Interactive Verification of Statecharts

Andreas Thums and Michael Balser
Lehrstuhl Softwaretechnik,
Universität Augsburg, 86135 Augsburg, Germany
{thums,balser}@informatik.uni-augsburg.de

Abstract

This paper presents an approach to the integration of statecharts, temporal logic and algebraic specification within an interactive verification environment. Currently some integrated formalisms exist [13, 7], but there is no proof support for theses approaches. Also model checkers are able to prove temporal properties of statecharts [3, 10], but they can only be used to verify properties based on a small, finite data domain.

Our goal is to provide a uniform, interactive proof support for verifying temporal properties of statecharts with algebraic data types and functions over infinite data domains. As an implementation platform the KIV system [2] is used. The semantics of statecharts is based on [6], which formalizes the STATEMATE semantics of statecharts [12].

1 Introduction

We present an approach which aims to support the interactive verification of (safety) properties for concurrent, reactive systems. For this, we use (i) statecharts to describe the operational system behavior, (ii) temporal logic to express properties of the complete execution trace, and (iii) algebraic specifications to formalize complex and possibly infinite data domains. Furthermore (iv) sequential programs are used as action language within stateqcharts.

We tightly integrate the different formalisms on the level of the semantics, interpreting statecharts as temporal formulas. Also, we provide a uniform proof method based on symbolic execution and induction. Symbolic execution is an intuitive proof method widely used for the interactive verification of sequential programs. We adapt this technique to the verification of temporal logic and statecharts.

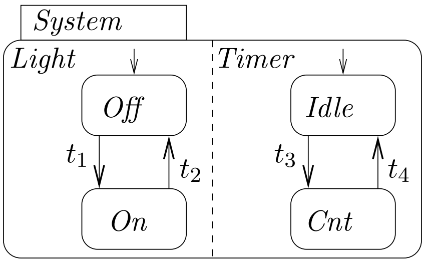
In this paper, we focus on statecharts, explaining how they can be interpreted as temporal formulas and how to symbolically execute statechart formulas. Details on executing temporal formulas can be found in [1]. Sequential programs are executed using Dynamic Logic (DL) [8]. Execution traces are linear sequences of states. Every state is interpreted over a many-sorted algebra allowing (generated) data types, functions and predicates. We assume the reader to be familiar with the statechart notation and sequent calculus.

To symbolically execute statecharts, every possible transition has to be considered. Because of parallel statecharts, more than one transition may be executed in one step and, if the statechart is indeterministic, all possible steps have to be considered. The execution of a step changes the configuration of a statechart. Such a step corresponds to a *micro-step* in STATEMATE. Time passes in a *macro-step* only, which occurs if no transition takes place, i.e. if the system is in a stable configuration. Then the system environment may react and create new input data.

Because the specification and verification environment KIV [2] offers strong proof support for algebraic specifications with higher order logic, verification of sequential programs with DL, and verification of temporal properties with interval temporal logic is currently implemented, this system was chosen as an implementation platform for the approach.

In the following, the toy example of an automatic light control will be used for explanation.

Example 1 (automatic light control) This light control automatically switches off the light five minutes after it has been switched on. Initially the light is *Off* and the timer is *Idle*. When the *pressed* event is enabled, the event *set* is generated (t_1) to start the timer and the light is switched on (t_3). While the timer is in state *Cnt*, x is incremented in every macro-step. Therefore we generate a *tick* event to characterize a macro-step. When x is greater than five, the timer leaves *Cnt* and generates a *sw_off* event (t_4), to switch off the light (t_2).



transitions:	
t_1:	$pressed/set$
t_2:	sw_off
t_3:	$set/x = 0$
t_4:	$x > 5/sw\_off$
static reactions:	
cnt :	$tick/x = x + 1$

Sect. 2 gives the necessary foundations of Intervall Temporal Logic (ITL) and DL and sketches an overall proof method. Sect. 3 describes, how the operational semantics of statecharts is integrated into the ITL framework. An example proof in Sect. 4 illustrates the approach and the paper concludes with Sect. 5.

2 The Temporal Logic Framework

The basis for our approach is Interval Temporal Logic (ITL) [11]. We use a first order extension of ITL which considers finite and infinite intervals as described in [5] (as a variation, we also consider primed variables and Dynamic Logic operators, see below). 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 $\sigma_i(x') = \sigma_{i+1}(x)$. All variables are flexible (also called dynamic), i.e. their values can be different in every state. Function and predicate symbols are rigid and are interpreted using algebras (see for example [14]).

As temporal operators we use $\varphi \frown \psi$ (chop), $\Box\ \varphi$ (always), $\Diamond\ \varphi$ (eventually), $\circ\ \varphi$ (strong next), $\bullet\ \varphi$ (weak next) and others with their standard semantics in ITL.

A particularity is the use of Dynamic Logic operators within the temporal logic framework. Dynamic Logic (DL) [8] can be used to describe complex relations between variables using (sequential) programs. In the following example we require variable x' to be equal to the value of variable x after a program has been executed.

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

Here, x' is either 1 or 2 depending on y. (Note that this "‘trick"’ of copying the value of variables after program execution to primed variables is also used in the following.) We are not restricted to conditionals and simple assignments. Especially we are able to use parallel assignments within programs

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

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

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

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

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

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

are used to execute assignments and conditionals. For a full set of rules for Dynamic Logic and their explanations, see for example [9].

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 $\varphi \frown \psi$ and others, see [1].

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* (which is similar to rule *next* in [4]).

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

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 in the first step in new variables v_0 and advanced one step by removing all primes and next operators. We can now continue unwinding the remaining temporal formulas until the induction hypothesis can be applied.

3 Embedding Statecharts in ITL

We support the STATEMATE semantics of statecharts [12] and our formalization strongly corresponds to the operational semantics presented in [6]. We also adopted the notations. A *configuration* of a statechart is represented as a valuation of *statechart variables*. In addition to the data variables X_{date} of STATEMATE, boolean variables for each state X_{st} (we use lower case letters for the variables to distinguish from the state names beginning with a capital letter) and each event X_{ev} are required, describing whether the corresponding state/event is active or not. The input variables $X_{inp} \subset X_{data} \cup X_{ev}$ are variables, which the systems environment may modify. A statechart SC describes a *transition system* where a statechart configuration is a *valuation* σ of the transition system and a *statechart step* is the corresponding transition relation. A trace of a statechart $trace(SC)$ is a sequence of valuations

$(\sigma_0, \sigma_1, \sigma_2, \ldots)$ where the relation (σ_i, σ_{i+1}) corresponds to the transition relation of SC. Statechart traces correspond to traces in ITL and we can interpret statecharts as temporal formulas as follows

$$(\sigma_0, \sigma_1, \ldots) \models SC :\Leftrightarrow (\sigma_0, \sigma_1, \ldots) \in \mathit{traces}(SC).$$

Thus, the integration of statecharts on the semantical level is straightforward.

Now we would also like to adopt the proof method of symbolic execution to statecharts. The following rule scheme describes how to unwind a statechart SC.

$$\frac{\Big(\ \mathit{cond}(\mathit{stp}_i), \mathit{exec}(\mathit{stp}_i), SC, \Gamma \vdash \Delta\ \Big)_{i=1\ldots n}}{SC, \Gamma \vdash \Delta}\ \mathit{sc\ unwind}$$

As statecharts can be indeterministic, we need to consider n premises, one for each possible step stp_i. The activation condition $\mathit{cond}(\mathit{stp}_i)$ introduces preconditions for executing a step. The formula SC describes the static transition system of the statechart whereas the dynamic behavior is encoded in the boolean variables for states and events, modified through executing $\mathit{exec}(\mathit{stp}_i)$.

Because we define events as boolean variables, they are – as conditions – first order formulas, and a transition t is, instead of a triple $e[\mathit{cnd}]/\mathit{act}$ in STATEMATE, a condition/action pair $t = (e \wedge \mathit{cnd}, \mathit{act})$, where act is a sequential program. The functions $\mathit{src}(t)$, $\mathit{trg}(t)$, $\mathit{en}(t)$, and $\mathit{act}(t)$ are computing the source state, the target state, the enabled condition $\mathit{src}(t) = \mathit{true} \wedge e \wedge \mathit{cnd}$, and the action act, of a transition, respectively. $\mathit{scope}(t)$ computes the state representing the scope of a transition, $\mathit{states}(SC)$ the (distinct) states of a statechart SC, $\mathit{mode}(\mathit{st}) \in \{AND, OR, BASIC\}$ the mode, and $\mathit{childs}(\mathit{st}) \in 2^{\mathit{states}(SC)}$ the substates of a state. For a detailed definition of these functions, we refer to [6].

3.1 The Initial Valuation

The function

$$\mathit{init}(SC) = \begin{array}{ll} x = \mathit{true}, & \text{if } x \in X_{st},\ x \text{ is an initial state} \\ x = \mathit{false}, & \text{if } x \in X_{st},\ x \text{ is not an initial state or } x \in X_{ev} \end{array}$$

computes the initialization for a statechart SC. The initial states are active and no event exists. To prove a property φ of a statechart SC, we have to prove $\mathit{init}(SC)$, $SC \vdash \varphi$ in the sequent calculus.

Example 2 The initialization for the Statechart Timer is

$$\mathit{init}(\mathit{Timer}) = \mathit{timer} \wedge \mathit{idle} \wedge \neg\ \mathit{cnt} \wedge \neg\ \mathit{set} \wedge \neg\ \mathit{sw\_off}$$

3.2 Computing Possible Statechart Steps

This section shows, how to compute the possible steps for the current configuration. The number of the indeterministic steps is reflected in the number of premises of the rule *sc unwind.* Basically we use the *step algorithm* from [6] which we adopte to treat conditions over symbolic values from algebraic specifications.

Therefore, in addition to the conflict free transition set T our step algorithm has to compute the condition *cnd* under which a step may take place and interactive proof steps decide, whether the condition holds. So, $steps(\sigma, st, cnd) \in 2^{(cnd,T)}$ is defined (our adoptions are highlighted) as

1. $mode(st) = \text{BASIC}$:

   $steps(\sigma, st, \text{cnd}) = \{(\text{cnd}, \emptyset)\}$, i.e. no transitions take place.

2. $mode(st) = \text{AND}$:

   $steps(\sigma, st, \text{cnd}) = \{\bigcup_i (cnd_i, T_i) | (cnd_i, T_i) \in steps(\sigma, st_i, \text{cnd})\}$,

   where $st_i \in childs(st)$, $(cnd, T) \cup (cnd', T') := (cnd \wedge cnd', T \cup T')$

3. $mode(st) = \text{OR}$:

   $$steps(\sigma, st, \text{cnd}) = \{(cnd_1, \{t_1\}), \ldots, (cnd_k, \{t_k\})\} \cup \\ steps(\sigma, st', cnd \wedge \neg\, en(t_1) \wedge \ldots \wedge \neg\, en(t_k))$$

   for $(cnd_i, \{t_i\}) \in \{(cnd_t, \{t\}) | scope(t) = st \text{ and } cnd_t := cnd \wedge en(t)\}$ and st' the unique active child of st in σ.

All possible steps for a statechart SC and a valuation σ are computed with $Steps(\sigma) = steps(\sigma, SC, true)$. The major difference between this algorithm and the original one is the third case. Because the enabled condition of a transition cannot be decided, every transition of the corresponding scope has to be considered and, under the condition that none of these transitions is enabled, also the transitions from the active subchart.

In addition to the transitions, for every state neither entered nor exited the corresponding *static reactions* have to be added to the action set. For the Ex. 1 the result for the first step is as follows:

Example 3 (conflict free transition sets)

$$\begin{array}{rcl} steps(\sigma, Light, true) & = & \{(pressed, \{t_1\}), (\neg\, pressed, \emptyset)\} \\ steps(\sigma, Timer, true) & = & \{(set, \{t_3\}), (\neg\, set, \emptyset)\} \\ steps(\sigma, System, true) & = & \{(press \wedge set, \{t_1, t_3\}), (press \wedge \neg\, set, \{t_1\}), \\ & & (\neg\, press \wedge set, \{t_3\}), (\neg\, press \wedge \neg\, set, \emptyset)\} \end{array}$$

3.3 A Statechart Step

Statecharts are unwound by executing the different steps computed by the *step* algorithm. Executing one step stp_i consists of four tasks. First, the events are reset with $prg_{reset}(stp_i)$, because they are only active for a single step. Thereafter the actions of the corresponding transitions are executed by $prg_{act}(stp_i)$ and the derived events – we consider entering (*enter*) and exiting (*exit*) of states – are generated with $prg_{set}(stp_i)$. Finally the active states for the next step are computed with $prg_{nxt}(stp_i)$.

Because the underlying logic already supports execution of DL programs (see Sect. 2), they are used for executing STATEMATE actions. Setting and resetting of events e is expressed by the assignment $e := true$ resp. $e := false$. The program $\langle prg_{reset}(stp_i); prg_{act}(stp_i); prg_{set}(stp_i); prg_{nxt}(stp_i)\rangle\ \varphi$ encodes a step execution for the step $stp_i = (cnd_i, T_i)$[1]. If the statechart has reached a stable configuration ($T = \emptyset$), no transitions take place. The systems environment assigns arbitrary values to the input variables and generates the *tick* event, characterizing a macro-step. This is reflected by the DL program $\langle prg_{reset}(stp); prg_{env}(stp)\rangle\ \varphi$. The DL programs are defined as

$$\begin{array}{rcl}
prg_{reset}(stp_i) & = & e_1 := false \mid \ldots \mid e_n := false, \text{ where } e_i \in X_{ev} \\
prg_{act}(stp_i) & = & act(t_1) \mid \ldots \mid act(t_m), \text{ where } T_i = \{t_1, \ldots, t_m\} \\
prg_{set}(stp_i) & = & y_1 := true \mid \ldots \mid y_l := true, \text{ where} \\
 & & y_i \in \{e \mid e = exit(src(t) \text{ or } e = enter(trg(t)) \text{ and } t \in T_i\} \\
prg_{nxt}(stp_i) & = & \mid_i st_i := true \mid_j st_j := false, \text{ where } st_{i,j} \in X_{st}, \\
 & & st_i \text{ active, } st_j \text{ not active in the next step} \\
prg_{env}(stp_i) & = & tick := true \mid z_1 :=? \mid \ldots \mid z_k :=?, \\
 & & \text{where } z_i \in X_{inp} \text{ and } z_i :=? \text{ assigns any value to } z_i \\
\varphi & = & \bigwedge_i x'_i = x_i, \text{ for every } x_i \in X.
\end{array}$$

The DL program computes the new valuation for the variables and the formula φ copies this valuation to the next configuration.

In addition to the transitions we get for every step a condition $cond(stp_i)$ under which this step may take place. This condition consists of the different activation conditions of the transitions and is added to the antecedent of the sequent. If this precondition is not fulfilled, the antecedent is contradictory and the proof for this case is finished (i.e. this step cannot take place). The whole step execution is implemented in the rule scheme *sc unwind* (see above) and will be exemplified in the following.

[1] Remember, that ; in DL is the sequential operator and **not** a parallel operator as in STATEMATE. DL supports the | operator for parallel assignments.

Example 4 (unwinding) Considering the (macro) step $stp = (\neg\ set, \emptyset)$ for the statechart $Timer$ (see Ex. 3), we get $cond(stp) = \neg\ set$,

$$\begin{aligned} prg_{reset}(stp) &= sw\_off := false \mid set := false, \\ prg_{act}(env) &= tick := true \mid set :=?, \\ prg_{nxt}(stp) &= timer := true \mid idle := true \mid cnt := false, \\ \varphi &= (sw\_off' = sw\_off\ \wedge\ set' = set \wedge x' = x \wedge \\ &\quad timer' = timer \wedge idle' = idle \wedge cnt' = cnt) \end{aligned}$$

4 An Example Proof

Symbolically executing statecharts will be explained by proving the property $\Box\ cnt \to x \leq 6$ for the subchart $Timer$ of the chart $System$ (see Ex. 1), i.e. $init(Timer), Timer \vdash \Box\ cnt \to x \leq 6$. This states that the lamp never lights up for more than 6 time units. Because the statechart $Timer$ is cyclic, induction over the length of the infinite trace is needed to prove that property.

First of all, the property has to be proven for the first step. Therefore the temporal operator has to be unwound with the rule *always right.*

$$\frac{\begin{array}{c} init(Timer), Timer \\ \vdash cnt \to x \leq 6 \end{array} \qquad \begin{array}{c} init(Timer), Timer \\ \vdash \bullet\ \Box\ cnt \to x \leq 6 \end{array}}{init(Timer), Timer \vdash \Box\ cnt \to x \leq 6} \quad ,$$

resulting in the properties for the first step on the left. This case is trivial, because cnt is initially false (see $init(Timer)$ in Ex. 2). The proof continues with the right case which states $cnt \to x \leq 6$ from the next step on. To prove this, the statechart has to be unwound by computing the next statechart steps $\{(set, t_1), (\neg\ set, \emptyset)\}$, and executing the transitions. Because two different steps are possible, the rule *sc unwind* branches into two cases:

$$\frac{\begin{array}{l} set, \\ \ldots \end{array} \qquad \begin{array}{c} \neg\ set, \\ exec((\neg\ set, \emptyset)), init(Timer), Timer \vdash \bullet\ \Box\ cnt \to x \leq 6 \end{array}}{init(Timer), Timer \vdash \bullet\ \Box\ cnt \to x \leq 6}$$

The left step – only sketched – is possible, if the event set is active, which contradicts the initialization $init(Timer)$, so this case is simple. The right case corresponds to a macro-step, because the transition set is empty. Therefore the $tick$ event is generated and the input variables get random values from the environment. The execution of the DL program (event reset and reaction of the environment, see Ex. 4) results in

$$\begin{gathered} \neg\ sw\_off, x = 0, tick, timer, idle, \neg\ cnt, sw\_off' = sw\_off, set' = set, \\ x' = x, tick' = tick, timer' = timer, idle' = idle, cnt' = cnt, \\ Timer \vdash \bullet\ \Box\ cnt \to x \leq 6 \end{gathered}$$

Now, every temporal formula talks about the next state, because it is preceded with a next operator and we can advance one temporal step (unnecessary variables are omitted here) resulting in

$$timer, idle, \neg\ cnt, \neg\ sw\_off, x = 0, Timer \vdash \Box\ cnt \to x \leq 6.$$

Now, it cannot be determined, if the environment has assigned *set* to *true* or *false*. When *set* is *false*, the statechart is in the same configuration as after initialization and this case can be proven with the trace induction argument. For the second case, additional steps, following the presented proof scheme, are necessary. We will omit them here and note, that the strategy of unwinding tl operators, unwinding statecharts, executing actions, and advancing in the trace until the induction hypothesis completes the proof can be automated. The example is proven fully automatic in the KIV system.

5 Conclusion

In this paper we presented an integration of STATEMATE statecharts into the framework of ITL. Symbolic execution is used for proving temporal properties of statecharts. By using DL programs, the action language of STATEMATE statecharts was for free and even enriched by the possibility of using arbitrary algebraic data types and functions.

Example proofs have shown, that symbolic execution is an intuitive way to prove properties, because the proof follows the execution paths of the statechart. Also this proof strategy may easily be mapped to heuristics to automate proofs. In addition, the proof strategy of symbolic execution allows simple integration into the existing ITL framework. Beside the implementation of the step algorithm, only the two proof rules *sc init* and *sc unwind* were needed.

Currently we are working on a generalization of the presented approach, where it is not necessary to fully specify, whether a state is active or not. This approach is helpful for generalizing induction hypothesis. If, e.g. the property $\Box\, cnt \to x \leq 6$ has to be proven for the statechart *System*, it is not necessary to know, if the subchart *Light* is in state *Off* or *On*, so we may abstract from this state. Examples have shown, that this approach shortens proofs.

References

[1] M. Balser, C. Duelli, W. Reif, and G. Schellhorn. Verifying concurrent systems with symbolic execution. *Journal of Logic and Computation (Special Issue)*, 2002. (to appear).

[2] 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*, LNCS 1783. Springer, 2000.

[3] T. Bienmöller, W. Damm, and H. Wittke. The STATEMATE verification environment – making it real. In E. A. Emerson and A. P. Sistla,

editors, *12th international Conference on Computer Aided Verification, CAV*, LNCS 1855. Springer, 2000.

[4] S. Biundo, D. Dengler, and J. Koehler. Deductive planning and plan reuse in a command language environment. In *Proceedings of the 10th European Conference on Artificial Intelligence*, 1992.

[5] 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. www.cms.dmu.ac.uk/ cau/itlhomepage.

[6] 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*, LNCS 1536. Springer, 1998.

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

[8] D. Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 2. Reidel, 1984.

[9] 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.

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

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

[12] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In *Symposium on Theoretical Aspects of Computer Software*, 1991.

[13] 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.

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