Interactive Verification of Statecharts

Andreas Thums, Gerhard Schellhorn, Frank Ortmeier, and Wolfgang Reif

Lehrstuhl Softwaretechnik,
Universität Augsburg, 86135 Augsburg, Germany
{thums,schellhorn,ortmeier,reif}@informatik.uni-augsburg.de

Abstract. In this paper, we present an approach to the interactive verification of statecharts. We use STATEMATE statecharts for the formal specification of safety critical systems and Interval Temporal Logic to formalize the proof conditions. To handle infinite data, complex functions and predicates, we use algebraic specifications.
Our verification approach is a basis for the aim of the project ForMoSA to put safety analysis techniques on formal grounds. As part of this approach, fault tree analysis (FTA) has been formalized yielding conditions that can be verified using the calculus defined in this paper. Verification conditions resulting from the formal FTA of the radio-based level crossing control have been successfully verified.

1 Introduction

We present an approach which aims to support the interactive verification of (safety) properties for concurrent, reactive systems.

We chose STATEMATE statecharts as modelling notation since they are broadly accepted, have a formal semantics, and STATEMATE [HLN+90] is widely used in industrial practice as a specification tool. Our use of statecharts overcomes the problem that typically engineers use more complex semi-formal languages than the fully formal models (usually automata) used by verification engineers.

Safety properties are stated in a first-order variant of Interval Temporal Logic (ITL, [Mos85, CMZ02]). This logic is expressive enough to describe most safety relevant properties. In particular a standard safety analysis technique - fault tree analysis (FTA) - has been formalized [STR02]. Formal FTA allows to rigorously prove cause-consequence relationships between individual component faults and failure of the whole system (see also [OTSR04] in this volume and [OT02]). Statechart verification is the basis for proving such dependencies.

We use the KIV system [BRS+00] as an implementation platform for the developed proof calculus. The proof strategy in KIV is symbolic execution. Symbolic execution is an intuitive proof method widely used for the interactive verification of sequential programs. KIV supports Dynamic Logic (DL, [Har84]) for proving properties of sequential programs by symbolic execution. This proof method has been extended to interval temporal logic and parallel programs [BDRS02] as well. The main contribution of this paper is to integrate proof

support for STATEMATE statecharts. This allows us to directly use the model used by the engineers as the formal system model.

In Sect. 2 we present a small example to explain the basics of STATEMATE statecharts. This example is also used in Sect. 3 to demonstrate the proof strategy of symbolic execution informally. The logical foundations of ITL are given in Sect. 4. They form the basis for embedding statecharts in Sect. 5. The main part of this paper is the presentation of the proof calculus for statecharts in Sect. 6. Sect. 7 gives a statechart model of a radio-based level crossing control. Algebraic specifications over integers are used to specify the velocity and braking behavior of a train. For this specification we have verified the proof obligations of a formal FTA for the hazard "collision on crossing". Finally, Sect. 8 concludes the paper and gives an outlook to future work.

2 Example: Automated Light Control

In the following, the example of an automatic light control will be used to explain important statechart terminology. The statechart in Fig. 1 models an automated

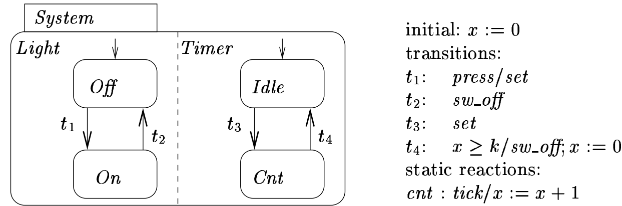


Fig. 1. Automated Light control

light control, which switches off the light k minutes after the light has been switched on. Initially, the light is *Off*, the timer is *Idle*, and $x = 0$. When the *press* event is active, the transition t_1 switches the light on, generates the event *set* to enable transition t_3, and starts the timer. In state *Cnt*, the timer x is incremented through the *tick* event (*tick* is generated, when the system clock advances) and finally, when x is greater or equal to k, the timer leaves *Cnt* and generates a *sw_off* event (t_4), to switch off the light (t_2).

The model of light control SYS consists of an *and*-state *System*, which executes the *or*-states *Light* and *Timer* in parallel. *Off*, *On*, *Idle*, and *Cnt* are *basic*-states, which do not encapsulate any sub-charts. $states(SC)$ is the set of states of a statechart SC and $states(SYS) = \{System, Light, On, Off, Timer, Idle, Cnt\}$ the states of the light control. Sub-charts of a state are computed by the function $childs : states(SC) \rightarrow \wp(states(SC))$, so the $childs$ relation describes a tree of states, with the root $root(SC)$ ($root(SYS) = System$, $childs(System) =$

$\{Light, Timer\}$). Statecharts define *termination*-states, as well. *Termination*-states stop the execution of the statechart, but are not used in the example. The mode of a state is computed by $mode : states(SC) \to \{and, or, basic, term\}$, e.g. $mode(System) = and$.

The structure of a statechart defines a consistency predicate $cons(SC)$ over states. If an *and*-state is active, every sub-state is active, as well, and if an *or*-state is active, exactly on of its sub-states is active. A configuration which fulfills this requirement is called *consistent.* E.g., a configuration, where *Off* and *On* is active, is inconsistent, but *Off* and *Cnt* describe a consistent configuration.

A transition is labeled with a guard and an action. $trans(SC)$ is the set of all transitions of a statechart SC. $source(t)$ computes the source state of a transition t, $target(t)$ the target state, $guard(t)$ the guard and $action(t)$ the actions. A transition can be executed, if $source(t)$ is active and $guard(t)$ holds. Then $action(t)$ is executed and the state $target(t)$ is entered. For t_4, $source(t_4) = Cnt$, $target(t_4) = Idle$, $guard(t_4) = x \geq k$, and $action(t_4) = sw_off; x := 0$.

The state transition from *Off* to *On* describes a so called *micro*-step. This micro-step triggers the transition t_3 and a second micro-step is executed. The event *press* activated a *chain reaction* of micro-steps. If no more micro-steps are possible, a *macro*-step, marked with the *tick* event, reads in new input values from the environment and time passes. While micro-steps are executed, no input events are considered.

In state *Cnt*, the timer x is incremented in every macro-step through a static reaction. The set of all static reactions is $sreactions(SC)$. Static reactions are assigned to a state and $sreact : states(SC) \to \wp(sreactions(SC))$ computes all static reactions of one state. Like transitions, static reactions sr are labeled with a guard $guard(sr)$ and an action $action(sr)$. The action is executed, if the corresponding state of the static reaction is active, not exited, and the guard holds.

3 Symbolic Execution

In this section we give an informal description of the proof strategy *symbolic execution* to prove properties of statecharts. If we want to prove a temporal logic property for the light control, e.g. that the counter is never greater than k, we execute the statechart and show, that in every (reachable) configuration the property holds. With an inductive argument we prove the property for infinite system traces (see Sect. 4 for details). We describe valuations of variables symbolically by equations, e.g. $x \leq k$, and therefore call the proof strategy *symbolic execution.* In our approach, states and events are boolean variables and their values are described symbolically, as well. A state or an event is active, if the corresponding variable is true.

Let us consider the statechart of the light control. In Fig. 2 we execute the statechart and compute the successor configurations, depending on the active events. The stars $\star$ mark the currently active states, the equations beneath the chart the values of the variable x, and the transitions between state configura-

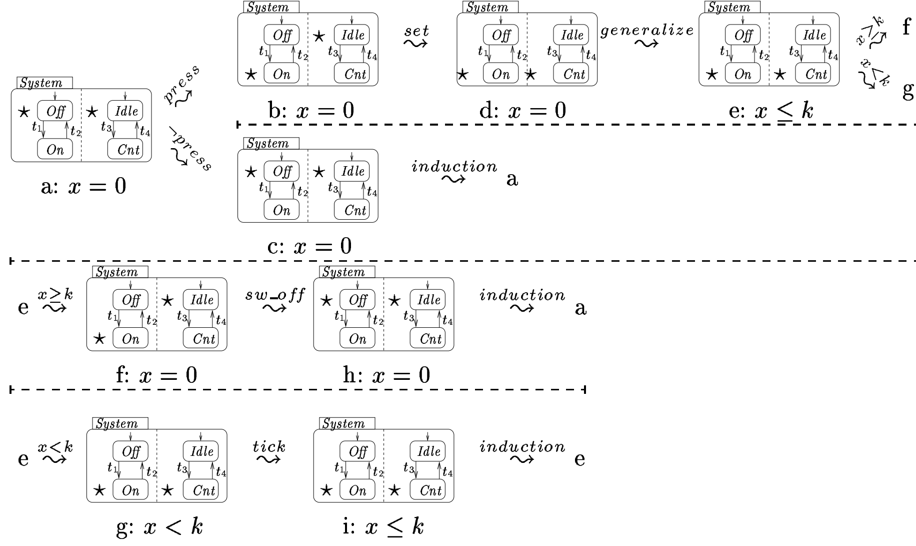


Fig. 2. symbolic execution of statecharts

tions the active events. If we start in an initial state configuration, we have two successors, depending on the *press* event. If *press* holds we step into the state *On* (chart b) and generate the event *set*. Otherwise, we stay in *Off* and get the initial configuration, again (chart c). With an inductive argument, we close this branch of the proof. In chart b, again $x = 0$ and the generated *set* event causes a state transition to chart d. In chart d, the state *Cnt* is active and increases x until $x \geq k$. We generalize $x = 0$ to $x \leq k$ (chart e) to prove that the invariant $x \leq k$ holds for the state *Cnt*. This condition allows two transitions. If $x < k$, we stay in *Cnt*, no further micro-step is possible and a macro-step generates the *tick* event (chart g). The *tick* event enables the static reaction *cnt*, which increases x. After increasing $x < k$, $x \leq k$ holds, chart i equals to chart e, and we can close this proof branch by induction. Otherwise, if $x \geq k$, state *Cnt* will be exited, state *Idle* entered (chart f), the event *sw_off* generated, and x set to 0 by transition t_4. Finally, *sw_off* triggers transition t_2 to enter state *Off*. We reach chart h, which is equal to chart a and again use induction to finish the proof. Because $x \leq k$ holds from chart a to chart i, the property "x is never greater than k" is proven.

Partially specified State Configurations In the previous proof, we generalized the condition $x = 0$ to $x \leq k$. Analogously, we often want to prove properties over statecharts, where the current state configuration is not unambiguously given. We call such configurations *partially specified state configurations.*

Again, we want to prove the condition, that x is never greater than k. Because x only changes in the chart *Timer*, it is no difference, if *On* or *Off* is

active. Because active (and inactive) states are described symbolically (e.g. *On* = true), we can generalize state configurations by omitting informations about states. If we prove conditions with partially specified state configurations (no information, if *On* or *Off* are active or not), a generalized induction hypothesis is possible (we can apply it in *On* and *Off*) resulting in shorter proofs. However, the computation of successor states gets more complex, since we have to consider transitions for both source state *On* and *Off* (see Sect. 6.1).

4 The Temporal Logic Framework

The basis of our approach is Interval Temporal Logic (ITL, [Mos85]). We use a first order extension based on algebraic specifications [Wir90] and consider finite and infinite intervals as described in [CMZ02]. The semantics is based on intervals I (in the following also called traces) which are finite or infinite sequences of states (also called valuations) $I = (\sigma_0, \ldots)$. Every valuation σ_i maps unprimed variables $\sigma_i(x)$ and primed variables $\sigma_i(x')$ to values of our domain. In a trace, the values of the primed variables are equal to the values of the unprimed variables in the next state $\sigma_i(x') = \sigma_{i+1}(x)$. Flexible variables may have different values in each state, while for rigid variables $\sigma_i(x)$ must be the same for all i. Function and predicate symbols are rigid and are interpreted using algebras (see for example [SA91]). Possible algebras are given as the (loose) semantics of algebraic specifications.

As temporal operators we use φ **;** ψ (chop - the interval can be split in two subintervals, such that φ holds in the first one and ψ holds in the second), $\Box\varphi$ (always), $\Diamond\varphi$ (eventually), $\circ\ \varphi$ (strong next - there exists a next state and φ holds then), $\bullet\ \varphi$ (weak next - if there exists a next state, then φ holds there)), φ **until** ψ (until), and others with their standard semantics in ITL. Given an algebra, the semantics of a formula is a set of traces $I = (\sigma_0, \sigma_1, \ldots)$. E.g., $\mathcal{A}, I \models \Box\varphi$, if and only if for every $i \leq \mathit{length}(I)$ and $I_i := (\sigma_i, \sigma_{i+1}, \ldots)$: $\mathcal{A}, I_i \models \varphi$. This allows to view statecharts as a special kind of formula, because they define sets of traces, as well. Predicate logic formulas Φ only depend on the first state of an interval and $\mathcal{A}, I \models \Phi :\Leftrightarrow \mathcal{A}, \sigma_0 \models \Phi$. If the algebra $\mathcal{A}$ is not relevant for the current consideration, we omit $\mathcal{A}$.

Sequential Programs To define transactions and static reactions we use abstract sequential programs. Their effexts are expressed using Dynamic Logic (DL, [Har84]) in combination with the temporal logic framework. In the following example we require variable x' to be equal to the value of variable x after a program has been executed.

$$\langle \textbf{if } y = 0 \textbf{ then } x := 1 \textbf{ else } x := 2 \rangle\ x' = x$$

Here, x' is either 1 or 2 depending on y. Parallel assignments will be used in the following. A valid formula is e.g.

$$x = 1 \wedge y = 2 \rightarrow \langle x, y := y, x \rangle\ x = 2 \wedge y = 1.$$

Semantically, the program of a DL operator is used to modify the first valuation σ_0 of a trace, the following valuations $\sigma_1, \ldots$ (if any) are untouched.

$$(\sigma_0, \sigma_1, \ldots) \models \langle\alpha\rangle\ \varphi \ :\Leftrightarrow \quad \text{there exists } \tau \text{ with } \sigma_0[\![\alpha]\!]\tau \text{ with } (\tau, \sigma_1, \ldots) \models \varphi$$

where $\sigma_0[\![\alpha]\!]\tau$ is the input/output semantics of program α with input valuation σ_0 and output valuation τ.

Sequent Calculus We construct proofs using a sequent calculus. Proof rules for predicate logic are standard. For DL we employ rules to symbolically execute the sequential programs. For example the two rules

$$\frac{\varphi_x^\tau, \Gamma \vdash \Delta}{\langle x := \tau\rangle\ \varphi, \Gamma \vdash \Delta}\ \textit{assign left} \qquad \frac{\varepsilon, \langle\alpha\rangle\ \varphi, \Gamma \vdash \Delta \quad \langle\beta\rangle\ \varphi, \Gamma \vdash \varepsilon, \Delta}{\langle \textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta\rangle\ \varphi, \Gamma \vdash \Delta}\ \textit{if left}$$

are used to execute assignments and conditionals by reducing the conclusion to the premise. Here, Γ denotes all other premises and Δ the rest f the conclusions. For a full set of rules for Dynamic Logic see for example [HRS88].

Rules for Temporal Logic The same strategy of symbolic execution is applied to temporal operators. Our first goal is to construct – for each temporal formula – separate formulas restricting the first valuation and the rest of the trace. For example $\Box\varphi$ in the succedent is treated as follows.

$$\frac{\Gamma \vdash \varphi, \Delta \quad \Gamma \vdash\ \bullet \Box\varphi, \Delta}{\Gamma \vdash \Box\varphi, \Delta}\ \textit{always right}$$

In the first premise, we prove that φ holds in the first state, in the second, we establish the property for the rest of the trace. This is what we call unwinding of temporal operators, which is comparable to executing programs. Unwinding $\Box\varphi$ and $\Diamond\varphi$ is straightforward, for more details on unwinding $\varphi\ ;\ \psi$ and others, see [BDRS02].

We unwind temporal operators until every temporal formula Γ and Δ is prefixed with a next operator and all other formulas γ and δ are formulas in predicate logic involving unprimed and primed variables. Then we can advance one step in the trace by applying rule *step*.

$$\frac{\gamma_0, \Gamma \vdash \delta_0, \Delta}{\gamma,\ \circ\, \Gamma \vdash \delta,\ \bullet\, \Delta}\ \textit{step}$$

Here γ_0 and δ_0 are obtained from γ and δ by replacing all unprimed variables v with new variables v_0 and all primed variables v' with their unprimed version v. The leading next operators are removed. Thus we have stored the values of variables of the initial state of the trace in new variables v_0 and advanced one step in the trace by removing all primes and next operators.

Induction Since traces may be arbitrary long or even infinite, it is not feasible, to execute the whole trace. Therefore, we need induction. The basic idea is to advance in the trace until a state of the statechart is repeated and some value decreased. In this case we have executed a loop and if we can show an invariant property, the proof can be finished by induction. To prove an always property, we use the fact that $\neg \Box \varphi \leftrightarrow \Diamond \neg \varphi$ to derive the following induction rule. The rule

$$\frac{N = N' + 1 \textbf{until} \neg \varphi, n = N, \text{IndHyp}(n), \Gamma \vdash \Delta}{\Gamma \vdash \Box\varphi, \Delta} \; \textit{ind. always}$$

uses noetherian induction to prove safety properties $\Box\varphi$ with

$$\text{IndHyp}(n) :\equiv \forall\, m < n.\ Cl_{\forall}^{m}(\bigwedge \Gamma \wedge m = N \rightarrow \bigvee \Delta)$$

and $Cl_{\forall}^{m}(\varphi)$, the all closure of φ excepting m. Informally we prove, that $\Box\varphi$ is false, if it is possible to decrement a variable N (note that $N = N' + 1$ is equivalent to $N' = N - 1 \wedge N \neq 0$) until we reach a valuation, where φ does not hold. The initial value of N is the number of steps until we reach a situation, where φ does not hold. Inductive proof are by contradiction and we refer to [Bal04] for details.

5 Embedding Statecharts in ITL

In this section we describe the integration of statecharts into the ITL framework. The key idea is to treat a statecharts SC as a formulas, with the idea that I $\models$ SC holds if I is a trace of SC. A configuration of a statechart is represented as a valuation of statechart variables $\textit{variables}(SC)$. The statechart variables are flexible variables and, therefore, additional primed variables v' exists for every $v \in \textit{variables}(SC)$. In addition to the data variables $\textit{vars}(SC)$, boolean variables for each state $\textit{states}(SC)$ and each event $\textit{events}(SC)$ are required, describing whether the corresponding state/event is active or not. The input variables $\textit{events}_{env}(SC) \cup \textit{vars}_{env}(SC)$, with $\textit{events}_{env}(SC) \subseteq \textit{events}(SC)$ and $\textit{vars}_{env}(SC) \subseteq \textit{vars}(SC)$, are variables, which the systems environment may modify. Local events are $\textit{events}_{loc}(SC) := \textit{events}(SC) \setminus \textit{events}_{env}(SC)$, local variables are $\textit{vars}_{loc}(SC) := \textit{vars}(SC) \setminus \textit{vars}_{env}(SC)$. A statechart step describes a transition from one valuation to another, corresponding to the transition relation of the statechart. A trace of a statechart is a sequence of valuations $(\sigma_0, \sigma_1, \ldots)$ where all relations $\sigma_i\ \rho\ \sigma_{i+1}$ correspond to the transition relation ρ described by the statechart. The semantics of a statechart is defined as all such sequences of states.

Semantics The statechart semantics depends on the definition of a statechart step. Our formalism supports the STATEMATE semantics of statecharts [HN96] and is based on the operational semantics presented in [DJHP98]. Two important extensions are necessary to integrate statecharts into our ITL framework. In

[DJHP98], a statechart step directly computes the successor valuation of the statechart, whereas in our setting, a step has to compute the values of the primed variables. Then the *tl step* rule computes the actual transitions to the successor valuations. The second extension is the following: [DJHP98] defines the semantics of statecharts as the set of traces $(\sigma_0, \ldots, \sigma_n)$, where the transition relation of the statechart ρ holds for every step, i.e. $\sigma_i \; \rho \; \sigma_{i+1}$, the valuation σ_0 is an initial valuation (the initial states are active). Traces are required to be maximal, i.e. for finite traces: $\sigma_n \notin dom(\rho)$. This is not sufficient for our calculus, since symbolic execution leaves the initial state. Therefore, we generalize the definition of the statechart semantics to traces with a *consistent* initial valuation σ_0. A valuation is consistent, if it assigns true to exactly one sub-chart of every or-state. The set of all maximal statechart traces is defined as

$$traces_{\mathcal{A}}(SC) := \{(\sigma_0, \ldots, \sigma_n) \mid n \in \mathbb{N}_\infty, \sigma_i \; \rho \; \sigma_{i+1}, \sigma_n \notin dom(\rho), \sigma_0 \text{ consistent}\}$$

Note, that the algebra $\mathcal{A}$ used in the definition makes it dependent on the underlying algebraic specification which defines the data part of the statechart. Because all initial valuations are consistent, every initial trace is in $traces_{\mathcal{A}}(SC)$. This definition enables us to prove properties for statecharts, which do not start in an initial configuration. Such properties are often useful as lemmas.

Embedding The semantics of statecharts is the set of maximal traces of the statechart and the semantical integration into the ITL framework is straightforward.

$$\mathcal{A}, I \models SC :\Leftrightarrow I \in traces_{\mathcal{A}}(SC)$$

In this paper, we will not present the definition of ρ, because it basically depends on the definitions in [DJHP98]. Instead, we show in Sect. 6 how to compute the successor valuations in the calculus. A formal definition of ρ and a correctness proof for the calculus is given in [Thu04].

6 Statechart Calculus

Unwinding a statechart step can be split up into two tasks. First, we have to compute all possible steps by $Steps(SC, \Gamma \vdash \Delta)$.Second, for each possible step $stp \in Steps(SC, \Gamma \vdash \Delta)$ we have to compute the effects of the active transitions and static reactions contained in *stp*. These actions modify variables and generate events. Because transitions leave and enter states, the state configuration changes, too. The rule scheme for unwinding statecharts is defined as

$$\frac{\bigvee_{stp_i \in Steps(SC, \Gamma \,\vdash\, \Delta)} exec(stp_i), \Gamma \vdash \Delta}{SC, \Gamma \vdash \Delta} \; sc\ unwind, \tag{1}$$

with $exec(stp) := cond(stp) \wedge step(stp) \wedge \circ\, nxt(stp)$. Because our approach allows complex predicates over algebraic specifications as guards, we cannot always

determine automatically if a guard is satisfied or not. This is in contrast to approaches which allow finite data structures only, where a model checker can decide such conditions. Therefore, we compute activation conditions $cond(stp)$ for a possible statechart step. Only if $cond(stp)$ holds, the step is executed. Otherwise, the condition is contradictory to the other preconditions.

$step(stp)$ executes a single statechart step and $nxt(stp)$ computes the active states of the successor valuation. Together, they implement the transition relation ρ, defined by the statechart. The formal definitions of $cond(stp)$, $step(stp)$, and $nxt(stp)$ follow in Sect. 6.2.

6.1 Computing Possible Steps

A step consists of maximal sets of non-conflicting active transitions. Transitions are in conflict, if they leave the same state. We require that the sets of transitions are maximal to guarantee, that an active transition is actually executed, unless a conflicting transition is executed in this step. Because of concurrent states (and-states), a maximal set of non-conflicting active transitions is not necessarily a singleton.

The STATEMATE semantics of statecharts defines a top-down priority for conflicting transitions: assume an active or-state s with an active sub-state s_1 and two active transitions t with source state s and t_1 with source state s_1. Because the execution of t leaves s and all its sub-states (even s_1), t and t_1 are in conflict. The priority rule of STATEMATE statecharts solves this conflict in favor of t.

Analogously to [DJHP98], we present a computation of steps, which respects all these requirements. Roughly speaking, the algorithm computes the transition set in a top-down fashion. For an and-state, all possible steps from the sub-states have to be considered. The algorithm computes the Cartesian product of the steps from the sub-states. For an or-state, the algorithm computes a set of active transitions. If this set is empty, the transitions of the active sub-state are considered (the priority rule is respected), otherwise all these transitions are in conflict and the result is a set of steps, where every step is a singleton set of one active transition (non-conflicting is respected). Finally, a basic-state yields an empty step.

Depending on the current sequent $SC, \Gamma \vdash \Delta$, a state s and an activation condition g the function

$$steps : Sequent \times states(SC) \times Fma \to (Fma, \wp(trans(SC)))$$

computes all possible steps $stp = (g_T, T)$ with activation condition g_T for the maximal non-conflicting transitions T. Here, Fma denotes the set of all interval temporal logic formulae. Based on this auxiliary function

$$Steps(SC, \Gamma \vdash \Delta) := steps(\Gamma \vdash \Delta, root(SC), \mathrm{true})$$

computes all possible statechart steps. $steps(\Gamma \vdash \Delta, s, g)$ is defined as

1. $mode(s) \in \{basic, term\}$:

   $steps(\Gamma \vdash \Delta, s, g) := \{(g, \emptyset)\}$

2. $mode(s) = and$:

   Let $\{s_1, \ldots, s_n\} = childs(s)$, then

$$steps(\Gamma \vdash \Delta, s, g) := \\ \{(g_1 \wedge \ldots \wedge g_n, \bigcup_{i=1}^{n} T_i) \mid (g_i, T_i) \in steps(\Gamma \vdash \Delta, s_i, s_i \wedge g)\},$$

3. $mode(s) = or$: Let

   - $S' = \{\tilde{s} \mid \tilde{s} \in childs(s)$ and $\Gamma \vdash \Delta, \neg\, \tilde{s}$ is not provable$\}$,
   - $T_{\tilde{s}} = \{t_{\tilde{s}_1}, \ldots, t_{\tilde{s}_k}\} = \{t \mid source(t) = s\}$ and
   - $T = \{t_1, \ldots, t_k\} = \bigcup_{\tilde{s} \in S'} T_{\tilde{s}}$.

   Every transition in T has the same priority and they are in conflict.

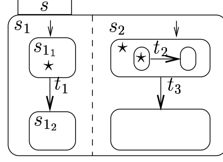
$$steps(\Gamma \vdash \Delta, s, g) := \\ \{(g_t, \{t\}) | t \in T \text{ and } g_t := g \wedge guard(t) \wedge source(t)\} \cup \\ \bigcup_{s' \in S'} steps(\Gamma \vdash \Delta, s', g \wedge s' \wedge \neg guard(t_{s'_1}) \wedge \ldots \wedge \neg guard(t_{s'_k})) \qquad (2)$$

A basic state cannot execute any transition, so the result is a set of an empty set. An and-state executes every transitions of its sub-states in parallel, so we have to compute the Cartesian product of the transitions from the sub-states. The main difference to the step computation in [DJHP98] is in item 3., where an *or*-state is considered. First, it has to be determined, which states are possible source states for active transitions. Since we allow partially specified state configurations, active states cannot be determined fully automatic. We approximate the set by collecting those states $\tilde{s}$ where $\Gamma \vdash \Delta, \neg\tilde{s}$ is not provable using the simplifier of KIV in S'. The set $T_{\tilde{s}}$ computes the transitions with source state $\tilde{s}$ and the set T contains all possibly active transitions.

All transitions in T are in conflict (in an or-state, only one transition can be executed) and can be executed, if the guard is evaluated to true and its source state is actually active. Therefore, the activation condition g_t for a transition t is $g_t := g \wedge guard(t) \wedge source(t)$. The precondition g describes, that no transition of a super-state is active.

If a state $\tilde{s}$ is active, but none of the guards of any transitions $t \in T_{\tilde{s}}$ holds, transitions from sub-states of $\tilde{s}$ can be executed. We add the condition, that no guard of $t \in T_{\tilde{s}}$ is enabled to the activation condition for the recursive call of *steps* in equation (2).

Example 1 Step Computation (I)



Consider the statechart on the left where $\star$ marks active states. Let $guard(t_1) = g_1$, $guard(t_2) = g_2$, and $guard(t_3) = g_3$ be the guards of the transitions. We only add active states to the activation condition, if necessary.

$$\begin{aligned}
&steps(\Gamma \vdash \Delta, s_1, \text{true}) = \{(g_1, \{t_1\}), (\neg g_1, \emptyset)\}\\
&steps(\Gamma \vdash \Delta, s_2, \text{true}) = \{(g_3, \{t_3\}), (\neg g_3 \wedge g_2, \{t_2\}), (\neg g_3 \wedge \neg g_2, \emptyset)\}\\
&Steps(\Gamma \vdash \Delta) = steps(\Gamma \vdash \Delta, s, \text{true}) =\\
&\quad \{(g_1 \wedge g_3, \{t_1, t_3\}), (\neg g_1 \wedge g_3, \{t_3\}), (g_1 \wedge \neg g_3 \wedge g_2, \{t_1, t_2\}),\\
&\quad (\neg g_1 \wedge \neg g_3 \wedge g_2, \{t_2\}), (g_1 \wedge \neg g_3 \wedge \neg g_2, \{t_1\}), (\neg g_1 \wedge \neg g_3 \wedge \neg g_2, \emptyset)\}
\end{aligned}$$

Now, let us consider the state s with a generalized state configuration. We abstract from the active states in s_1. If either s_{1_1} or s_{1_2} is active, we get:

$$\begin{aligned}
&steps(\Gamma \vdash \Delta, s_1, \text{true}) = \{(g_1 \wedge s_{1_1}, \{t_1\}), (\neg g_1 \wedge s_{1_1}, \emptyset), (s_{1_2}, \emptyset)\}\\
&steps(\Gamma \vdash \Delta, s_2, \text{true}) = \{(g_3, \{t_3\}), (\neg g_3 \wedge g_2, \{t_2\}), (\neg g_3 \wedge \neg g_2, \emptyset)\}\\
&Steps(\Gamma \vdash \Delta) = steps(\Gamma \vdash \Delta, s, \text{true}) =\\
&\quad \{(g_1 \wedge s_{1_1} \wedge g_3, \{t_1, t_3\}), (\neg g_1 \wedge s_{1_1} \wedge g_3, \{t_3\}), (s_{1_2} \wedge g_3, \{t_3\}), (g_1 \wedge s_{1_1} \wedge\\
&\quad \neg g_3 \wedge g_2, \{t_1, t_2\}), (\neg g_1 \wedge s_{1_1} \wedge \neg g_3 \wedge g_2, \{t_2\}), (s_{1_2} \wedge \neg g_3 \wedge g_2, \{t_2\}), (g_1 \wedge\\
&\quad s_{1_1} \wedge \neg g_3 \wedge \neg g_2, \{t_1\}), (\neg g_1 \wedge s_{1_1} \wedge \neg g_3 \wedge \neg g_2, \emptyset), (s_{1_2} \wedge \neg g_3 \wedge \neg g_2, \emptyset)\}
\end{aligned}$$

Example 2 Step Computation (II)
Consider the statechart *SYS* of the timer example and assume that the initial states *Off* and *Idle* are active. If we want to prove $\Box x \leq k$, we get the sequent *SYS*, *On*, *Idle* $\vdash \Box x \leq k$ and the step algorithm computes the following possible steps:

$$\begin{aligned}
steps((On, Idle, \vdash \neg \Box x \leq 6), \text{true}) := \{&(press \wedge set, \{t_1, t_3\}),\\
&(press \wedge \neg set, \{t_1\}), (\neg press \wedge set, \{t_3\}), (\neg press \wedge \neg set, \emptyset)\}
\end{aligned}$$

The presented step computation extends the one, presented in [DJHP98]. We added the possibility to compute and return activation conditions for steps. This extension is necessary, because we have to consider guards for transitions, which cannot be decided automatically. A second extension is the possibility to cope with partially specified state configurations. Therefore, we have to add the source state of a transition to its activation condition.

We also consider static reactions and inter level transitions in our approach, but omitted these concepts for this presentation. For details we refer to [Thu04].

6.2 Step-Execution

We define the execution of a statechart step with sequential programs. The programs 'implement' the step semantics. We sequentialize the execution of parallel transitions according to the semantics from Damm et al. [DJHP98]. We respect the execution of parallel transitions, although we implement the step execution

with sequential programs. To solve read conflicts, where an action reads a variable already changed by a parallel action, we copy variables v in a copy $\tilde{v}$. In the following, we assume for every flexible variable $v \in \mathit{variables}(SC)$ an additional auxiliary variable $\tilde{v}$ and that the actions of transitions assign their values to these auxiliary variables.[1] To solve write conflicts, where different actions change the value of one variable, we introduce nondeterminism, where every action 'wins' once (see below).

We distinguish between micro- and macro-steps. A macro-step is executed, if an empty set of enabled transitions is computed.

$$\mathit{step}((g,T)) := \begin{cases} T = \emptyset, & \mathit{step}_{makro}(T) \\ \text{otherwise}, & \mathit{step}_{mikro}(T) \end{cases}$$

Micro-Step A micro-step $\mathit{step}_{mikro} : \mathit{trans}(SC) \rightarrow \mathit{Fma}$ is defined as

$$\mathit{step}_{mikro}(T) := \bigvee_{\underline{\alpha} \in perm(T)} \langle \mathit{copy}; \mathit{reset}^l; \mathit{reset}^e; \mathit{exec}(\underline{\alpha}) \rangle \mathit{set}.$$

As described above, we respect write conflicts through nondeterminism by executing every possible sequence of actions ($\underline{\alpha} \in \mathit{perm}(T)$, perm computes the permutations of the transition set T).

To solve read conflicts, we copy variables from $\underline{v} = \mathit{vars}(SC) \cup \mathit{events}(SC)$ to $\underline{\tilde{v}}$ with copy. Then, we reset every event with reset^l, by assigning false to the boolean variable, representing an event. To observe generated environment steps at the end of a macro-step, we reset them only in the first micro-step after a macro-step. Then tick holds. Resetting environment events is done with reset^e. Now, we can execute the actions. Because we already considered read and write conflicts, the execution can be sequential. Finally, set requires, that the primed variable (unprimed ones in the next configuration) get the values, computed before. The corresponding programs are defined as

$$\begin{aligned}
\mathit{copy} &:= \underline{\tilde{v}} := \underline{v} \text{ for } \underline{v} = \mathit{vars}(SC) \cup \mathit{events}(SC), \\
\mathit{reset}^l &:= \underline{\tilde{e}} := \text{false, for } \underline{e} = \mathit{events}_{loc}(SC) \cup \{\mathit{tick}\}, \\
\mathit{reset}^e &:= \textbf{if } \dot{\mathit{tick}} \textbf{ then } \underline{\tilde{e}} := \text{false for } \underline{e} = \mathit{events}_{env}(SC), \\
\mathit{exec}(\underline{\alpha}) &:= \alpha_1; \ldots; \alpha_n \text{ for } \underline{\alpha} = \alpha_1, \ldots, \alpha_n, \text{ and} \\
\mathit{set} &:= \underline{\tilde{v}} = \underline{v}' \text{ for } \underline{v} = \mathit{vars}(SC) \cup \mathit{events}(SC).
\end{aligned}$$

In addition to the given step computation, STATEMATE statecharts generate also events when states are entered and exited. We consider these events in our calculus, too, but have omitted this detail here (see [Thu04]).

Macro-Step A macro-step $\mathit{step}_{makro} : \mathit{trans}(SC) \rightarrow \mathit{Fma}$ is defined as

$$\mathit{step}_{makro}(\mathit{stp}) := \langle \mathit{copy}; \mathit{reset}^l \rangle \mathit{set}^e \wedge \mathit{set}^l$$

[1] Reading context variables $\$v$ from STATEMATE corresponds to reading $\tilde{v}$, reading 'normal' STATEMATE variables, corresponds to reading v in our setting.

Analogous to a micro-step, we copy the variable values and reset the events. Then, we label the macro-step with the *tick* event and require, that the primed variables for local events and variables values get the current values.

$$\begin{aligned} set^e &:= tick' = \text{true} \\ set^l &:= \tilde{\underline{v}} = \underline{v}' \text{ for } \underline{v} = events_{loc}(SC) \cup vars_{loc}(SC) \end{aligned}$$

A macro-step does not restrict the primed variables of environment events. A temporal logic step assigns arbitrary values to them to model new input to the environment variables.

State Configuration In addition to the step computation, we have to compute the active states for the successor valuation. They depend on the current transition set T. Let $S := entered(T) \cup active(T)$ be the set of active states in the next configuration, $\overline{S} := states(SC) \backslash (entered(T) \cup active(T))$ the set of inactive states. *entered* computes the states which will be entered by executing the transitions T, *active* the states, which were neither entered nor exited and *entered*, the states, which were entered through the current step. Then

$$nxt(T) := \bigwedge_{s \in S} s \wedge \bigwedge_{\overline{s} \in \overline{S}} \neg\, \overline{s} \wedge SC$$

computes the active states of the next configuration and requires that statechart formula SC holds again (and can be executed further).

Example 3 Step Execution
Let us consider the timer example once again and denote the statechart SYS. We want to prove the sequent $SYS, Off, Idle \vdash \Box x \leq k$, which we abbreviate to $SYS, \Gamma \models \varphi$. Then the step computation computes four possible steps and therewith four possible premises for the *sc unwind* rule.

$$\frac{\begin{array}{l} (1)\ (press \wedge set) \wedge step((press \wedge set, \{t_1, t_3\})) \wedge \circ nxt(\{t_1, t_3\}), \Gamma \vdash \varphi \\ (2)\ \ (\neg press \wedge set) \wedge step((\neg press \wedge set, \{t_3\})) \wedge \circ nxt(\{t_3\}), \Gamma \vdash \varphi \\ (3)\ \ (press \wedge \neg set) \wedge step((press \wedge \neg set, \{t_1\})\}) \wedge \circ nxt(\{t_1\}), \Gamma \vdash \varphi \\ (4)\ \ \ \ (\neg press \wedge \neg set) \wedge step((\neg press \wedge \neg set, \emptyset)) \wedge \circ nxt(\emptyset), \Gamma \vdash \varphi \end{array}}{SYS, \Gamma \vdash \varphi}$$

We only consider the cases 3 and 4 in detail. The 3rd case is a micro-step and the 4th case a macro-step. Copying variables and resetting the events is independent from the concrete step.

$$\begin{aligned} copy &= \widetilde{tick}, \widetilde{set}, \widetilde{sw_off}, \widetilde{x} := tick, set, sw_off, x \\ reset^l &= \widetilde{tick}, \widetilde{set}, \widetilde{sw_off} := \text{false}, \text{false}, \text{false} \\ reset^e &= \textbf{if } tick \textbf{ then } \widetilde{press} := \text{false} \end{aligned}$$

The micro-step has to execute the transition t_1 and assign the computed values from the step to the primed variables, to describe the valuation of the next step.

$$\begin{array}{ll} exec(< set >) & = \widetilde{set} := \text{true} \\ set & = \widetilde{tick}, \widetilde{set}, \widetilde{sw_off}, \widetilde{press}, \widetilde{x} = tick', set', sw_off', press', x' \end{array}$$

The whole micro-step results in

$$step_{mikro}((press \wedge \neg set, \{t_1\})) = \langle copy; reset^l; reset^e; exec(< set >)\rangle set.$$

The macro-step in the 4th premise resets every event and generates the *tick*-event. Because we do explicitly not assign any value to primed variables for environment variables and events, the following temporal logic step assigns an arbitrary value to environment variable *press*.

$$\begin{array}{ll} set^e & = tick' = \text{true} \\ set^l & = \widetilde{set}, \widetilde{sw_off}, \widetilde{x} = set', sw_off', x' \end{array}$$

We get the following macro-step.

$$step_{makro}((\neg press \wedge \neg set, \emptyset)) = \langle copy; reset^l\rangle set^e \ \wedge \ set^l.$$

6.3 Tool Support

We implemented specification and proof support for statecharts in KIV. The specifications are incorporated into the correctness management of the KIV system. If a statechart specification is changed, every lemma based on this specification becomes invalid.

Because our statechart semantics does not require to specify lemmas for statecharts, which start execution in an initial state, we are able to specify and prove lemmas for intermediate states. E.g., we can prove something like "if we are in state *Cnt*, we will stay there for k macro-steps, at most". We can use this statement in other proofs as a lemma and apply it, if we reach the state *Cnt*.

The proof strategy of symbolic execution leads to diagrammatic proofs. We have to unwind the statechart and execute the statechart step. Thereafter, we unwind the temporal operator, get a predicate logic proof condition for the current valuation and a temporal logic proof condition for the rest of the trace. If we have proven the condition for the current valuation, we turn to the successor valuation, where we can unwind the statechart again. This leads to proofs, following the execution of the statechart. This means, that we do not translate a statechart into a flat transition system, but preserve the structure of the statechart.

We automated this sequence of rule application through heuristics in the KIV system. We extend the heuristics for ITL verification, to apply the *sc unwind* rule. The resulting heuristics enable us to automate simple statechart proofs. As an example, the proof for the property $\Box x \leq t$ of the light control, as shown in Fig. 2 of Sect. 2, can be automated except for the generalisation step from $x = 0$ to $x \leq k$. If a concrete k is given, e.g. $k = 6$ the proof is fully automatic.

7 Example: Radio-Based Level Crossing Control

In this section we will describe the application of the presented calculus on a real world case study. This case study is a reference case study of the German research councils priority program 1064. we will briefly describe the problem, then present how it is modeled in statechart notion and finally give some verification results.

The German railway organization, Deutsche Bahn, prepares a novel technique to control level crossings: the decentralized, radio-based level crossing control (FunkFahrBetrieb, FFB [FFB96]). The main difference between this technology and the traditional control of level crossings is, that signals and sensors on the route are replaced by radio communication and software computations in the train and in the level crossing.

Instead of detecting an approaching train by a sensor, the train computes the position where it has to send a signal to secure the level crossing. Therefore, the train has to know the position of the level crossing, the time needed to secure the level crossing, and its current speed and position. When the level crossing receives the 'secure' command it switches on the traffic lights and closes the barriers. When they are closed, the level crossing is 'safe' for a certain period of time. The stop signal, indicating an insecure crossing, is also substituted by computation and communication. The train requests the status of the level crossing. Depending on the answer the train will brake or pass the crossing. A STATEMATE reference model for the radio based crossing control is presented in [KT02] and a safety analysis with fault trees in [TO03, RST00a].

7.1 Specification with Statecharts

We specified a version of the radio-based crossing control with statecharts in KIV. Fig. 3 shows the graphical representation of the specification, that focuses on the problem of closing the barriers early enough and braking in time, if something went wrong (the reference model also considers traffic lights, etc.). We also modeled faulty behavior, highlighted by the gray box, but concentrate on the functional behavior, at first.

The whole setting is specified with three parallel statecharts, one chart for every system component. The chart *train* specifies the behavior of the train. An approaching train sends a closing request to the crossing, if the predicate $close(pos, v, a_{brk}, lc)$ holds. pos indicates the current position of the train, v the current speed, a_{brk} the maximal deceleration, and the level crossing lc. The closing request $close_{snd}$ will be received ($close_{rcv}$) by the crossing if no communication error has happened. Then the crossing closes the barriers within tc_{cl} time units. Thereafter, the barriers are closed for maximal t_{max} time units or until the train passes the crossing.

The train sends a status request $status_{snd}$, which will also be delayed by the communication, and awaits an answer within a certain amount of time. If the crossing receives the status request $status_{rcv}$ and the barriers are closed (the state *closed* is active), the crossing acknowledges the status request (ack_{snd}). If

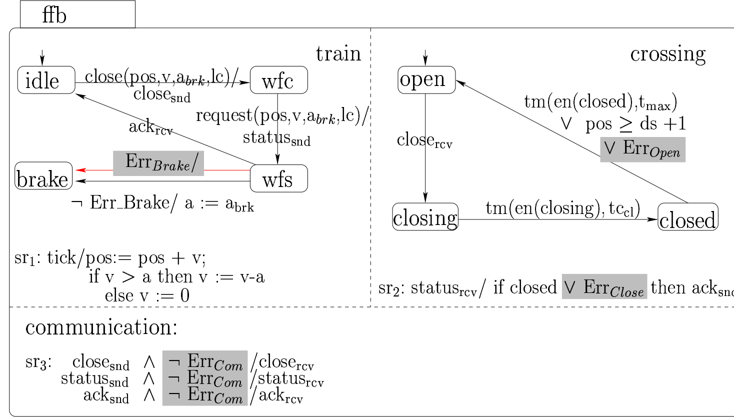


Fig. 3. Radio-Based Level Crossing Control

the train receives ack_{rcv} in time, it will pass the crossing, otherwise the train will brake and stop before the level crossing lc.

Now, let us consider faulty behavior. We modeled the failure of brakes (Err_{Brake}), the break down of the communication channel (Err_{Com}), a faulty acknowledgment of a status request (Err_{Close}) and an undesired opening of the barriers (Err_{Open}). Every fault is modeled by an indeterministic choice. E.g., the failure of the brakes are modeled by an additional transition from the state *wfs* to *brake*. In contrast to the 'correct' transition, this additional one does not set the deceleration to a_{brk}, so the train does not brake. The other failures are modeled analogously.

7.2 Algebraic Specification

The underlying data types for the radio-based crossing control system are algebraically specified using natural numbers. In addition, we specified the predicates $close(pos, v, a_{brk}, lc)$ and $request(pos, v, a_{brk}, lc)$ to compute the optimal time for closing the barriers and requesting the status of the crossing. These predicates rely on the braking distance $brake_d(v, a_{brk})$. Within each macro-step, the static reaction sr_1 computes the position and velocity of the train.

$$tick/pos := pos + v; \textbf{if } v > a \textbf{ then } v := v - a \textbf{ else } v := 0$$

With the deceleration $a > 0$ and the initial velocity v_0, the distance until the train stops is

$$brake_d(v_0, a) := v_0 + (\sum_{i=0}^{\lceil v_0/a \rceil} v_0 - i * a) \tag{3}$$

(with the distance $v_0 \frac{m}{s} * 1s = v_0 m$). Based on $brake_d$, we can define the predicates *close* and *request*

$$close(pos, v, a_{brk}, lc) := pos \geq lc - (brake_d(v, a_{brk}) + tc_{cl} * v) \text{ and} \tag{4}$$

$$request(pos, v, a_{brk}, lc) := pos \geq lc - brake_d(v, a_{brk}). \tag{5}$$

7.3 Formal Safety Analysis and Verification

In the ForMoSA project, we developed the safety analysis technique formal fault tree analysis (formal FTA, [STR02]). Formal FTA integrates classical fault tree analysis (FTA, [VGRH81]) with formal methods. For the radio-based crossing control we analysed the hazard "collision of the train on crossing". A detailled fault tree is described in [**?**]. 20 verification conditions are attached to this fault tree. Alltogether they guarantee that the hazard can happen only if one of the causes described by the leaves of the fault tree (e.g. failure of brakes Err_{Brake}) happens. Informally, each verification condition is of the form: "if some consequence happens, then the cause must have happened before or at the same time" (for a formal definition of the conditions, see [OTSR04] in this volume).

All verification conditions could be verified with KIV. The proofs typically proceed by induction over the number of steps it takes to reach the consequence (if the consequence is never reached on a trace, the condition is trivially true).

Some conditions are trivial to prove, but most have rather complex proofs with several hundred proof steps, so they are too complex to show them in detail. Two reasons contribute to the complexity: one is the indeterminism inherent in statecharts, the other is a lack of modular proofs: even though most conditions give properties of only one component (e.g. the train), all properties must still be proved over the full statechart. Research in modular proofs is still an important topic for further research.

As subtasks of the proofs algebraic properties of the definitions (3), (4) and (5) have to be shown: thereby it is proved that the requests to close the gates and the distance where to brake (when the gates does not acknowledge closing) are indeed a safe distance away from the crossing, such that the train is able to stop in time.

8 Conclusion and Future Work

We developed interactive proof support for statecharts with infinite data structures. We consider statecharts with finitely many states, but specify data, functions, and predicates by algebraic specifications. Guard conditions of transitions refer to data types using first-order logic. Statechart actions are described by sequential programs and proof conditions for statecharts are interval temporal logic formulas.

We support STATEMATE statecharts with their asynchronous macro-step semantics. As far as we know, this is the first approach which offers proof support for infinite statechart models. We defined a sequent calculus for verifying

statechart properties, which is implemented in the KIV system. The calculus has been successfully applied on the radio-based level crossing control.

Up to the present, we considered the verification of statecharts. Damm et al. [DJHP98] gave a modular semantics based on activities, where complex systems can be decomposed in modular sub-activities. Each activity is controlled by a statechart. Modularization is a prerequisite for proving properties for complex systems. Future work will be to extend our approach to modular statechart verification.

References

[Bal04] M. Balser. *Verifying Concurrent System with Symbolic Execution – Temporal Reasoning is Symbolic Execution with a Little Induction.* PhD thesis, University of Augsburg, Augsburg, Gemany, 2004. (to appear).

[BDRS02] M. Balser, C. Duelli, W. Reif, and G. Schellhorn. Verifying concurrent systems with symbolic execution. *Journal of Logic and Computation*, 12(4):549–560, 2002.

[BDW00] T. Bienmöller, W. Damm, and H. Wittke. The STATEMATE verification environment – making it real. In E. A. Emerson and A. P. Sistla, editors, *CAV'00: 12th international Conference on Computer Aided Verification*, number 1855 in LNCS, pages 561–567, Chicago, IL, USA, 2000. Springer-Verlag.

[BRS+00] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS, pages 363–366. Springer-Verlag, 2000.

[CMZ02] A. Cau, B. Moszkowski, and H. Zedan. *ITL – Interval Temporal Logic.* Software Technology Research Laboratory, SERCentre, De Montfort University, The Gateway, Leicester LE1 9BH, UK, 2002. www.cms.dmu.ac.uk/~cau/itlhomepage.

[DJHP98] W. Damm, B. Josko, H. Hungar, and A. Pnueli. A compositional real-time semantics of STATEMATE designs. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *COMPOS' 97*, volume 1536 of *LNCS*, pages 186–238. Springer-Verlag, 1998.

[FFB96] Betriebliches Lastenheft für FunkFahrBetrieb, 1996. Stand 1.10.1996.

[Gei99] R. Geisler. *Formal Semantics for the Integration of Statecharts and Z in a Metamodel-Based Framework.* PhD thesis, Technical University of Berlin, 1999.

[GHD98] W. Grieskamp, M. Heisel, and H. Dörr. Specifying embedded systems with statecharts and Z: An agenda for cyclic software components. In E. Asteaiano, editor, *FASE'98*, number 1382 in LNCS, pages 88–107. Springer-Verlag, 1998.

[Har84] D. Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 2, pages 496–604. Reidel, 1984.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4), 1990.

[HN96] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[HP98] D. Harel and M. Politi. *Modeling Reactive Systems With Statecharts: The Statemate Approach*. McGraw Hill, 1998.

[HRS88] M. Heisel, W. Reif, and W. Stephan. Program Verification Using Dynamic Logic. In E. Börger, H. Kleine Büning, and M. Richter, editors, *1st Workshop on Computer Science Logic. Proceedings*, Springer LNCS 329, 1988.

[KT02] J. Klose and A. Thums. The STATEMATE reference model of the reference case study 'Verkehrsleittechnik'. Technical Report 2002-01, Universität Augsburg, 2002.

[LMS97] Y. Lakhnech, E. Mikk, and M. Siegel. Hierarchical automata as model for statecharts. In *Asian Computing Science Conference (ASIAN'97)*, volume 1345 of *LNCS*. Springer-Verlag, 1997.

[Mos85] B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Computer*, 18(2):10–19, 1985.

[OT02] F. Ortmeier and A. Thums. Formale Methoden und Sicherheitsanalyse. Technical Report 15, Universitt Augsburg, 2002. (in German).

[OTSR04] F. Ortmeier, A. Thums, G. Schellhorn, and W. Reif. Combining formal methods and safety analysis – the ForMoSA approach. In H. Ehrig, editor, *Integration of Software Specification Techniques for Applications in Engineering*, volume (this volume) of *LNCS*. Springer-Verlag, 2004.

[RR00] G. Reggio and L. Repetto. Casl-Chart: A combination of statecharts and of the algebraic specification language Casl. Technical report, DISI Universitá di Genova, 2000.

[RST00a] W. Reif, G. Schellhorn, and A. Thums. Formale Sicherheitsanalyse einer funkbasierten Bahnübergangsteuerung. In E. Schnieder, editor, *Forms2000 – Formale Techniken für die Eisenbahnsicherung*, volume Reihe 12, Nr. 441 of *Fortschritt-Bericht VDI*, 2000.

[RST00b] W. Reif, G. Schellhorn, and A. Thums. Safety analysis of a radio-based crossing control system using formal methods. In E. Schnieder and U. Becker, editors, *9th IFAC Symposium Control in Transportation Systems 2000*, June 2000.

[SA91] V. Sperschneider and G. Antoniou. *Logic: A Foundation for Computer Science*. Addison Wesley, 1991.

[STR02] G. Schellhorn, A. Thums, and W. Reif. Formal fault tree semantics. In *Proceedings of The Sixth World Conference on Integrated Design & Process Technology*, Pasadena, CA, 2002.

[Thu04] A. Thums. *Formale Fehlerbaumanalyse*. PhD thesis, Universität Augsburg, Augsburg, Germany, 2004. (in German), (to appear).

[TO03] A. Thums and F. Ortmeier. Formal safety analysis in transportation control. In E. Schnieder, editor, *International Workshop on Software Specification of Safety Relevant Transportation Control Tasks*, volume 12 of *VDI Fortschritt-Berichte*. VDI Verlag GmbH, 2003.

[VGRH81] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook*. Washington, D.C., 1981. NUREG-0492.

[Wir90] M. Wirsing. *Algebraic Specification*, volume B of *Handbook of Theoretical Computer Science*, chapter 13, pages 675 – 788. Elsevier, Oxford, 1990.