Implementing Statecharts in PROMELA/SPIN

Erich Mikk, Yassine Lakhnech, Michael Siegel
Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Preußerstr. 1-9, D24105 Kiel, Germany
{erm,yl,mis}@informatik.uni-kiel.de

Gerard J. Holzmann
Bell Laboratories, MH 2C-521
600 Mountain Avenue
Murray Hill, New Jersey 07974
U.S.A.
gerard@research.bell-labs.com

ABSTRACT

We translate statecharts into PROMELA, the input language of the SPIN verification system, using extended hierarchical automata as an intermediate format We discuss two possible frameworks for this translation, leading to either sequential or parallel code. We show that in this context the sequential code can be verified more efficiently than the parallel code. We conclude with the discussion of an application of the resulting translator to a well-known case study, which demonstrates the feasibility of linear temporal logic model checking of statecharts.

Keywords
Statecharts, SPIN, hierarchical automata, automatic translation, model checking

1 INTRODUCTION

STATEMATE from iLogix [17] is a CASE-tool for the specification, analysis, design, and documentation of complex reactive systems. More than 2000 licenses are in use worldwide for this tool, with almost equal parity in the fields of aerospace, communications, electronics, and transportation. The graphical language of statecharts as proposed by David Harel [8, 9] serves in STATEMATE for the description of control, data transformation, and timing aspects of the system under development. STATEMATE supports among others specification simulation, static and dynamic checks, and code generation (e.g, into VHDL, or Ada).

The language of statecharts incorporates several abstractions that qualifys it as a specification language for reactive systems. This includes:

- *Uninterpreted events* which are means of communication between the environment and the specification (or between the parts of the specification). The communication mechanism is a synchronous broadcast of events.
- Statecharts is related to the *synchrony hypothesis* ([3]) which states that the controlling hardware/software is infinitely faster than the environment it controls.

Such abstractions make statecharts an attractive candidate for exhaustive automatic analysis with traditional model checking techniques that cope with finite state systems.

Providing STATEMATE with state-of-the-art technology for model-checking is in the focus of this paper. The aim is to describe how statecharts can be translated into PROMELA/SPIN to allow for linear temporal logic model checking of statecharts. We have implemented a translator based on this idea and report on first experiments with its application.

The current paper is based on previous work as follows. Harel and Naamad gave a description of statecharts as implemented in STATEMATE in [9]. The rigorous, but still informal, description of [9] was formalized in [23]. An operational semantics of statecharts based on [9, 23] is given in [24]. The latter operational semantics forms the basis for the translation discussed here. The basic idea of the translation is explained fully in the current paper.

Related work in the area of model checking statecharts is too extensive to be discussed here. Probably the most closely related work is that of Brockmeyer and Wittich [4] because they investigate the same version of statecharts. They report on translating almost the whole language of statecharts as supported by STATEMATE to the input language of a symbolic model checker [7]. However, their paper is not detailed enough on topics that are under focus here: operational semantics of statecharts and a description of compilation schema based on this semantics.

The structure of the paper is as follows. In Section 2 we explain the intermediate steps that are performed to obtain a format that is simple enough to describe the

TO APPEAR IN THE PROCEEDINGS OF WIFT'98 WORKSHOP. THIS IS A PRE-RELEASE OF THE PAPER TO BE DISTRIBUTED ON THE WORKSHOP.

translation of statecharts into Promela. In Section 3 we describe the operational semantics of extended hierarchical automata. Section 4 discusses the basic schema of the translation and Section 5 describes optimizations of the basic schema. Section 6 discusses a case study: the verification of a production cell specification.

2 INTERMEDIATE FORMAT FOR STATECHARTS SEMANTICS

The graphical statecharts language extends the standard state transition diagram notation with parallelism, hierarchy, and broadcast communication. Statecharts comprise powerful concepts, such as interlevel transitions, multiple-source/multiple-target transitions, transition priority, and simultaneous execution of maximal non-conflicting sets of transitions. Add-on tools, which can be linked with the Statemate tool, have to deal with the rather involved semantics of these concepts.

In order to simplify the understanding of statecharts, it is reasonable to restrict ourselves to a sub-language. A transition in statecharts can be labeled an event expression, a condition and an action expression (all optional). The first two form the guard of the transition, the last describes actions that are performed when the transition is taken. In the sub-set we consider, transition labels are, restricted as follows:

- only boolean combinations of predicates *in(st)* are allowed in expression *Cond*; informally, predicate *in(st)* is true iff the system currently resides in state *st*;
- the only effect of taking a transition is the generation of events.

We do not consider data transformations, history, and timing issues. These aspects are orthogonal to the investigations of this paper in the sense that they do not interfere with basic concepts presented in the following sections. This sub-set is supported by our first translator.

The actual syntax of transitions is rather complicated in Statemate. Instead, Harel&Naamad [9] introduce an intermediate format, where transitions are transformed into so called *full compound transitions* (full CT's). The purpose of full CT's is to collect labeling information that is spread over transition segments and to compute in advance states to be entered when a transition is taken.

When translating statecharts to Promela we would like to be able to verify the correctness of the translation or even verify the compiler correctness. In order to do so, we need a structure-directed operational semantics. Unfortunately, full CT's language is not well-suited for giving such semantics definition because it contains so-called *interlevel transitions*, which are an important part of the language (cf. [8]). Interlevel transitions (possibly with multiple sources/multiple targets) are transitions which do not respect the hierarchy of states, i.e. which cross several hierarchy levels. They can be understood as a powerful goto mechanism which allows one to arbitrarily move control across the state hierarchy. The price one has to pay for interlevel transitions is their intricate semantics, especially when combined with the statecharts priority mechanism. Interlevel transitions spoil the clean decomposition of a system into subsystems (since "dangling" transitions without source or target can result) and thus they destroy a clean structural operational semantics definition for statecharts.

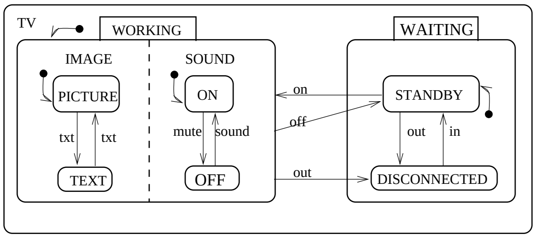
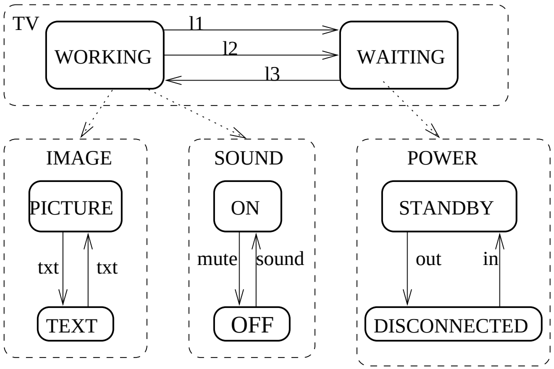


Figure 1: A statechart with inter-level transitions.

Example 1 Figure 1 depicts a model of a TV-set as statecharts with interlevel transitions, e.g. from state *STANDBY* to state *WORKING*. This example is inspired by [16].

To overcome problems with interlevel transitions we proposed *extended hierarchical automata (EHA)* as an intermediate format in [24]. The EHA formalism uses single-source/single-target transitions as in usual automata definitions (without interlevel transitions) and has a simple priority concept which facilitates computing the next step of an EHA when compared to statecharts. The transformation from statecharts to EHA as described in [24] is basically lifting interlevel transitions to the uppermost states that are exited and entered, resp., when a transition is taken. In order to faithfully model the semantics of the original statechart, the transitions in EHA have extended labels that allow to constrain the enabledness of the statecharts transitions, and to determine the target-configuration of the transition.

Example 2 Figure 2 depicts an extended hierarchical automaton that corresponds to the statecharts in Figure 1. Now interlevel transitions are replaced by transitions labeled with $l1$, $l2$ and $l3$. The first component of such a label (*source restriction*) is used to restrict the enabledness of the transition. The last component (*target determinator*)is used to determine the states to be entered in the sub-automata of the transition target. So the transition labeled with $l3$ is enabled if *WAITING* and *STANDBY* are active and the event *on* is present; upon taking the transition, the states *WORKING*, *PICTURE* and *ON* are entered. This exactly corresponds to taking the transition labeled with *on* in Figure 1. A more detailed explanation of this figure follows in the sequel.



where labels $l1$ to $l3$ are described by the following table (these transitions do not generate events, so the action part is omitted here):

Label	Source restriction	Guard	Target determinator
$l1$	$\emptyset$	*off*	$\{STANDBY\}$
$l2$	$\emptyset$	*out*	$\{DISCONNECTED\}$
$l3$	$\{STANDBY\}$	*on*	$\{PICTURE, ON\}$

Figure 2: TV-set as extended hierarchical automata.

An empty source restriction denotes that the source is not constrained. In the current paper we describe how to translate EHA into PROMELA. A translator from statecharts to EHA is described in [24] and implemented as part of our tool-set.

3 EXTENDED HIERARCHICAL AUTOMATA AND THEIR OPERATIONAL SEMANTICS

Extended hierarchical automata consist of sequential automata and their hierarchy and parallel composition. In the following we introduce the notation for EHA and explain its semantics.

Sequential automata. An EHA consists of a set of sequential automata. *Sequential automaton* A is a 4-tuple (Σ, s_0, L, δ) where Σ is the set of states, $s_0 \in \Sigma$ is the initial state of A, L is the set of transition labels (we will be more specific about labels later), and $\delta \subseteq \Sigma \times L \times \Sigma$ is the transition relation.

In Figure 2 sequential automata are surrounded with dashed lines. The following sequential automata are used in our example:

- Automaton *TV* with states *WORKING* and *WAITING*, (TV-set can be on or off);
- Automaton *IMAGE* with states *PICTURE* and *TEXT*, (there are two images: the usual image and video-text);
- Automaton *SOUND* with states *ON* and *OFF*, (sound may be either on or off);
- Automaton *POWER* with states *STANDBY* and *DISCONNECTED*, (power supply may be on or off).

The interaction of the components of an EHA is defined by a state hierarchy and a broadcast communication mechanism.

State hierarchy. Hierarchy and parallel composition of extended hierarchical automata is defined using a single *composition function.* Consider a set of sequential automata $F = \{A_1, \ldots, A_n\}$ with mutually disjoint state spaces. Composition function $\gamma : \bigcup_{A \in F} \Sigma_A \rightarrow \mathbb{P}(F)$ maps a state s of a sequential automaton to a set of automata $G \subseteq F$. We say that s is *refined by* G. We require that a composition function is a tree-like function, with a designated root automaton γ_{root} (details in [24]).

If $\mid \gamma(s) \mid = 1$ then s is refined into a single automaton, if $\mid \gamma(s) \mid > 1$ then s is refined into a parallel composition of automata. Otherwise, $\gamma(s) = \emptyset$, and we call s a *basic state* (denoted by $Basic_{\gamma}(s)$).

In Figure 2 dotted arrows are used to depict the composition function. State *WORKING* is refined to a parallel composition of automata *IMAGE* and *SOUND*, and the state *WAITING* is refined into a sequential automaton *POWER*. Hence, the composition function is

$$\begin{array}{l} \gamma = \{ \mathit{WAITING} \mapsto \{\mathit{POWER}\}, \\ \quad \mathit{WORKING} \mapsto \{\mathit{SOUND}, \mathit{IMAGE}\}\} \\ \quad \cup \{s \mapsto \emptyset \mid s \in \{\mathit{PICTURE}, \\ \qquad\qquad \mathit{TEXT}, \\ \qquad\qquad \mathit{ON}, \mathit{OFF}, \\ \qquad\qquad \mathit{STANDBY}, \\ \qquad\qquad \mathit{DISCONNECTED}\}\}. \end{array}$$

Events and broadcast communication The communication media in the EHA as in statecharts is instantaneous broadcast of uninterpreted events (events don't have structure, they are either present or not). Events can be received from the environment or be generated as result of taking a transition in the EHA. In the latter case the events generated are also visible to the environment. We denote the set of events of the EHA by E. There are the following events in our TV-set model $E = \{txt, mute, sound, in, out, on, off\}$.

Configuration. As usual in automata, we speak of the states of sequential automata as being *control states*. While sequential automata form an EHA, the control state of an EHA, which we call a *configuration*, is a composition of control states of its component automata. Every sequential automaton can contribute at most one state to a configuration. Due to hierarchy, a configuration is upward and downward closed: whenever a state is in a configuration and it is a non-basic state, each of its direct sub-automata must contribute to the configuration too, and vice versa. We denote the set of all configurations $Conf(\gamma)$.

Note, that in case of parallel composition, each of the sub-automata contributes to the configuration, if their parent is in that configuration. In our example this means that whenever the TV-set is in state *ON*, also *IMAGE* and *SOUND* are active, i.e. one state of the respective automata belong to the current configuration.

Initial configuration. Initially, the hierarchical automaton is in a configuration derived from the initial states of the set of sequential automata. This derivation is performed in a top-down-manner: the root automaton contributes to the initial configuration by its initial state; if some state in the configuration is refined to further automata, then these automata must contribute their initial states to the initial configuration as well.

In our example the initial configuration is {*WAITING*, *STANDBY*}, which corresponds to the plugged-in TV-set in stand-by mode. This initial state is not depicted in the Figure 2.

Extended transitions. Transitions are depicted as labeled arrows between two boxes. In contrast to statecharts, a transition in an EHA always resides within one sequential automaton (thus transitions respect hierarchy). Taking an enabled transition means to leave its source state and enter its target state. But if a source and/or target of a transition is refined to further automata (they have sub-structure) then taking a transition affects this sub-structure as well: by leaving the source state, its sub-structure is left as well and while entering the target state some configuration below its target is entered as well. Next we will define these issues more precisely.

The label of a transition between source state s and a target state s' is a 4-tuple (sr, ex, ac, td), where

- *source restriction* $sr \in Conf(\gamma \mid_s)$ allows to constrain the enabledness of the transition to a set of sub-configurations below s,
- expression ex, the *transition guard*, is a proposition over event and state names (constant *true*, primitive events and/or states connected with $\wedge, \vee, \neg$). Models of ex are tuples (C, E), where C is a configuration and E is a set of events.,
- and $ac \subseteq \mathsf{E}$ is a set of generated events.
- *target determinator* $td \in Conf(\gamma \mid_{s'})$ is used to determine which states are entered simultaneously when entering the target state s'.

Given a tuple (C, E), where $\mathsf{C} \in Conf(\gamma)$ and $\mathsf{E} \subseteq E$. A transition $t = (s, (sr, ex, ac, td), s')$ of an automaton $A \in F$ is *enabled* in the status (C, E) (denoted by $enabled_{(\mathsf{C},\mathsf{E})}(t)$) iff the source state is an active state i.e. $s \in \mathsf{C}$, the source restriction is an active subconfiguration $sr \subseteq \mathsf{C}$ and transition guard ex is enabled $(\mathsf{C}, \mathsf{E}) \models ex$.

Semantics of EHA. Given an extended hierarchical automaton $EHA = (F, E, \gamma)$, where F is a set of sequential automata, E is a set of events and γ is the composition function. The semantics of *EHA* is a Kripke structure $\mathbf{K} = (\mathbf{S}, \mathbf{s_0}, \xrightarrow{STEP})$ where

- $\mathbf{S} = Conf(\gamma) \times \mathbb{P}(E)$ is the set of states of $\mathbf{K}$, elements of $\mathbf{S}$ are also called *status*;
- $\mathbf{s_0} \in \mathbf{S}$, where $\mathbf{s_0} = (C_0, \emptyset)$ is the initial state of $\mathbf{K}$. Initial configuration C_0 is derived from the initial states of sequential automata as described above.
- $\xrightarrow{STEP} \subseteq \mathbf{S} \times \mathbf{S}$ is the transition relation of $\mathbf{K}$, where $(\mathsf{C}, \mathsf{E}) \xrightarrow{STEP} (\mathsf{C}', \mathsf{E}')$ will be defined in the sequel.

The step relation. A maximal non-conflicting set of enabled transitions constitutes a transition in $\mathbf{K}$; we refer to it as a *step* of EHA. Non-conflicting means that it respects non-determinism (in a sequential automaton from several enabled transitions at most one can be taken) and priority (transitions emerging from a state with a sub-structure preempt all transitions in the substructure). Our idea is to define $\xrightarrow{STEP}$ inductively such that components of hierarchical automata can fire transitions and the results of components are composed into the step relation.

We distinguish between extended hierarchical automata 1) interacting with the environment by receiving events from the environment and suppling generated events to the environment (open system approach) and 2) not interacting with the environment (closed system approach).

CLOSED SYSTEMS:

$$\frac{\gamma_{root} :: (\mathsf{C}, \mathsf{E}) \rightarrow (\mathsf{C}', \mathsf{E}')}{(\mathsf{C}, \mathsf{E}) \xrightarrow{STEP} (\mathsf{C}', \mathsf{E}')}$$

OPEN SYSTEMS:

$$\frac{\gamma_{root} :: (\mathsf{C},\mathsf{E}) \to (\mathsf{C}',\mathsf{E}') \wedge \mathsf{E}' \subseteq \mathsf{E}''}{(\mathsf{C},\mathsf{E}) \xrightarrow{STEP} (\mathsf{C}',\mathsf{E}'')}$$

The latter rule allows the environment to insert arbitrary sets of events after each step. Note, that during the step (while several sequential automata fire their transitions) events from the environment do not have any impact (they are postponed until the end of the step).

These rules establish relation $\xrightarrow{STEP}$ based on relation $\to$; the latter is defined by inference rules given below. We refer to these rules as the *start rules*. An inference in our rule system is denoted by $A :: (\mathsf{C},\mathsf{E}) \to (\mathsf{C}',\mathsf{E}')$, where $A \in F$ is a sequential automaton; (C,E) is a status of the hierarchical automaton EHA, where $\mathsf{C}' \in Conf(\gamma \mid_{\Sigma_A})$ (i.e. a configuration of an EHA which has A as root automaton) and $\mathsf{E}' \subseteq E$.

Progress rule applies to a sequential automaton A if one of its states s is in the configuration C and if one of the outgoing transitions is enabled; then one of them is taken non-deterministically. The effect of the transition is determined by the transition label: the target state s' and the target determinator states td are entered, and events of ac are generated. This is specified by the following inference rule:

$$\frac{\begin{array}{l} \{s\} = \mathsf{C} \cap \Sigma_A \\ \exists\, tr \in \delta_A . enabled_{(\mathsf{C},\mathsf{E})}(tr) \wedge tr = (s,(sr,ex,ac,td),s') \end{array}}{A :: (\mathsf{C},\mathsf{E}) \to (\{s'\} \cup td, ac)}$$

Example 3 Consider the status (C,E), where

$$\begin{array}{l} \mathsf{C} = \{WAITING, STANDBY\} \\ \mathsf{E} = \{on, out\}. \end{array}$$

Then the progress rule applies to the root automaton TV and transition from *WAITING* to *WORKING*. Hence we obtain a derivation in our inference system:

$$\begin{array}{l} TV :: (\{WAITING, STANDBY\}, \{on, out\}) \to \\ \quad (\{WORKING, PICTURE, ON\}, \emptyset) \end{array}$$

But similarly, the progress rule applies to automaton *POWER* as well:

$$\begin{array}{l} POWER :: (\{WAITING, STANDBY\}, \{on, out\}) \to \\ \quad (\{DISCONNECTED\}, \emptyset) \end{array}$$

Obviously, these two transitions can not be taken in the same step because their effects are inconsistent. According to the priority concept in extended hierarchical automata (and statecharts), only the first transition contributes to the current step. This is already modeled by the rules presented so far: the derivation of the automaton TV satisfies the premise of the $\xrightarrow{STEP}$ rule which constitutes a complete derivation (and hence constitutes a step). Later on we will see that there is no derivation in our set of rules, that would include the transition of *POWER* as well.

Composition rule. This rule explains how an automaton delegates its step to its sub-automata. This rule applies to an automaton A that has one of its states s in the configuration C but all outgoing transitions are disabled, and the state s is refined to a set of automata $\{A_1, \ldots, A_m\}$. Then the relation $\to$ for A collects the results of $\to$ performed by its sub-automata $\{A_1, \ldots, A_m\}$. This is specified by the following inference rule:

$$\frac{\begin{array}{l} \{s\} = \mathsf{C} \cap \Sigma_A \\ \forall\, tr \in \delta_A . tr = (s,l,s') \Rightarrow \neg\, enabled_{(\mathsf{C},\mathsf{E})}(tr) \\ \gamma(s) = \{A_1, \ldots, A_m\} \neq \emptyset \\ A_1 :: (\mathsf{C},\mathsf{E}) \to (\mathsf{C}_1',\mathsf{E}_1') \\ \ldots \\ A_m :: (\mathsf{C},\mathsf{E}) \to (\mathsf{C}_m',\mathsf{E}_m') \end{array}}{A :: (\mathsf{C},\mathsf{E}) \to (\{s\} \cup \mathsf{C}_1' \cup \ldots \cup \mathsf{C}_m', \mathsf{E}_1' \cup \ldots \cup \mathsf{E}_m')}$$

Note, that if the active state is refined into a parallel composition then the components are executed simultaneously and they do not interfere during the step.

Example 4 Consider the status (C,E), where

$$\begin{array}{l} \mathsf{C} = \{WORKING, PICTURE, ON\} \\ \mathsf{E} = \{txt, mute\}. \end{array}$$

Then the root automaton TV doesn't have any enabled transitions, hence its sub-structures *IMAGE* and *SOUND* may execute a transition. For both of them, the progress rule applies resulting in the following derivations:

$$\begin{array}{l} IMAGE :: (\{WORKING, PICTURE, ON\}, \{txt, mute\}) \\ \qquad\qquad\qquad \to \\ \qquad\quad (\{TEXT\}, \emptyset) \\ SOUND :: (\{WORKING, PICTURE, ON\}, \{txt, mute\}) \\ \qquad\qquad\qquad \to \\ \qquad\quad (\{OFF\}, \emptyset) \end{array}$$

Hence we obtain a derivation for the automaton TV:

$$\begin{array}{l} TV :: (\{WORKING, PICTURE, ON\}, \{txt, mute\}) \\ \qquad\qquad\qquad \to \\ \qquad\quad (\{WORKING, TEXT, OFF\}, \emptyset) \end{array}$$

which satisfies the premise of the $\xrightarrow{STEP}$ rule, hence this derivation constitutes the step.

Stuttering rule applies to a sequential automata A that has one of its states s in the configuration C, but none of the outgoing transition guards is enabled and

s does not have further sub-structure. The effect of this rule is to remain in state s without generating any events. This is specified by the following inference rule:

$$\frac{\begin{array}{l} \{s\} = \mathsf{C} \cap \Sigma_A \\ Basic_\gamma(s) \\ \forall\, tr \in \delta_A . tr = (s, l, s') \Rightarrow \neg\, enabled_{(\mathsf{C},\mathsf{E})}(tr) \end{array}}{A :: (\mathsf{C}, \mathsf{E}) \to (\{s\}, \emptyset)}$$

Example 5 Consider the status (C, E), where

$$\begin{array}{l} \mathsf{C} = \{\mathit{WORKING}, \mathit{TEXT}, \mathit{OFF}\}, \\ \mathsf{E} = \{\mathit{txt}\}. \end{array}$$

As in the previous example the composition rule applies and both *IMAGE* and *SOUND* are requested to execute a transition. In *IMAGE* the progress rule applies, but in *SOUND* the stutter rule applies. The derivation with the stutter rule is:

$$\begin{array}{c} \mathit{SOUND} :: (\{\mathit{WORKING}, \mathit{TEXT}, \mathit{OFF}\}, \{\mathit{txt}\}) \\ \to \\ (\{\mathit{OFF}\}, \emptyset). \end{array}$$

The composition rule yields:

$$\begin{array}{c} \mathit{TV} :: (\{\mathit{WORKING}, \mathit{TEXT}, \mathit{OFF}\}, \{\mathit{txt}\}) \\ \to \\ (\{\mathit{WORKING}, \mathit{PICTURE}, \mathit{OFF}\}, \emptyset) \end{array}$$

which constitutes the step.

Note, that a non-determinism of an automaton in an EHA induces a non-determinism in the step relation of the EHA. This is immediately evident from the existential quantification in the *progress rule*.

The interplay between rules. The progress and stuttering rules may produce derivations that are not collected by the composition rule. As in example 3 consider the following status (C, E), where $\mathsf{C} = \{\mathit{WAITING}, \mathit{STANDBY}\}$ and $\mathsf{E} = \{on, out\}$. The premises of the progress rule are satisfied such that the automaton *POWER* can switch from state *STANDBY* to *DISCONNECTED*. However this transition is discarded by the composition rule (and there is no other rule that would collect this derivation of *POWER* into the set of possible steps from the above configuration).

These superfluous inferences can be avoided by the following traversing strategy through the state hierarchy: beginning in the root automaton we move downwards in the composition function if the current active automaton does not have an enabled transition (according to composition rule); the downward movement ends if the progress or the stutter rule applies. An application of the progress rule stops further inferences in sub-components.

4 TRANSLATION OF EXTENDED HIERARCHICAL AUTOMATA INTO Promela: BASIC SCHEMA

Our target system is the software package Spin for formal verification of distributed systems. Besides a protocol simulator and a so-called bit-state analyzer, Spin includes a linear temporal logic (LTL) model checker which is our object of interest. Programs are written in Promela (PROcess MEta LAnguage). Properties can be expressed as LTL formulae or as Omega automata. We restrict our description of Promela to the parts that are relevant to our translation. Promela is inspired by Dijkstra's guarded command language notation and Hoare's language CSP and offers a wide range of abstract constructs that can be used for the description of reactive systems:

- Parallel processes with an interleaving model of parallelism,
- Global and local data variables of finite types (types bit, byte, short and int),
- Statements to express assignment and conditions, combined statements for sequential execution, conditionals, loops.
- Synchronization of parallel processes on common channels: synchronous message passing.

We have a distinguished `atomic` statement, which glues several statements into an atomic one, thus preventing interleaved execution with statements of other parallel processes. This list is far from being complete, for more detailed info we refer to language manuals accessible via Web [26], the book [10] and the recent overview [12].

The model-checker of Spin uses explicit state enumeration, the reduction strategy of Spin is partial order reduction ([13]).

Given an extended hierarchical automaton $\mathit{EHA} = (F, E, \gamma)$ and its semantics $\mathbf{K} = (\mathbf{S}, \mathbf{s}_0, \xrightarrow{STEP})$. The translation maps *EHA* to a Promela model P. This model P contains variables necessary to encode states of $\mathbf{K}$ and a Promela process(es) that encodes the transition relation $\xrightarrow{STEP}$. The operational semantics as described in Section 3 is crucial to a structural translation: rules of our semantics can now be used as generic pattern while generating code for transitions, sequential automata, as well as parallel and hierarchy compositions of extended hierarchical automata. That is, the translation of a construct in the extended hierarchical automaton is an instantiation of the corresponding rule(s). This will be worked out in the sequel.

Implementation of status

A status has structure: every status of $\mathbf{K}$ is a tuple (C, E), where C is a configuration of EHA: $\mathsf{C} \in Conf(EHA)$, and E is a set of events: $\mathsf{E} \subseteq E$. Recall that a configuration C is a set of states of EHA: $\mathsf{C} \subseteq \bigcup_{A \in F} \Sigma_A$. To model configurations we define for every sequential automata with n states a variable that distinguishes at least $n+1$ different values; n values to model the states of the sequential automata, and the extra state to model that the automaton does not have an active state in the current configuration. To model events we define one boolean variable for each event of E. Hence the potential data space allocated by these variables indeed suffices to encode $Conf(\gamma) \times E$.

To implement concurrent access to those variables within the PROMELA processes that encode the transition relation $\overset{STEP}{\longrightarrow}$ we need pre and post state copies of state and event variables (because there is no synchronous parallel composition in PROMELA). Therefore we define two sets of variables: a variable with a suffix 0 belongs to the pre-state and a variable with a suffix 1 belongs to the post-state.

However this implementation still suffers from a theoretical blemish: the state vector is twice as long as expected because it consists of pre and post state variables (we would expect pre state variables only). But we will see in the following, that the post state variables are treated quite economically by the translator and SPIN such that this does not lead to the doubling of the state space.

The initial status

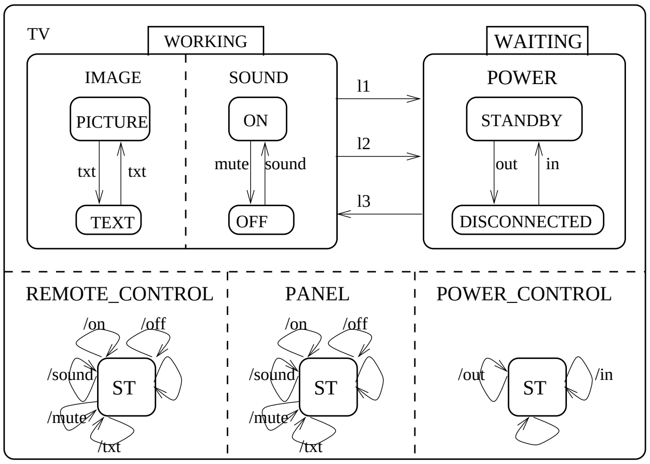
The initial status of $\mathbf{K}$ is (C, E), where C is the initial configuration $\mathsf{C} \in Conf(\gamma)$ and an empty set of events $\mathsf{E} = \emptyset$. According to our encoding we initialize pre-state variables of those automata that have states in C to the respective states and the rest of the automata to their non-active state. All post-state variables of automata are initialized to non-active states.

All event variables (pre and post state) are initialized to false. The idea is that state changes within one step of the Kripke structure $\mathbf{K}$ are computed into post-state variables. Since the initial status of post-state variables is set to non-active or 0, resp., the computation of the current step has to set variables only, there is no need to reset any variables.

Modeling the transition relation

Support for closed systems only. The specification to be checked should be a closed system, i.e. the specification is supposed to comprise also an abstract model of its environment.

Example 6 In Figure 3 we complemented the TV-set from Figure 2 by a remote control, manual service panel and power supply control. (This time we depict hierarchy in statecharts-style where composition function corresponds to state embedding.) The idea is that those components generate randomly events (or stutter) which are sensed by the TV control system. Remote control and panel coincide, the events *in* and *out* can be generated by the power control component only.



	Source restriction	Guard	Target determinator
$l1$	$\emptyset$	off	$\{STANDBY\}$
$l2$	$\emptyset$	out	$\{DISCONNECTED\}$
$l3$	$\{STANDBY\}$	on	$\{PICTURE, ON\}$

Figure 3: TV-set as closed system.

Implementation of parallel composition. An intriguing problem is now how to implement the parallel composition of extended hierarchical automata. For a set of sequential automata $\{A_1, \ldots A_n\} \subseteq F$ such that for some state $s \in \Sigma_A$, where $A \in F$ we have $\gamma(s) = \{A_1, \ldots A_n\}$, these automata are parallel. Assume that this set is considered actually by the composition rule. We make two general observations:

- The sequential automata A_i $(i \in 1 \mathinner{..} n)$ contribute to the transition relation $\overset{STEP}{\longrightarrow}$ simultaneously,
- There is no communication during simultaneous execution: events generated by transitions are sensed in the next step only (cf. [9]).

The question is now how to translate the stated pattern of parallel execution into PROMELA, which uses an interleaving model of parallelism. There are two choices:

Parallel solution. Transitions, formerly located in parallel automata, are now executed in an interleaved fashion: parallel automata are mapped to PROMELA processes. This solution requires a scheduler to implement the "lock-step" nature of the transition relation.

Sequential solution. Arbitrary interleaving of parallel transitions is determined at compile-time yielding a fully sequential implementation. This solution is sound because these transitions are independent according to the composition rule.

The drawback of implementing the synchronous parallel composition with interleaving is the introduction of non-determinism. This is illustrated with the following example.

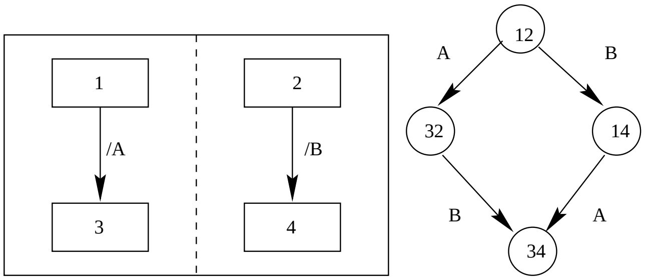


Figure 4: Implementing synchronous parallel composition by interleaving.

Example 7 Consider the statecharts depicted in Figure 4 on the left. Being in $\{1, 2\}$ this statecharts generates simultaneously events A and B. On the right, generation of these events in depicted using an interleaving model of parallelism: there are two orders in which events can be generated, which introduces intermediate states not present in the statecharts semantics.

Here, different interleavings within the step relation lead to the same result. Hence we can expect that the partial order reduction techniques of SPIN are able to reduce the number of possible interleavings to one. Since it is known which PROMELA patterns can be exploited by the reduction algorithm, ([13]) encourages the protocol designer to choose the structure of the model appropriate.

On the other hand the sequential implementation guarantees that exactly one interleaving is chosen. Since the sequential solution is at least as efficient as any possible parallel one, we proceed with describing the sequential solution only. This observation coincides with the compilation scheme of other synchronous languages.

As a consequence of the sequential solution partial order reduction techniques implemented in SPIN do not benefit our code. While partial order techniques cope with state explosion caused by interleaving, since there is no interleaving in statecharts, there is also no need for this reduction.

Flow of control between sequential automata. As described in Section 3 the operational semantics induces a traversing strategy of the composition function in order to avoid superfluous derivations in the inference system. We can imagine this as a depth-first search in the composition function, where at the deepest level we have an application of the progress or stuttering rule. This will be worked out in the sequel.

According to the rules the control between sequential automata is passed in the following manner: first the root automaton is considered to take a transition (according to the start rule). If it does not have an enabled outgoing transition in the active state then the composition rule applies and the sub-automata of the active state are scheduled (given the active state is a non-basic state, otherwise the stuttering rule applies and the step is complete). As described in the composition rule, the result of the rule depends on the result of its subcomponents. If there are several sub-automata, which corresponds to their parallel composition, we pass control to each of them in arbitrary order.

Advantages of this schema are: (1) hierarchical structure is mapped to if-clauses of PROMELA, thus the translation faithfully implements the hierarchy of statecharts using if-clauses; (2) redundant checks are avoided: first higher-level transitions are checked for enabledness and control is passed to lower transitions only if the higher ones are not enabled in the current step.

Implementation of sequential automata. The implementation of sequential automata either executes one of its transitions (progress rule), depends on the execution of its subcomponents (composition rule) or stutters (stutter rule). All those cases are implemented as conditions over the pre state variables and assignments to the post state variables. If the flow of control is passed to a sequential automaton it detects which is its active state – the formulae "let $\{st\} \in \mathsf{C} \cap \Sigma_A$" is common to all those rules. For a sequential automaton A with states `ON` and `OFF` this results in the following code:

```
if
:: (A0=ON) ->
   /* code associated with
      outgoing transitions of ON */
:: (A0=OFF) ->
   /* code associated with
      outgoing transitions of OFF */
fi;
```

(recall that the suffix 0 denotes that it is the pre state copy of A).

Now we discuss code associated with outgoing transitions of some state. To implement non-deterministic exclusive choice between outgoing transitions of s we again use if-clause. So in the TV-example (as in Figure

3) for the state ON we obtain

```
if :: out0 ->
   /* code implementing the effect
      of taking this transition */
   :: off0 ->
   /* code implementing the effect
      of taking this transition */
   :: else ->
   /* if none of the transitions are taken */
fi;
```

The effect of taking a transition is described by the progress rule: let
$tr = (s, (sr, ex, ac, td), s')$ be the transition chosen by the premise of the rule. Then the conclusion of the rule i.e. $(\{s'\} \cup td, ac)$ requires to set in the post-state variables encoding $(\{s'\} \cup td, ac)$ to true; and the post-state of variables encoding ac to true as well.

Complementary to the code of outgoing transitions, there must be code implementing the situation that none of the transitions is enabled. This code comprises the else-option of the above if-clause. There are two cases to consider:

- the state s is a basic state; then the post-state is $(\{s\}, \emptyset)$ according to the stuttering rule; to achieve this we set the variable implementing the post-state of s to true.
- the state s is a non-basic state; the control is passed to sub-automata $\{A_1, \ldots, A_m\}$ according to the composition rule. The contributions of the transitions in A_i to the overall transitions is computed sequentially. Recall that the composition rule results in the following status:

  $$(\{s\} \cup \mathsf{C}'_1 \cup \ldots \cup \mathsf{C}'_\mathsf{m}, \mathsf{E}'_1 \cup \ldots \cup \mathsf{E}'_\mathsf{m})$$

  where $(\mathsf{C}'_\mathsf{i}, \mathsf{E}'_\mathsf{i})$ is the result of sub-automata A_i. Since sub-automata independently set the corresponding post-state variables $(\mathsf{C}'_\mathsf{i}, \mathsf{E}'_\mathsf{i})$ to true, all that remains to do, after control is passed back from sub-automata, is to set the variable implementing the post-state of s to true.

The implementation of the start rule. We described already how sequential automata are implemented. The computation of $\overset{STEP}{\longrightarrow}$ is initiated by the start rule, which invokes the depth-first search along the composition function. By enclosing the code for the start rule and the code for sequential automata and their compositions into one `atomic` statement we hide all intermediate states which result from the successive computations of the overall effect of the transitions. The resulting PROMELA program just displays transitions $(\mathsf{C}, \mathsf{E}) \overset{STEP}{\longrightarrow} (\mathsf{C}', \mathsf{E}')$ which corresponds to steps in the original EHA.

Implementation of infinite runs
Up to now we have described how to encode the transition relation. In order to generate infinite runs of the PROMELA model we have to wrap the code into an infinite loop. The loop always begins with the actual status in the pre state variables and the post state variables set to zero. The loop consist of three consecutive parts:

1. The post state of the transition relation is accumulatively computed into the post variables as described in the previous section.
2. The resulting post state is copied to the pre state for the calculation of the next step.
3. Post state variables are reset: variables implementing control states of automata are set to the respective non-active state and event variables are set to zero (as discussed in the initialization section).

Those three parts are enclosed into an `atomic` statement in order to hide intermediate steps. The atomic statement is broken once per loop execution: at the beginning of the loop. Here the property in the never claim has access to the variables which comprises an observable state of the system. In such a manner, the number of states generated by the verifier in reachability analysis corresponds exactly to the number of states in the original statecharts model.

This means that the properties about the model can be formulated without actually knowing the implementation details. Also, the verification results (like counterexamples) can be returned to the user in terms of the original specification.

5 OPTIMIZATIONS OF BASIC SCHEMA

Whereas the basic schema is important to understand the correctness of the translation, is must be optimized to get better performance. The translator generates optimized code which leads to decrease of memory and time requirements compared to the basic schema.

Post copies of state variables only if needed. Sometimes we can reduce the number of post state variables. Write access to state variables is exclusive within one step: only one statement is accessing a state variable. But since transition guards may refer to arbitrary states within the whole statecharts, we may have read-write conflicts. The idea is to analyze the statecharts for situations where read-write conflicts may arise and then generate post state copies only in cases where it is needed to avoid the conflict.

The translator analyzes transition guards and generates post state copies only for variables that occur as conditions in transition guards.

On variable ordering. As described in the section 4 the post-state variables have zero values in control locations visible to the property (i.e. outside the atomic statement). The graph encoding technique ([14]) can take an advantage from this particular situation if those variables are grouped together. Those variables are declared as the last ones which assures that they comprise the suffix part of the state vector.

On marking of deterministic sequences. PROMELA provides a `d_step` sequence which is executed as if it were one single statement, and is therefore more efficient to use during verifications than an atomic sequence. The verification can gain in time and space if deterministic flows are marked so. The translator includes the following statements into `d_step`:

- As described in "Implementation of sequential automata", once a transition is scheduled to be taken or a stutter step is executed, the code performs a number of assignments that correspond to setting post state variables of control states and events, which is deterministic. Such a sequence is included into `d_step`.
- We can detect for deterministic sequential automata. One simple heuristic is that if each state of a sequential automata has only one outgoing transition then the automaton is deterministic. This check is implemented in the translator. A straightforward extension of this is if several deterministic automata are executed in a sequence, e.g., several deterministic automata are in parallel in the original statecharts (however, currently it is not supported by the translator).

6 PRODUCTION CELL CASE STUDY

The production cell case study [20] was launched by Forschungszentrum Informatik in Karlsruhe, Germany, to demonstrate benefits of formal methods for industrial applications. Up to now 36 contributions including 2 with statecharts are reported ([25]). We used the specification due to University of Oldenburg and OFFIS [20, 18]. Since the statechart specification is contributed by a third-party, we restrict the discussion to the verification figures only (very detailed explanation of this specification can be found in [20, 18]).

The statecharts specification contains 30 parallel automata, 30 events are used to synchronize components of the model. The number of variables needed to encode states and events in PROMELA is 137 (60 boolean and 77 enumeration type variables). The reachable state space is $3.08191e+06$ states as counted by SPIN version 3.2.3. The experiment is done on an Sparc Ultra 1 of 1GB memory. Here are the statistics of the state space analysis.

• Default search:	266MB , 1:37:21h,
• Graph encoding technique [14]:	111MB , 2:34:08h

These figures show that there is a trade-off between the default search and graph encoding technique: whereas the default search is fast and consumes more memory, the graph encoding technique is slower but reduces the memory requirements considerably. An experiment with the recursive indexing method (option `-DCOLLAPSE` of SPIN version 2.9.5 upwards) [11] turned out to be worse in time and space than the default search (313MB memory, 00:42:55h), which means that we could not benefit from this reduction technique. The reason is that all data is global in our models, but recursive indexing exploits the fact that data is associated with processes (like local variables and communication channels).

Unfortunately, the verification figures are incomparable with those of other contributions ([20]) because of different modeling approaches. The closest modeling is from P. Kelb [18], but he uses a different semantics of statecharts. We doubt that a comparison of his figures with ours would be meaningful.

We used SPIN for the verification of safety properties. However, our approach is not limited to safety properties only. Since we translate to PROMELA, the full range of the SPIN verification system is available (linear temporal logic model checker, bit-state hashing algorithm, hash-compact algorithm and other features).

Besides verification, we could successfully use the counterexample generation facility of SPIN for detecting errors in the statecharts specification. Our reference specification was already verified (and we did not find any errors there), but during manual entering this specification to the STATEMATE tool, we made some errors. Those errors were not detected by simulating the specification in STATEMATE but by model checking the result of our translation with SPIN. We could replay counterexamples in STATEMATE in interactive simulation mode which was very helpful for debugging.

7 INDUSTRIAL RELEVANCE AND FUTURE WORK

The commercial tool STATEMATE offers among others interactive simulation of statecharts and code generation from statecharts. We tried to verify safety properties with the dynamic test tools of STATEMATE (a reachability analysis tool, which is no longer part of the distribution). The case study was infeasible for this tool: we didn't manage to get any results within a runtime of less than one day. Furthermore, reachability analysis covers safety properties only. These limits are now extended by our contribution that allows for verification of safety and liveness properties with one of the most efficient linear temporal logic model checkers available.

This paper does not cover tools that make the use of our translator comfortable, but we made these tools and the translator available to the public ([1]). These tools include a *model extractor* that allows to use STATEMATE as graphical user interface for drawing statecharts. We also implemented a SPIN *back-end* that allows to translate counterexamples back into terms of the original statecharts specification. This facility allows to automatically replay SPIN-generated counterexamples in STATEMATE. With this back-end, the translation described in this paper is hidden from the user such that the user does not need to learn PROMELA in order to use SPIN for model checking statecharts. For those who use other graphical editors for drawing statecharts, we have documented our *abstract syntax,* such that the translator can be used separately from STATEMATE, too. In this case either ASCII or C-code level interface is available.

We admit, that the sub-set of statecharts discussed in this paper is not sufficient for many industrial specifications written in STATEMATE. However, conceptionally, STATEMATE statecharts can be model checked with traditional finite state model checking techniques. The full language of statecharts (apart from generic charts) supports nothing but finite state systems (even integer and real variables are restricted to sub-ranges in STATEMATE). We didn't try out, but we are confident that other "interesting" features of statecharts like timing primitives, data transformations, and history concept can be handled with PROMELA/ SPIN as well. We plan to extend the sub-language in the future work.

But even when the translator supported the full sub-language of statecharts, there is still no guarantee that the specification can be verified with SPIN. If the state space turns out to be infeasible for SPIN then it offers a supertrace/bit-state algorithm (bit-state analyzer) for a high-coverage approximation of the full state space. In this heuristic, one state is collapsed to a couple of bits thus allowing for the search of considerably bigger state spaces. SPIN also has an implementation of hash-compact algorithm [27] that gives a statistical guarantee of full coverage.

Further (general) solutions that alleviate the state explosion problem are, e.g.,

Abstraction techniques that can either manual or automatic. The idea is to find a smaller model that exposes the same behavior w.r.t. the property to be verified. Localization technique proposed by R. Kurshan ([19]) is one examples in automatic abstraction. P. Kelb [18] investigated abstraction techniques for statecharts (but he uses another dialect). Related work in this area is also [21].

Compositionality is the divide-and-conquer technique for verification. The literature is this area is far too extensive to be discussed here. In order to apply compositionality for verification of statecharts, one has to investigate the compositional aspects of the semantics as this is done in [6].

Symbolic techniques of model checking might deliver an alternative insight into the problem of model checking statecharts. Symbolic techniques are successful in hardware verification where OBDD representations of the state space leads to good performance results (both in time and space). Considering software, wide ranges of the variables involved might spoil the performance of OBDD-based techniques. However, first encouraging results in model checking software specifications also exists (e.g. [2, 5]).

We plan to investigate abstraction techniques and employ a symbolic model checker (e.g. SMV [22]) for statecharts.

8 CONCLUSION

We have described how statecharts can be translated into PROMELA and mentioned two possible options: the generation of sequential or parallel code. The translation allows for generation of code which size is linear in the size of the input statechart. Hence the translator passes the parallel structure of statecharts faithfully to the model-checker SPIN, which is designed to cope with the state explosion problem limiting the practical model-checking. We motivated that generating sequential code is more efficient than parallel code. Further we verified some safety properties of the production cell case study. Space and time requirements of this verification showed that the translator and SPIN can be used for the verification of systems of this size, and possibly larger.

Our translator and other support tools are available to the public for experiments ([1]).

ACKNOWLEDGMENTS

We thank Peter Kelb for making the specification of production cell available and Bernd Weber for adapting it to STATEMATE. The translator was written using the VDM Domain Compiler [15] and we thank Uwe Schmidt and Hans-Martin Hörcher for providing this tool. Erich Mikk thanks Bell Labs, where he worked on this topic in the summer of 1997.

REFERENCES

[1] MOCES: MOdel ChEcking Statecharts. `http://www.informatik.uni-kiel.de/~erm/MOCES/`

[2] R. J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 21_6 of *ACM Software Engineering Notes*, pages 156–166, New York, October16–18 1996. ACM Press.

[3] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[4] U. Brockmeyer and G. Wittich. Tamagotchis Need Not Die – Verification of STATEMATE Designs. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'98)*, number 1384 in LNCS, pages 217–231. Springer Verlag, 1998.

[5] W. Chan, R. J. Anderson, P. Beame, and D. Notkin. Improving efficiency of symbolic model checking for state-based system requirements. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA-98)*, volume 23,2 of *ACM Software Engineering Notes*, pages 102–112, New York, March2–5 1998. ACM Press.

[6] W. Damm, H. Hungar, B. Josko, and A. Pnueli. A Compositional Real-Rime Semantics of STATEMATE Designs. In *Proceedings of COMPOS'97*, To appear. Springer Verlag, 1998.

[7] T. Filkorn and SIEMENS AG. Applications of Formal Verification in Industrial Automation and Telecommunication. In *Proceedings of Workshop on Formal Design of Safety Critical Embedded Systems*, 1997.

[8] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

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

[10] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[11] G.J. Holzmann. State Compression in Spin. *Proc. Third Spin Workshop*, April 1997.

[12] G.J. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[13] G.J. Holzmann and D. Peled. An improvement in formal verification. In *Proc. FORTE94*, Berne, Switzerland, October 1994.

[14] G.J. Holzmann and A. Puri. A Minimized Automaton Representation of Reachable States. Manuscript, 1997.

[15] H-M. Hörcher and U. Schmidt. Programming with VDM Domains. In D. Bjørner, H. Langmaack, and C.A.R. Hoare, editors, *VDM'90 – VDM and Z*, number 428 in Lecture Notes in Computer Science, pages 122–134. VDM Europe, Springer Verlag, April 1990.

[16] C. Huizing and W.-P. de Roever. Introduction to design choices in the semantics of Statecharts. *Information Processing Letters*, 37:205–213, February 1991.

[17] iLogix Web Site. `http://www.ilogix.com/company/company.htm`

[18] P. Kelb. *Abstraktionstechniken für Automatische Verifikationsmethoden.* PhD thesis, Universität Oldenburg, 1995.

[19] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton Univ. Press, 1994.

[20] C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: Case Study Production Cell.* Number 891 in LNCS. Springer Verlag, 1995.

[21] J. Lind-Nielsen, H. R. Andersen, G. Behrmann, H. Hulgaard, K. Kristoffersen, and K. G. Larsen. Verification of Large State/Event Systems using Compositionality and Dependency Analysis. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'98)*, number 1384 in LNCS. Springer Verlag, 1998.

[22] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publisher, 1993.

[23] E. Mikk, Y. Lakhnech, C. Petersohn, and M. Siegel. On formal semantics of Statecharts as supported by STATEMATE. In *2nd BCS-FACS Northern Formal Methods Workshop.* Springer-Verlag, July 97.

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

[25] Production Cell Web Site. `http://www.fzi.de/prost/projects/production_cell/ProductionCell.h`

[26] SPIN Web Site. `http://netlib.bell-labs.com/netlib/spin/whatispin.html`

[27] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Proc. 5th Int. Conference on Computer Aided Verification*, LNCS. Springer Verlag, 1993.