNSTREAM; // OUTCAN<n>Identifier
234   input int OUTCANIdentifierUPSTREAM; // OUTCAN<n+1>Identifier
235   output int TOSENDUPSTREAM;
236   output int TOSENDDOWNSTREAM;
237   automaton {
238     init state NOINPUT {
239       transition[not (OUTCANIdentifierDOWNSTREAM = 0)]
240         INPUTUPSTREAM;
241       transition[not (OUTCANIdentifierUPSTREAM = 0)]
242         INPUTDOWNSTREAM;
243     }
244     state INPUTUPSTREAM {
245       (((OUTCANIdentifierDOWNSTREAM = 10)
246         => ((TOSENDUPSTREAM = OUTCANIdentifierDOWNSTREAM) and (TOSENDDOWNSTREAM =
                12 + myAddress)))
247        and
248        (((OUTCANIdentifierDOWNSTREAM = 11 + myAddress))
249         => (TOSENDUPSTREAM = 0))
250        and
251        (((OUTCANIdentifierDOWNSTREAM = 11 + myAddress) and (smoke = true))
252         => (TOSENDDOWNSTREAM = 14))
253        and
254        (((OUTCANIdentifierDOWNSTREAM = 11 + myAddress) and (smoke = false))
255         => (TOSENDDOWNSTREAM = 15))
256        and
257        (((OUTCANIdentifierDOWNSTREAM = 11) and (false))
258         => ((TOSENDUPSTREAM = OUTCANIdentifierDOWNSTREAM) and
259              (TOSENDDOWNSTREAM = 0)))
260        and
261        ((not ((OUTCANIdentifierDOWNSTREAM = 10) or
262                (OUTCANIdentifierDOWNSTREAM = 11)))
263         => ((TOSENDUPSTREAM = 0) and
264              (TOSENDDOWNSTREAM = 0)))
265        );
```

```
266
267         transition[OUTCANIdentifierDOWNSTREAM = 0]
268           NOINPUT;
269       }
270       state INPUTDOWNSTREAM {
271         ((((OUTCANIdentifierUPSTREAM = 13) or
272            (OUTCANIdentifierUPSTREAM = 15) or
273            (OUTCANIdentifierUPSTREAM = 14))
274           => ((TOSENDUPSTREAM = 0) and
275                (TOSENDDOWNSTREAM = OUTCANIdentifierUPSTREAM)))
276          and
277          ((not ((OUTCANIdentifierUPSTREAM = 13) or
278                  (OUTCANIdentifierUPSTREAM = 15) or
279                  (OUTCANIdentifierUPSTREAM = 14)))
280           => ((TOSENDUPSTREAM = 0) and
281                (TOSENDDOWNSTREAM = 0)))
282          );
283
284         transition[OUTCANIdentifierUPSTREAM = 0]
285           NOINPUT;
286       }
287     };
288 }
289
290 // ////////////////////////////////////
291 // [1.4.3] Statemachine SDSEND (see EA: UPSTREAMSD2)
292 // ////////////////////////////////////
293
294
295 model[none] SDSEND(""){
296   input int DATATOSEND;
297   input bool elapsed;
298   input int OUTCANIdentifier;
299   output int OUTPUT;
300   output int DATATOSENDold; // should be input
301   output bool reset;
302   automaton {
303     init state IDLE {
304       (reset = false);
305       transition[DATATOSEND != DATATOSENDold]
306         ATTEMPTTOTRANSMIT;
307     }
308     state ATTEMPTTOTRANSMIT {
309       ((DATATOSENDold = DATATOSEND) and
310        (reset = false) and
311        (OUTPUT = DATATOSEND));
312       transition[not(DATATOSENDold = DATATOSEND)]
313         ATTEMPTTOTRANSMIT;
314       transition[DATATOSENDold = OUTCANIdentifier]
315         TRANSMITEntry;
316     }
317     state TRANSMITEntry {
318       (reset = true);
319       transition
320         TRANSMIT;
321     }
322     state TRANSMIT {
```

```
323         (reset = false);
324         transition[not(DATATOSENDold = OUTCANIdentifier)]
325           ATTEMPTTOTRANSMIT;
326         transition[not(DATATOSENDold = DATATOSEND)]
327           ATTEMPTTOTRANSMIT;
328         transition[elapsed = true]
329           IDLEEntry;
330       }
331       state IDLEEntry {
332         (OUTPUT = 0);
333         transition
334           IDLE;
335       }
336     };
337 }
338
339 // ////////////////////////////////////////////
340 // [1.5] AUXILIAR: GLOBAL CLOCK and dependent limit-range int (WRAPCLOCK)
341 // ////////////////////////////////////////////
342
343 model[none] CLOCK("") {
344   output int time;
345   input int timeold; // faulty: mapped to 'input', but should be 'internal'
346   automaton {
347     init state INITIAL {
348       transition[time = 0]
349         TICK;
350     }
351     state TICK {
352       timeold = time;
353       transition[ time = timeold + 1]
354         TICK;
355     }
356   };
357 }
358
359 // wraparound at TIMEOUTIDENTREQUESTMODE
360 model[none] WRAPCLOCK("") {
361   input int tick;
362   output int time;
363   // below: modulo-computation
364   // x mod M = x - (x/M)*M
365   always time = tick - ((tick / (3600 + 1)) * (3600 + 1));
366 }
367
368 // can be resetted; timeout at TIMEOUTIDENTACK
369 model[none] STOPWATCHIDENTACK("") {
370   input int tick;
371   input bool reset;
372   output bool elapsed;
373   input int memorisedtime; // faulty: mapped to 'input', but should be 'internal'
        or 'output'
374   automaton {
375     init state SWIDLE {
376       ((reset = false) and
377        (elapsed = false));
378       transition[(reset = true) and (memorisedtime = tick)]
```

```
379           SWRUNNING;
380       }
381       state SWRUNNING {
382         ((memorisedtime + 50 < tick) and
383          (elapsed = false));
384         transition[memorisedtime + 50 = tick]
385           SWTRIGGER;
386       }
387       state SWTRIGGER {
388         (elapsed = true);
389         transition[(reset = true) and (memorisedtime = tick)]
390           SWRUNNING;
391       }
392     };
393   }
394
395   // can be resetted; timeout at TIMEOUTSEND
396   model[none] STOPWATCHSEND("") {
397     input int tick;
398     input bool reset;
399     output bool elapsed;
400     input int memorisedtime; // faulty: mapped to 'input', but should be 'internal'
          or 'output'
401     automaton {
402       init state SWIDLE {
403         ((reset = false) and
404          (elapsed = false));
405         transition[(reset = true) and (memorisedtime = tick)]
406           SWRUNNING;
407       }
408       state SWRUNNING {
409         ((memorisedtime + 5 < tick) and
410          (elapsed = false));
411         transition[memorisedtime + 5 = tick]
412           SWTRIGGER;
413       }
414       state SWTRIGGER {
415         (elapsed = true);
416         transition[(reset = true) and (memorisedtime = tick)]
417           SWRUNNING;
418       }
419     };
420   }
421
422   // ////////////////////////////////////////////
423   // [1.6] AUXILIAR: generating CONSTANTS (as parameters for instances)
424   // ////////////////////////////////////////////
425
426   model[none] CONST1("") {
427     output int constant;
428     always (constant = 1);
429   }
430
431   model[none] CONST2("") {
432     output int constant;
433     always (constant = 2);
434   }
```

```
435
436
437 // ////////////////////////////////////////////////////////////////
438 // [2] INSTANCES AND CONNECTIONS
439 // ////////////////////////////////////////////////////////////////
440
441 instance CLOCK clock0;
442 instance WRAPCLOCK triggerIRR;
443
444 instance CIDS cids0;
445
446 instance CAN can0;
447 instance CAN can1;
448 instance CAN can2;
449
450 instance SMOKESENSOR sensor1;
451 instance SMOKESENSOR sensor2;
452
453 instance STOPWATCHIDENTACK stopwatch1;
454 instance STOPWATCHIDENTACK stopwatch2;
455
456 instance STOPWATCHSEND stopwatchsend1DOWNSTREAM;
457 instance STOPWATCHSEND stopwatchsend1UPSTREAM;
458 instance STOPWATCHSEND stopwatchsend2DOWNSTREAM;
459 instance STOPWATCHSEND stopwatchsend2UPSTREAM;
460
461
462 instance SDIdentRequestWatchdog SDwatchdog1;
463 instance SDIdentRequestWatchdog SDwatchdog2;
464
465 instance SDLOGIC SDlogic1;
466 instance SDLOGIC SDlogic2;
467
468 instance SDSEND DOWNSTREAMSD1;
469 instance SDSEND UPSTREAMSD1;
470 instance SDSEND DOWNSTREAMSD2;
471 instance SDSEND UPSTREAMSD2;
472
473 instance CONST1 const1;
474 instance CONST2 const2;
475
476 connect clock0.time cids0.t;
477 connect clock0.time triggerIRR.tick;
478 connect clock0.time stopwatch1.tick;
479 connect clock0.time stopwatch2.tick;
480 connect clock0.time stopwatchsend1DOWNSTREAM.tick;
481 connect clock0.time stopwatchsend1UPSTREAM.tick;
482 connect clock0.time stopwatchsend2DOWNSTREAM.tick;
483 connect clock0.time stopwatchsend2UPSTREAM.tick;
484
485 connect sensor1.smoke SDlogic1.smoke;
486 connect sensor2.smoke SDlogic2.smoke;
487
488
489 // -- SD: watchdog
490 connect can1.OUTCANIdentifier SDwatchdog1.OUTCANIdentifier;
491 connect stopwatch1.elapsed SDwatchdog1.elapsed;
```

```
492 connect SDwatchdog1.reset stopwatch1.reset;
493 connect SDwatchdog1.TOSENDDOWNSTREAM DOWNSTREAMSD1.DATATOSEND;
494 connect can2.OUTCANIdentifier SDwatchdog2.OUTCANIdentifier;
495 connect stopwatch2.elapsed SDwatchdog2.elapsed;
496 connect SDwatchdog2.reset stopwatch2.reset;
497 connect SDwatchdog2.TOSENDDOWNSTREAM DOWNSTREAMSD2.DATATOSEND;
498
499
500 // -- SD: logic
501 connect can0.OUTCANIdentifier SDlogic1.OUTCANIdentifierDOWNSTREAM;
502 connect can1.OUTCANIdentifier SDlogic1.OUTCANIdentifierUPSTREAM;
503 connect SDlogic1.TOSENDUPSTREAM UPSTREAMSD1.DATATOSEND;
504 connect SDlogic1.TOSENDDOWNSTREAM DOWNSTREAMSD1.DATATOSEND;
505 connect can1.OUTCANIdentifier SDlogic2.OUTCANIdentifierDOWNSTREAM;
506 connect can2.OUTCANIdentifier SDlogic2.OUTCANIdentifierUPSTREAM;
507 connect SDlogic2.TOSENDUPSTREAM UPSTREAMSD2.DATATOSEND;
508 connect SDlogic2.TOSENDDOWNSTREAM DOWNSTREAMSD2.DATATOSEND;
509
510 connect const1.constant SDlogic1.myAddress;
511 connect const2.constant SDlogic2.myAddress;
512
513
514 // -- SD: data transmission
515 connect stopwatchsend1DOWNSTREAM.elapsed DOWNSTREAMSD1.elapsed;
516 connect DOWNSTREAMSD1.reset stopwatchsend1DOWNSTREAM.reset;
517 connect DOWNSTREAMSD1.OUTPUT can0.DOWNINCANIdentifier;
518
519 connect stopwatchsend2DOWNSTREAM.elapsed DOWNSTREAMSD2.elapsed;
520 connect DOWNSTREAMSD2.reset stopwatchsend2DOWNSTREAM.reset;
521 connect DOWNSTREAMSD2.OUTPUT can1.DOWNINCANIdentifier;
522
523 connect stopwatchsend1UPSTREAM.elapsed UPSTREAMSD1.elapsed;
524 connect UPSTREAMSD1.reset stopwatchsend1UPSTREAM.reset;
525 connect UPSTREAMSD1.OUTPUT can1.UPINCANIdentifier;
526
527 connect stopwatchsend2UPSTREAM.elapsed UPSTREAMSD2.elapsed;
528 connect UPSTREAMSD2.reset stopwatchsend2UPSTREAM.reset;
529 connect UPSTREAMSD2.OUTPUT can2.UPINCANIdentifier;
530
531 // -- CIDS
532 connect triggerIRR.time cids0.IdentRequestRepeater;
533 connect cids0.TOSENDUPSTREAM can0.UPINCANIdentifier;
534 connect can0.OUTCANIdentifier cids0.OUTCAN0Identifier;
535
536
537
538 // ////////////////////////////////////////////////////////////////
539 // [3] VERIFICATION GOALS
540 // ////////////////////////////////////////////////////////////////
541
542 // safety (simple)
543 verify {
544   always not (cids0.TOSENDUPSTREAM = 99);
545   not always (can1.BusFault = false);
546 }
```

Listing A.1: Full Smoke Model in GTL (2 Smoke Detectors)

A.1 Part that triggers large PROMELA output

```
1
2
3  // ////////////////////////////////////
4  // [2.4.2] Statemachine SD_LOGIC (NOT SIMPLIFIED)
5  // ////////////////////////////////////
6
7
8  model[none] SDLOGIC("") {
9    input bool smoke;
10   input int myAddress;
11   input int OUTCANIdentifierDOWNSTREAM; // OUTCAN<n>Identifier
12   input int OUTCANIdentifierUPSTREAM;    // OUTCAN<n+1>Identifier
13   output int TOSENDUPSTREAM;
14   output int TOSENDDOWNSTREAM;
15   automaton {
16     init state NOINPUT {
17       transition[not (OUTCANIdentifierDOWNSTREAM = 0)]
18         INPUTUPSTREAM;
19       transition[not (OUTCANIdentifierUPSTREAM = 0)]
20         INPUTDOWNSTREAM;
21     }
22     state INPUTUPSTREAM {
23       (((OUTCANIdentifierDOWNSTREAM = 10)
24         => ((TOSENDUPSTREAM = OUTCANIdentifierDOWNSTREAM) and (TOSENDDOWNSTREAM =
               12 + myAddress)))
25        and
26        (((OUTCANIdentifierDOWNSTREAM = 11 + myAddress))
27         => (TOSENDUPSTREAM = 0))
28        and
29        (((OUTCANIdentifierDOWNSTREAM = 11 + myAddress) and (smoke = true))
30         => (TOSENDDOWNSTREAM = 14))
31        and
32        (((OUTCANIdentifierDOWNSTREAM = 11 + myAddress) and (smoke = false))
33         => (TOSENDDOWNSTREAM = 15))
34        and
35        (((OUTCANIdentifierDOWNSTREAM = 11) and (false))
36         => ((TOSENDUPSTREAM = OUTCANIdentifierDOWNSTREAM) and
37              (TOSENDDOWNSTREAM = 0)))
38        and
39        ((not ((OUTCANIdentifierDOWNSTREAM = 10) or
40                (OUTCANIdentifierDOWNSTREAM = 11)))
41         => ((TOSENDUPSTREAM = 0) and
42              (TOSENDDOWNSTREAM = 0)))
43        );
44       transition[OUTCANIdentifierDOWNSTREAM = 0]
45         NOINPUT;
46     }
47     state INPUTDOWNSTREAM {
48       ((((OUTCANIdentifierUPSTREAM = 13) or
49          (OUTCANIdentifierUPSTREAM = 15) or
50          (OUTCANIdentifierUPSTREAM = 14))
51         => ((TOSENDUPSTREAM = 0) and
52              (TOSENDDOWNSTREAM = OUTCANIdentifierUPSTREAM)))
53        and
54        ((not ((OUTCANIdentifierUPSTREAM = 13) or
```

```
55                  (OUTCANIdentifierUPSTREAM = 15) or
56                  (OUTCANIdentifierUPSTREAM = 14)))
57          => ((TOSENDUPSTREAM    = 0) and
58              (TOSENDDOWNSTREAM = 0)))
59        );
60        transition[OUTCANIdentifierUPSTREAM = 0]
61          NOINPUT;
62      }
63    };
64  }
65
66  instance SDLOGIC SDlogic1;
67
68  verify {
69    always true;
70  }
```

Listing A.2: Original Part of the Smoke Detection Model in GTL

A.2 Modified part with small PROMELA output

```
1   // ////////////////////////////////////
2   // [2.4.2] Statemachine SD_LOGIC (SIMPLIFIED)
3   // ////////////////////////////////////
4
5   model[none] SDLOGICsmall("") {
6     input bool smoke;
7     input int myAddress;
8     input int OUTCANIdentifierDOWNSTREAM; // OUTCAN<n>Identifier
9     input int OUTCANIdentifierUPSTREAM;    // OUTCAN<n+1>Identifier
10    output int TOSENDUPSTREAM;
11    output int TOSENDDOWNSTREAM;
12    automaton {
13      init state NOINPUT {
14        transition[not (OUTCANIdentifierDOWNSTREAM = 0)]
15          INPUTUPSTREAM;
16        transition[not (OUTCANIdentifierUPSTREAM = 0)]
17          INPUTDOWNSTREAM;
18      }
19      state INPUTUPSTREAM {
20        // SIMPLIFIED:
21        TOSENDDOWNSTREAM = OUTCANIdentifierUPSTREAM;
22
23        transition[OUTCANIdentifierDOWNSTREAM = 0]
24          NOINPUT;
25      }
26      state INPUTDOWNSTREAM {
27        // SIMPLIFIED:
28        TOSENDUPSTREAM = OUTCANIdentifierDOWNSTREAM;
29        transition[OUTCANIdentifierUPSTREAM = 0]
30          NOINPUT;
31      }
32    };
33  }
34
```

```
35 instance SDLOGICsmall SDlogicsmall;
36
37 verify {
38   always true;
39 }
```

Listing A.3: Simplified Part of the Smoke Detection Model in GTL

Appendix B

Abbreviations used in the Smoke Detection Case Study

BITE Built-in Test Equipment

CAN Controller Area Network

CBC Can Bus Control

CIDS Cabin Intercommunication and Data System

DEU-B Decoder/Encoder Unit, variant-B (middle-line)

DLC Data Length Code

DSL Domain-Specific Language

FEDC Fire Extinguishing Data Converter

GAC Globally Asynchronous Component

GALS Globally Asynchronous, Locally Synchronous

HSI Hardware Integration

LSC Locally Synchronous Component

MSB Most Signification Bit

S/D Smoke Detector

SDF Smoke Detection Facility

SUT System Under Test

UML Unified Modeling Language

XMI XML Metadata Interchange

XML eXtended Markup Language

Appendix C

Appendix

Table C.1: Datatypedefinitions ”BahnUebergangsSteuerung”

Type name	Data type	Definition
TAZR_Zustand	Enumeration	enum {AZRFrei, AZRBelegt, AZRGestoert, AZRTestOB, AZRKeinSignal}
TAZRKommando	Enumeration	enum {AZRKeinKommando, AZRInit, AZRTest, AZRBelegtEmpfangen}
TSchrankeZustand	Enumeration	enum {SchrankeOffen, SchrankeGeschlossen, SchrankeUnbekannt, SchrankeGestoert}
TSchrankeKommando	Enumeration	enum {SchrankeKeinKommando, SchrankeOeffnen, SchrankeSchliessen}
TStrassenSignalZustand	Enumeration	enum {StrasseRotAusGelbAus, StrasseRotAusGelbAn, StrasseRotAnGelbAus, StrasseRotAnGelbAn, StrasseGestoert, StrasseTestOB, StrasseKeinSignal}
TStrassenSignalKommando	Enumeration	enum {StrasseInit, StrasseKeinKommando, StrasseTest, StrasseSichern, StrasseEntsichern, StrasseAus, StrasseStoerung}
TUeberwachungZustand	Enumeration	enum {UeberwachungAn, UeberwachungAus, UeberwachungGestoert,

Continued on next page ...

Table C.1 – ... continued from previous page

Type name	Data type	Definition
		UeberwachungTestOB, UeberwachungKeinSignal, UeberwachungAn, UeberwachungAus, UeberwachungGestoert, UeberwachungTestOB, UeberwachungKeinSignal}
TUeberwachungKommando	Enumeration	enum {UeberwachungKeinKommando, UeberwachungInit, UeberwachungTest, UeberwachungSichern, UeberwachungEntsichern}
TUmgebungZug	Enumeration	enum {ZugVorhanden, ZugAbwesend}
TUmgebungVerkehr	Enumeration	enum {VerkehrBlockiert, VerkehrFrei}
TUmgebungKfz	Enumeration	enum {KfzRotAusGelbAus, KfzRotAusGelbAn, KfzRotAnGelbAus, KfzRotAnGelbAn}
TUmgebungTfZug	Enumeration	enum {TfZugFahrtbegriff, TfZugHaltbegriff}

Table C.2: Interfacedescription ”BahnUebergangsSteuerung” (input)

Interface name	input	Data type	Value	Short description
ZustandAZR	”AchsZaehlRechner”	TAZR_Zustand	AZR_Frei	detection of leaving trains, release of protection
			AZR_Belegt	detection of incoming trains, signaling rail sections occupied
			AZR_Gestoert	malfunction message
			AZR_TestOB	feedback self test axle counter, error detection, functional check
			AZR_KeinSignal	placeholder/dummy
ZustandSchranke	”Schranke”	(Vector-) (TSchrankeZustand	SchrankeOffen	feedback/responds of barrier
			SchrankeGeschlossen	feedback/responds of barrier
			SchrankeUnbekannt	placeholder/dummy
			SchrankeGestoert	malfunction message
ZustandStrassenSignal	”StrassenSignal”	(Vector-) StrassenSignalZustand	StrasseRotAusGelbAus	state of the traffic light (none)
			StrasseRotAusGelbAn	state of the traffic light (yellow)
			StrasseRotAnGelbAus	state of the traffic light (red)
			StrasseRotAnGelbAn	state of the traffic light (yellow+red)
			StrasseGestoert	malfunction message
			StrassekeinSignal	placeholder/dummy
			StrasseTestOB	feedback self test traffic lights
ZustandUeberwachungs-Signal	”UeberwachungsSignal”	TUeberwachungZustand	UeberwachungAn	feedback that monitoring signal has been switched on (orange)
			UeberwachungAus	feedback that monitoring signal has been switched off (none)
			UeberwachungGestoert	malfunction message
			UeberwachungTestOB	feedback self test monitoring signal
			UeberwachungKeinSignal	placeholder/dummy

Continued on next page ...

Table C.3 – ... continued from previous page

Interface name	output	Data type	Value	Short description

Table C.3: Interfacedescription ”BahnUebergangsSteuerung” (output)

Interface name	output	Data type	Value	Short description
KommandoAZR	”AchsZaehlRechner”	TAZR_Kommando	AZR_KeinKommando	placeholder/dummy
			AZR_Init	initialization axle counter
			AZR_Test	envoking built in self-test
			AZR_BelegtEmpfangen	detection of incoming trains, release of protection, fault analysis
KommandoSchranke	”Schranke”	(Vector-) TSchrankeKommando	SchrankeKeinKommando	placeholder/dummy
			SchrankeOeffnen	initiate opening of barrier
			SchrankeSchliessen	initiate closing of barrier
KommandoStrassenSignal	”StrassenSignal”	(Vector-) StrassenSignalZustand	StrasseStoerung	malfunction message
			StrasseKeinKommando	placeholder/dummy
			StrasseInit	initialization of traffic light
			StrasseTest	envoking built in self-test
			StrasseSichern	enable traffic light
			StrasseEntsichern	disable traffic lights
			StrasseAus	<unused signal >
KommandoUeberwachungs-Signal	”UeberwachungsSignal”	TUeberwachung-Kommando	UeberwachungKeinKommando	malfunction message
			UeberwachungInit	initialization of monitoring signal
			UeberwachungTest	envoking built in self-test
			UeberwachungSichern	enable monitoring signal
			UeberwachungEntsichern	disable monitoring signal

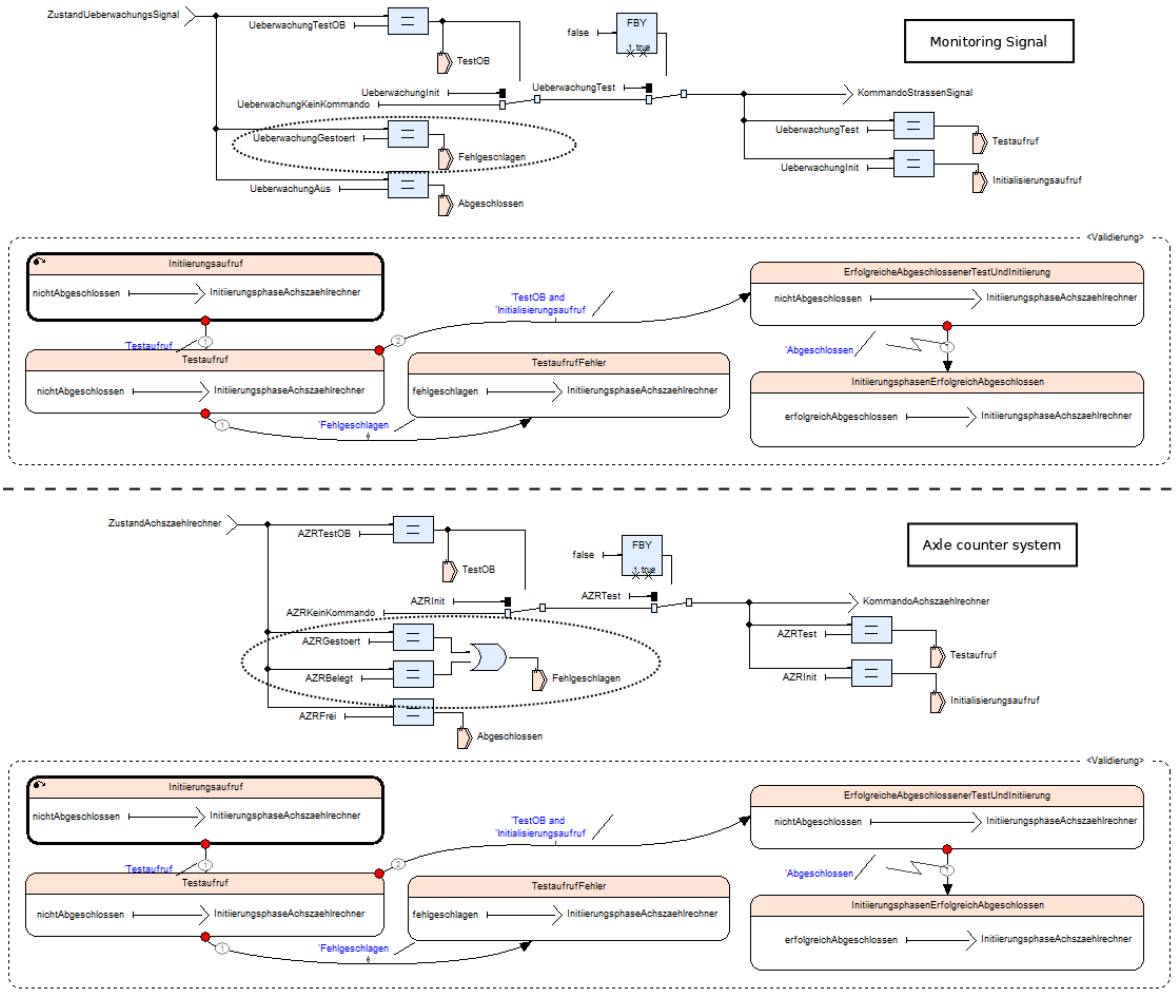

Figure C.1: Example of similar implementation for the initialization of subsystems.

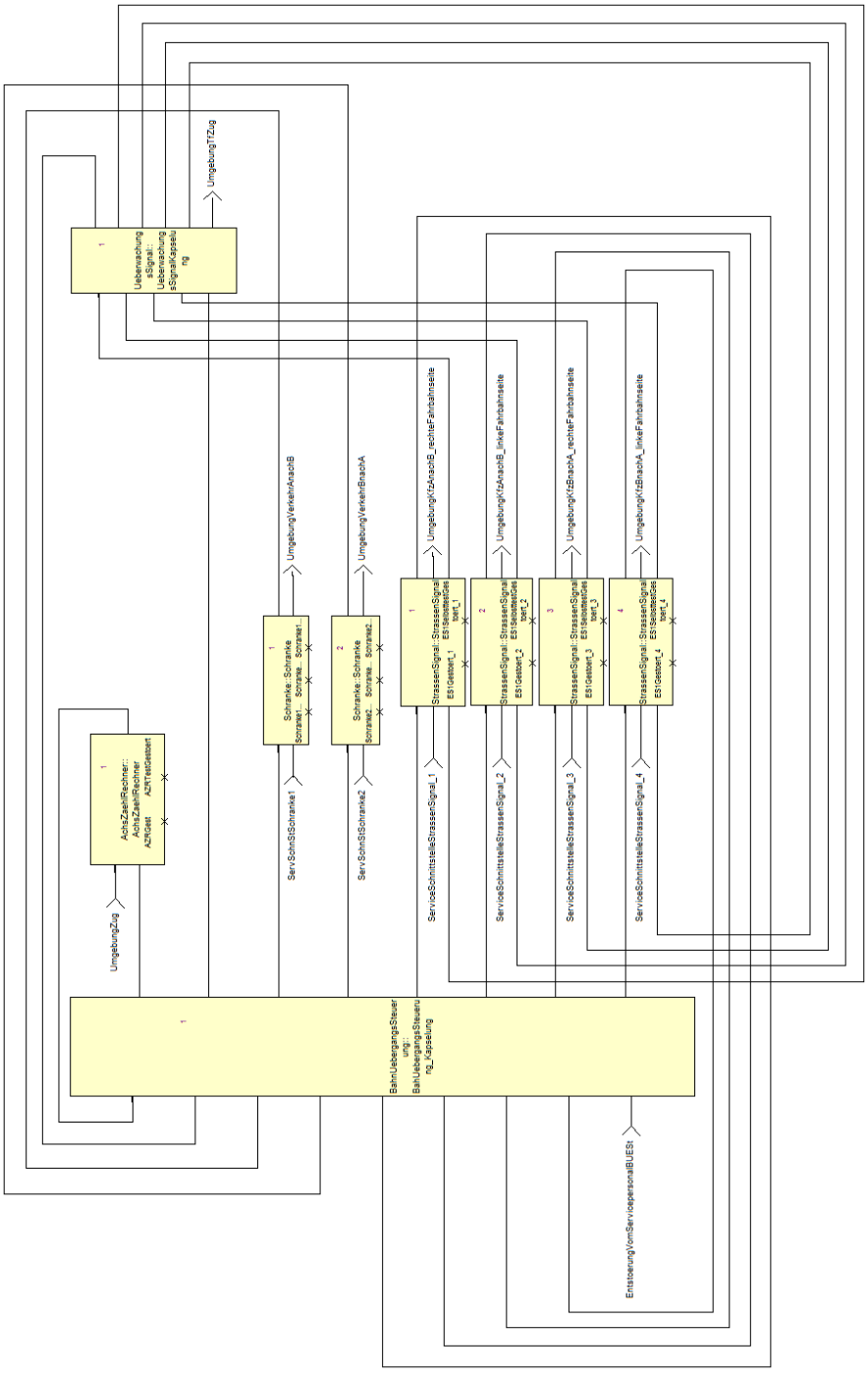

Figure C.2: System architecture of the level crossing system.

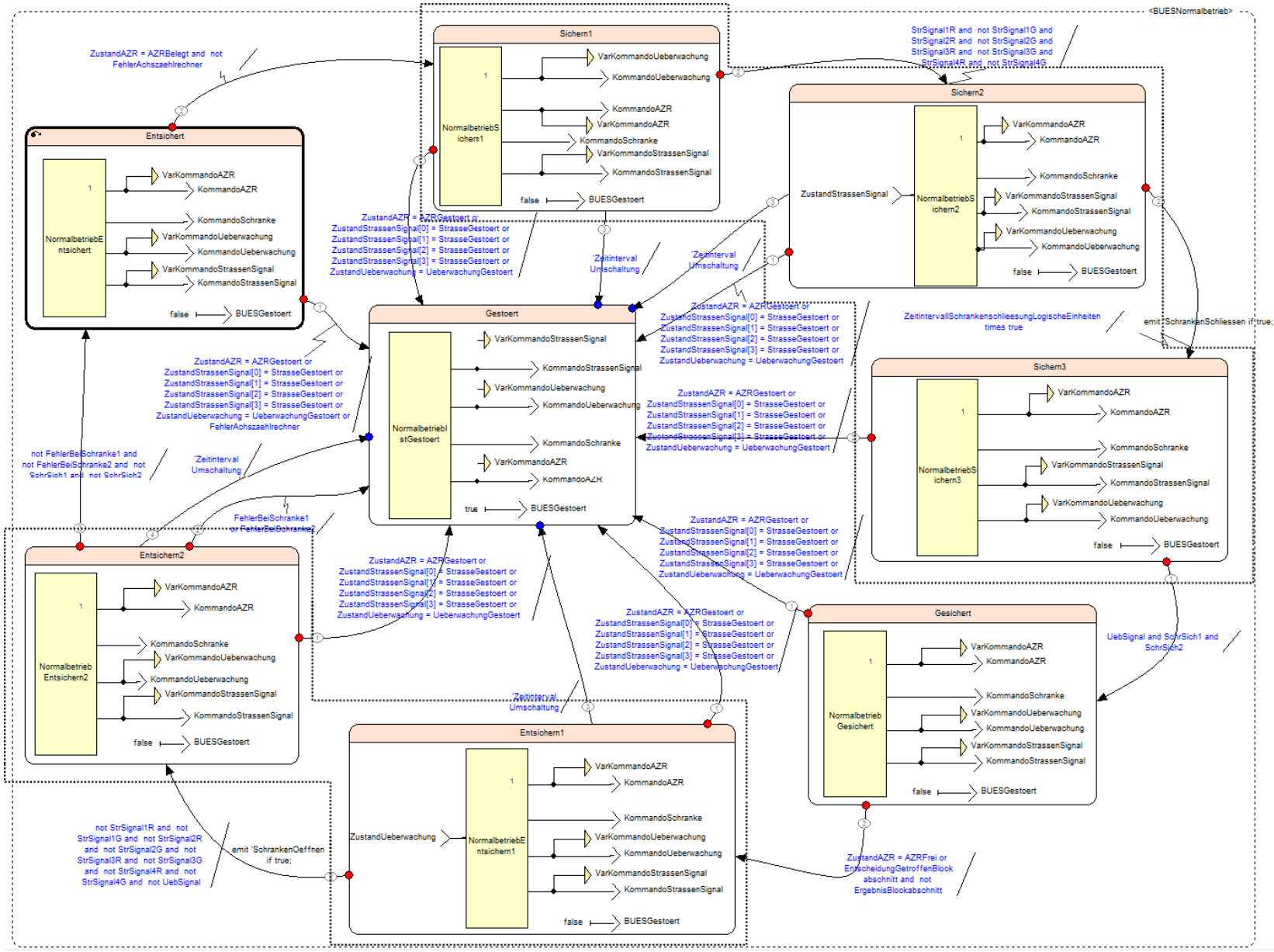

Figure C.3: Subcomponent ”BahnUebergangsSteuerung” including provided and required interfaces.