Verifying Statecharts with State Invariants

Emil Sekerinski
McMaster University
Hamilton, Ontario, Canada
emil@mcmaster.ca

Abstract

Statecharts are an executable visual language for specifying the reactive behavior of systems. We propose to statically verify the design expressed by a statechart by allowing individual states to be annotated with invariants and checking the consistency of the invariants with the transitions. We present an algorithm that uses the locality of state invariants for generating "many small" verification conditions that should be more amenable to automatic checking than an approach based on a single global invariant.

1 Introduction

Graphical notations are commonly used in the analysis and design of complex systems for communication and for documentation. Statecharts are such a "visual language" for specifying the reactive behavior of systems [11]. Statecharts address the state explosion problem of simple state transition diagrams when modeling complex systems with parallel threads of control by allowing *hierarchy*, *concurrency*, and *communication*. The appeal of statecharts is that on one hand, the intuitive notation allows requirements to be "easily expressed" and communicated to "domain experts" and that on the other hand, statecharts can be directly executed through interpretation or compilation. Hence statecharts are "executable specifications".

In this paper, we propose a method for statically verifying the design expressed by a statechart, as a way to increase the confidence in the design that complements validation through testing, see Fig. 1. The approach is to supplement a statechart with a number of *state invariants*; we call the resulting diagrams *invariantcharts*. State invariants are conditions attached to individual states and specify what has to hold in a state configuration, e.g. as in the TV control in Fig. 2: The activity is partitioned into two modes, *Standby* and *Working*. When in *Working* state, the system is in both *Picture* and *Sound* substates. Within *Picture* the system is in one of the substates *WarmingUp* and *Displaying*,

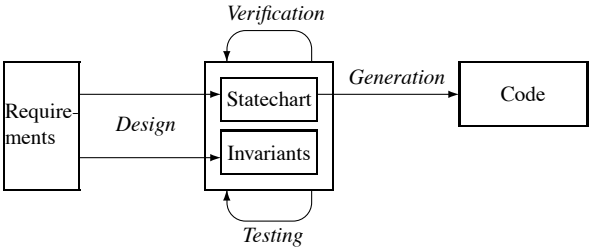

Figure 1. Development Process.

within *Sound* the system is in one of the substates *Waiting*, *On*, and *Off*. The invariant of *Working* is that whenever *Sound* is not in *Waiting*, i.e. is in *On* or *Off*, *Picture* must be in *Displaying*. The invariant of *Sound* states that the sound level *lev* must be between 1 and 10. The event *power* causes the system to flip between *Standby* and *Working*, no matter in which substates of *Working* the system is. The transition on event *warm* broadcasts event *soundOn*. The transition on events *down* can only be taken if $lev > 1$ and when taken, will decrement *lev*. The transition on *power* to *Working* sets *Picture* and *Sound* to the default initial states *WarmingUp* and *Waiting* and sets *lev* to 5.

State invariants are not meant for execution but allow the consistency to be checked in a way that goes beyond structural well-formedness: if a state S has invariant I attached to it, then every incoming transition must ensure that I holds in S. Dually, every outgoing transition can assume that I holds initially. Invariants can be attached to basic and composed states. Intuitively, state invariants document the "purpose" of states.

Invariants state safety properties of a control system or consistency properties of a software system and are an essential means to checking a design. Compared to writing an equivalent combined invariant as a single global predicate, the approach of state invariants decomposes a potentially large invariant into parts that are in visual proximity to affected transitions, making complex invariants more comprehensible. The approach also leads to shorter invariant expressions as *state tests* are implicit to the state to which

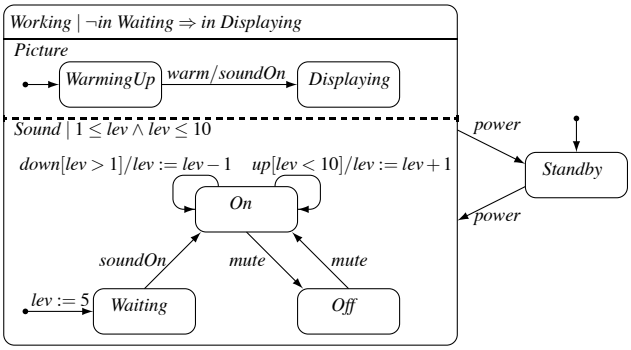

Figure 2. Invariantchart for a TV set.

an invariant is attached. Since the invariant of a composite state is "inherited" by all its children, this gives a natural way of factoring out commonalities, that in a purely textual representation would either lead to a (more difficult to read) parenthesized predicate or to repetition if "multiplied out".

We present an approach to generate "local" verification conditions for checking the consistency of state invariants with the transitions. Rather than generating a single "large" verification condition per event, we generate many smaller ones. Automated theorem provers are more effective at dealing with many small verification conditions than with few large ones. Thus the promise is that state invariants make it easier to specify correctness conditions for statecharts and make it easier to verify them. We present the algorithms for generating the verification conditions, as they are implemented in the *iState* tool.

The approach is inspired by the *nested invariant diagrams* of [2], in which inner states inherit the invariant of outer states, though the purpose is different. Nested invariant diagrams describe sequential code, including loops; transitions define conditional execution and are taken based only on their guards. Here, transitions require an (external) event and their execution cannot lead to looping. In [10] inheritance of annotations of states in hierarchical state machines (without concurrent states) is also discussed.

Our goal is have a simple calculus for the correctness of statecharts rather than to be inclusive of statechart features; our believe is that this simplicity will contribute to the reliability of designs. The presented theory can be considered as a Hoare calculus for statecharts.

Formal verification of statecharts by theorem proving and model checking has been studied repeatedly, e.g. in [6, 8, 13, 14, 15, 16], see [5] for a survey on model-checking approaches. These approaches allow invariants to be expressed as global properties only, rather than being attached to states, although they allow more general temporal properties than invariance.

Our statecharts resemble those originally proposed in [11], though the difference in the meaning of a "step" is subtle but substantial when compared to e.g. [7, 12, 17]. If a transition "generates" further events that cause other transitions, then the original meaning is that these lead to a sequence of internal micro-steps. Our interpretation is that all transitions on generated events are taken simultaneously as one atomic operation together with the initiating transition. This requires restrictions on statecharts that rule out many cases in which statechart interpretations differ [4]. These restrictions allow an event-centric semantics of statecharts with events as operations rather than data as in the more common state-centric semantics [19, 20]. The restriction to atomic "chains" of transitions avoids the question of whether state invariants have to hold in intermediate states through which micro-steps may go but which are not observable from the outside.

Our statecharts differ from UML statecharts, as formalized e.g. in [3, 9], by not considering queuing of events, not considering asynchronous messages, and hence not considering run-to-completion. In our model, caller and callee proceed simultaneously and free of interference.

The interpretation of a chain of transitions as an atomic operation makes use of the parallel composition of statements. Its use for that purpose was first suggested in [18], inspired by the use of parallel composition in the B Method [1].

The next section presents a model of program statements that includes the parallel composition of statements. Section 3 gives the formal definition of the structure of statecharts. Section 4 continues with the formal definition of the meaning of statecharts in terms of program statements. Section 5 defines state invariants and the correctness of invariantcharts, and gives an algorithm for generating verification conditions. We conclude with some observations in Section 6.

2 Preliminaries

We use generalized program statements to define the effect of an event. Program statements can also appear as the body of a transition. We assume that the body of a transition may read variables (program variables or sensors), may write to variables (program variables or actuators), and may do so depending on some conditions, but does not contain loops or recursion. For analyzing a certain class of reactive systems abstractly, the effect of an event can be taken to be "fast" and be sensed "instantaneously" by the environment that generated that event, hence this is a plausible restriction on statements. To simplify matters, we also assume that the evaluation of expressions is always defined. These assumptions allow us to give a simple relational semantics to statements.

For a statement P, we let $[P]$ be its "meaning" as a predicate relating the initial and final states, referred to by un-

primed and primed variables, and let αP be the list of variables that are assigned to by P. Both are defined over the structure of statements. Let g be a Boolean expression, xv a list of variables without duplicates, and ev a list of expressions of the same length as xv:

P	$[P]$	αP	side-condition
skip	true	$\emptyset$	
stop	false	$\emptyset$	
$xv := ev$	$xv' = ev$	xv	
$g \to Q$	$g \wedge [Q]$	αQ	
$Q \,[]\, R$	$[Q] \wedge xv' = xv \vee$	$\alpha Q \cup \alpha R$	$xv = \alpha R - \alpha Q$
	$[R] \wedge yv' = yv$		$yv = \alpha Q - \alpha R$
$Q \parallel R$	$[Q] \wedge [R]$	$\alpha Q \cup \alpha R$	$\alpha Q \cap \alpha R = \emptyset$

The parallel (or rather independent) composition $Q \parallel R$ is well-defined only if the variables assigned to in Q and R are disjoint. However, Q and R may read the variables assigned by the other; in that case, their initial value is read. The parallel composition is executed in one atomic step, without any interleaving. We write $Q = R$ if $[Q] = [R]$ and $\alpha Q = \alpha R$. Parallel composition is a generalization of multiple assignment, in the sense that $(x, y := e, f) = (x := e \parallel y := f)$. The nondeterministic choice $Q \,[]\, R$ selects either operand that is enabled; if both are enabled, their choice is arbitrary, if neither is enabled, $Q \,[]\, R$ blocks. The guarded statement $g \to Q$ is interpreted as blocking if g does not hold, otherwise Q is executed. It is used to define the conditional statement:

$$\begin{array}{lcl} \mathsf{if}\, g\, \mathsf{then}\, Q & \widehat{=} & (g \to Q) \,[]\, (\neg g \to \mathsf{skip}) \\ \mathsf{if}\, g\, \mathsf{then}\, Q\, \mathsf{else}\, R & \widehat{=} & (g \to Q) \,[]\, (\neg g \to R) \end{array}$$

The enabledness domain $\mathsf{en}\, P$ is the domain of the relation of statement P, i.e. $\mathsf{en}\, P = \exists xv' \cdot [P]$ where $xv = \alpha P$. For example, $\mathsf{en}\, \mathsf{skip} = \mathsf{true}$ and $\mathsf{en}\, \mathsf{stop} = \mathsf{false}$. That is, both statements don't change any variables, but skip terminates and stop blocks. The prioritizing composition $P /\!/ Q$ executes P, if P is enabled, otherwise it executes Q:

$$P /\!/ Q \quad \widehat{=} \quad P \,[]\, \neg\, \mathsf{en}\, P \to Q$$

The nondeterministic choice and parallel composition are associative and commutative, hence can be generalized to a choice over a finite number of alternatives, written as $[]\, i \in s \cdot P$ and $\parallel i \in s \cdot P$, where s is a finite set. The correctness assertion $\{p\}\, Q\, \{r\}$ states that under precondition p statement Q terminates with postcondition r:

$$\{p\}\, Q\, \{r\} \quad \widehat{=} \quad \forall xv' \cdot p \wedge [Q] \Rightarrow r[xv \backslash xv'] \text{ where } xv = \alpha Q$$

The common verification rules for statements hold, for example:

$$\begin{array}{lcl} \{p\}\, xv := ev\, \{r\} & \equiv & p \Rightarrow r[xv \backslash ev] \\ \{p\}\, g \to Q\, \{r\} & \equiv & \{p \wedge g\}\, Q\, \{r\} \\ \{p\}\, Q \,[]\, R\, \{r\} & \equiv & \{p\}\, Q\, \{r\} \;\wedge\; \{p\}\, R\, \{r\} \\ \{p\}\; Q /\!/ R\; \{r\} & \equiv & \{p\}\; Q\; \{r\} \;\wedge\; \{p \wedge \neg\, \mathsf{en}\, Q\}\; R\; \{r\} \end{array}$$

3 Statechart Structure

A statechart $\mathscr{S}$ is a structure (*Basic*, *AND*, *XOR*, *Root*, *parent*, *Event*, *Transition*, *default*), with a number of constraints on the components that shall be visited in turn. The finite sets *Basic*, *AND*, *XOR* are mutually disjoint sets of states. We let $Composite = AND \cup XOR$ be the set of composite states and $State = Basic \cup Composite$ be the set of all states. Among the XOR states is a distinguished root state, $Root \in XOR$.

The partial function (or functional relation) $parent : State \nrightarrow State$ maps every element of *State* except *Root* to a composite state, $\mathsf{dom}\, parent = State - \{Root\}$ and $\mathsf{ran}\, parent = Composite$. All states form a tree that is rooted in *Root*, formally $Root \in parent^*[\{s\}]$ for any $s \in State$, where r^* is the transitive and reflexive closure of relation r and $r[S]$ is the image of the set S under r. We let the relation *children* be the inverse of *parent*, i.e. $children = parent^{-1}$. The children of an AND state are said to be concurrent, the children of an XOR state are said to be exclusive.

The finite set *Event* is that of event names. The elements of the finite set *Transition* are tuples t, written as $t = ss \xrightarrow{E[g]/b} ts$, where $ss = source(t) \subseteq State$ is the set of source states, $ts = target(t) \subseteq State$ is the set of target states, $E = event(t) \in Event$ is the transition event, $guard(t) = g$ is a Boolean expression, the transition guard, and $body(t) = b$ is a statement, the transition body. The state *Root* must not be the source or target of any transition. All transitions must have at least one source state and one target state, i.e. $source(t) \neq \{\}$ and $target(t) \neq \{\}$ for any $t \in Transition$.

The partial function $default : XOR \nrightarrow Transition$ maps XOR states to default transitions. The source of a default transition of an XOR state s, if defined, is s itself, i.e. $source(default(s)) = \{s\}$. A fat dot inside the source state is used to visualize the source of a default transition in charts. A default transition must be defined for the root state and any XOR state that is the target of some transition (default or regular) or that is being implicitly entered as it has an AND ancestor that is being entered; this is being made precise shortly. The default transition of a state s, if defined, must go to a descendant of s, i.e. $target(default(s)) \subseteq children^+[\{s\}]$, where r^+ is the transitive closure of relation r.

Expressions are composed of program variables, the state tests $\mathsf{in}\, S_1, \ldots, S_m$, where S_i is any state except *Root*, and functions *fn* applied to zero or more arguments (functions with zero arguments being constants). We assume that the functions include common Boolean, arithmetic, and relational operators.

$$Ex \quad ::= \quad v \;\mid\; \mathsf{in}\, S_1, \ldots, S_m \;\mid\; fn(Ex_1, \ldots, Ex_n)$$

Statements are the skip statement, the multiple assignment, the broadcast E, with $E \in Event$, the parallel composition,

and the conditional:

$$St \quad ::= \quad \mathsf{skip} \mid v_1,\ldots,v_m := Ex_1,\ldots,Ex_m \mid E \mid St \parallel St \mid \mathsf{if}\, Ex\, \mathsf{then}\, St\, [\mathsf{else}\, St]$$

In charts, we allow transition guards and bodies to be left out. If a transition guard is missing, it is assumed to be true. If a transition body is missing, it is assumed to be skip. The event of a default transition does not play any role and is always left out.

The closest common ancestor $cca(ss)$ of a set ss of states is the state that is an ancestor of each state in ss and all other common ancestors are also its ancestor. We write $x\,r\,y$ for the pair (x,y) belonging to relation r.

$$c = cca(ss) \quad \Leftrightarrow \quad (\bigwedge s \in ss \cdot s\,parent^+\,c) \wedge (\bigwedge a \in State \cdot (\bigwedge s \in ss \cdot s\,parent^+\,a) \Rightarrow c\,parent^*\,a))$$

The closest common ancestor exists and is unique for any non-empty set of states that does not include the root state. States r, s are orthogonal, written $r \perp s$, if their closest common ancestor is an AND state and neither is an ancestor of the other. For example, *Displaying* and *On* are orthogonal, but *Displaying* and *Standby* are not and *Displaying* and *Working* are not. A set ss of states is called orthogonal, written $\perp ss$, if every pair of distinct states of ss is orthogonal. For any transition, both its source and target states must be orthogonal, $\perp source(t)$ and $\perp target(t)$ for all $t \in Transition$. The scope of a transition is the state closest to the root through which the transition passes.

$$scope(t) \quad \widehat{=} \quad cca(source(t) \cup target(t))$$

We define what it means that E leads to t, in symbols $E \rightsquigarrow t$ and that E broadcasts F, also written as $E \rightsquigarrow F$: (1) $E \rightsquigarrow t$ for all t with $E = event(t)$, (2) $E \rightsquigarrow F$ if $E \rightsquigarrow t$ and $body(t)$ broadcasts F, (3) $E \rightsquigarrow t$ if $E \rightsquigarrow F$ and $F \rightsquigarrow t$, for some F, and (4) $E \rightsquigarrow F$ if $E \rightsquigarrow G$ and $G \rightsquigarrow F$, for some G.

The final set of constraints on statecharts is on broadcasting. Any event can be broadcast only once. More precisely, for any event E, for distinct t,u such that $E \rightsquigarrow t$ and $E \rightsquigarrow u$, $body(t)$ and $body(u)$ must not contain broadcasts of the same event. Broadcasting can only trigger transitions that can be taken "simultaneously", meaning that for any event E, for distinct t,u such that $E \rightsquigarrow t$ and $E \rightsquigarrow u$, the states $scope(t)$ and $scope(u)$ must be orthogonal. Broadcasting must lead to assignments to disjoint variables. That is, for any event E, for distinct t,u such that $E \rightsquigarrow t$ and $E \rightsquigarrow u$, $body(t)$ and $body(u)$ assign to disjoint variables. The broadcast relation $\rightsquigarrow$ must be a tree, i.e. an undirected acyclic graph. Figure 3 illustrates the restrictions on broadcasting: in (a) and (b) a transition on E broadcasts F but both transitions on E and F have same scope, *Root*; in (c) transitions on E and F have different scopes, U and *Root*, but these

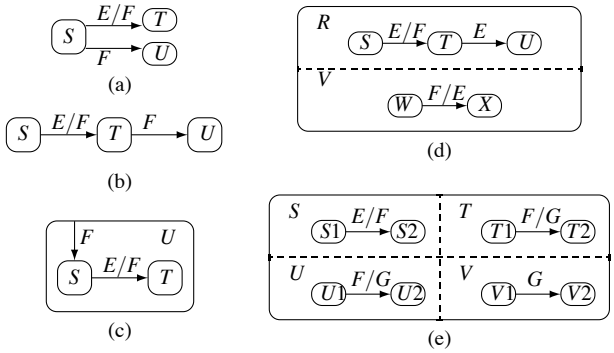

Figure 3. Examples violating restrictions on broadcasting.

are not orthogonal; in (d) the broadcast relation is cyclic as $E \rightsquigarrow F$ and $F \rightsquigarrow E$; in (e) event G is broadcast twice in one transition.

The path from state s to a set ss of descendants of s is the set of those states that are descendants of s and ancestors of states in ss, excluding s but including the states of ss.

$$path(s,ss) \quad \widehat{=} \quad children^+[\{s\}] \cap parent^*[ss]$$

The states entered by a transition are all the states on the path from the scope of the transition to its targets. The states exited by a transition are all the states on the path from the scope of the transition to its sources.

$$entered(t) \quad \widehat{=} \quad path(scope(t), target(t))$$
$$exited(t) \quad \widehat{=} \quad path(scope(t), source(t))$$

The implicit targets $imp(t)$ of a transition t are those children of the entered AND states of t that are not being explicitly entered.

$$imp(t) \quad \widehat{=} \quad children[entered(t) \cap AND] - entered(t)$$

The completion $comp(t)$ of a transition t is the set of all transitions that are taken when t is taken: it adds all default transitions of XOR targets of t and all default transitions of implicit targets of t. In Fig. 4 (a), the transition to U is added to the transition to T, in (b) the transition to X is added to the transition to T,U, and in (c) the transitions to U and W are added to the transition to S.

$$comp(t) \quad \widehat{=} \quad \{t\} \cup (\bigcup s \in (target(t) \cap XOR) \cup imp(t) \cdot comp(default(s)))$$

Completion requires $default(s)$ to be defined for all XOR targets of t and all implicit targets $imp(t)$, for all transitions t. Completion also requires that the children of an AND state are XOR states, and hence can have a default transition. The recursion is well-defined as the level, i.e. the distance to the root, of the scope of the parameter t increases with each call and the depth of every chart is bounded.

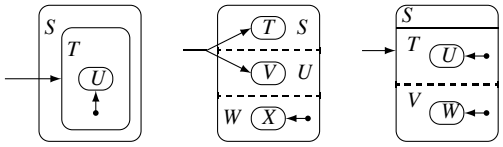

Figure 4. Examples for transition completion.

4 Configurations and Operations

The state of a statechart $\mathscr{S}$ is given by its *configuration* of states and by the global variables. In [12, 17] a configuration is defined as a maximal set of statechart states such that (1) it contains the root state, (2) for any XOR state it contains exactly one of its children, and (3) for any AND state it contains all of its children. We use here a different model that makes it easier to explain independent (concurrent) updates of a configuration [18]. For every XOR state s, including *Root*, a variable $lc(s)$, ranging over $uc(c)$ for every child c of s, is created. We interpret $lc(s)$ and $uc(s)$ to be the state s starting with a lowercase or an uppercase letter, for example:

$$\begin{array}{lcl} root & : & \{Standby, Working\} \\ picture & : & \{Displaying, WarmingUp\} \\ sound & : & \{Off, On, Waiting\} \end{array}$$

Formally, it is sufficient to assume that lc and uc are injective functions with disjoint ranges. This allows us to define the state test and state assignment for any state s that is a child of an XOR state:

$$\begin{array}{lcl} test(s) & \widehat{=} & lc(parent(s)) = uc(s) \\ assign(s) & \widehat{=} & lc(parent(s)) := uc(s) \end{array}$$

In the TV example, $test(s)$ and $assign(s)$ are defined for all states s except *Picture* and *Sound*, for example:

$$\begin{array}{lcl} test(Displaying) & = & (picture = Displaying) \\ test(Working) & = & (root = Working) \\ assign(On) & = & sound := On \\ assign(Working) & = & root := Working \end{array}$$

All other operations on configurations are expressed in terms of *test* and *assign*. The predicate $in(ss)$ tests if the current state is in the set ss; similarly $goto(ss)$ sets the current state to ss.

$$\begin{array}{lcl} in(ss) & \widehat{=} & \bigwedge s \in ss \cap children[XOR] \cdot test(s) \\ goto(ss) & \widehat{=} & \parallel s \in ss \cap children[XOR] \cdot assign(s) \end{array}$$

The statement $effect(t)$ executes the body of the transition t, goes to the states entered by t, and continues this for all transitions of the completion of t.

$$effect(t) \quad \widehat{=} \quad \parallel u \in comp(t) \cdot body(u) \parallel goto(entered(u))$$

The meaning of an event E is given by the operation $op(E)$, a statement; the meaning of a statechart is given by the set of all variables representing the statechart states and the set of operations of all events. The operation $op(E)$ gives priority to transitions on E on the outermost scope, *Root*, and then recursively to inner scopes. For brevity, we let $Trans(E,s)$ stand for the set of transitions on event E with scope s:

$$Trans(E,s) \;\widehat{=}\; \{t \in Transition \mid event(t) = E \wedge scope(t) = s\}$$

Transitions with the same event and scope are chosen nondeterministically. Transitions in children of AND states are taken in parallel.

$$\begin{array}{l} op(E) \quad \widehat{=} \quad scopeop(E, Root) \\ scopeop(E,s) \quad \widehat{=} \\ \quad (\,[]\; t \in Trans(E,s) \cdot \\ \qquad in(exited(t)) \wedge guard'(t) \rightarrow effect'(t)) \\ \quad /\!/\, childop(E,s) \\ childop(E,s) \quad \widehat{=} \\ \quad \mathsf{case}\, s\, \mathsf{of} \\ \qquad Basic : \mathsf{skip} \\ \qquad XOR : [] \; c \in children[\{s\}] \cdot test(c) \rightarrow scopeop(E,c) \\ \qquad AND : \parallel c \in children[\{s\}] \cdot scopeop(E,c) \\ \quad \mathsf{end} \end{array}$$

The expression $guard'(t)$ is like $guards(t)$, except that tests $\mathsf{in}\, S_1, \ldots, S_n$ are replaced by $in(parent^*[\{S_1, \ldots, S_n\}])$; that is, state tests are "completed". Statement $effect'(t)$ is like $effect(t)$ except that tests are replaced as in guards and broadcasts of E are replaced by $op(E)$. This may need to be repeated until all broadcasts are eliminated. As the broadcast relation $\rightsquigarrow$ is acyclic, this is well-defined. For example, with some simplifications we get:

$$\begin{array}{l} op(mute) \quad = \\ \quad (\; test(Standby) \rightarrow \mathsf{skip} \\ \quad [] \; test(Working) \rightarrow \\ \qquad (\; test(On) \rightarrow assign(Off) \\ \qquad [] \; test(Off) \rightarrow assign(On)) \\ \qquad /\!/\, \mathsf{skip}) \\ op(down) \quad = \\ \quad (\; test(Standby) \rightarrow \mathsf{skip} \\ \quad [] \; test(Working) \rightarrow \\ \qquad (\; test(On) \wedge lev > 1 \rightarrow lev := lev - 1 \parallel assign(On)) \\ \qquad /\!/\, \mathsf{skip}) \\ op(warm) \quad = \\ \quad (\; test(Standby) \rightarrow \mathsf{skip} \\ \quad [] \; test(Working) \rightarrow \\ \qquad (\; test(WarmingUp) \rightarrow \\ \qquad\quad op(soundOn) \parallel assign(Displaying)) \\ \qquad /\!/\, \mathsf{skip}) \end{array}$$

The simplifications carried out above are that the nondeterministic choice over the empty range is stop, $[]\ i \mid \mathsf{false} \cdot P = \mathsf{stop}$, that parallel composition over the empty range is skip, $\|\ i \mid \mathsf{false} \cdot P = \mathsf{skip}$, that $\mathsf{skip} \parallel P = P$, and that $\mathsf{stop} /\!/ P = P$.

This defines a semantics of statecharts in terms of nondeterministic programs. The semantics also serves as a translation scheme: the generated code is of the form $(g_1 \to P_1\ [] \ldots [] \ g_n \to P_n) /\!/ Q$, where Q is either skip, or a nondeterministic choice over guarded statements of that form, or the parallel composition of statements of that form. An implementation would evaluate g_i in some order and execute the corresponding P_i, or execute Q if all g_i are false. Care has to be taken in the implementation of parallel composition, as that requires that copies of selected variables for the chart states and selected global variables to be made such that their initial values are available.

The main differences compared to [7, 12, 17] are: (1) Boolean combinations of events are not allowed; it is not clear how they can be dealt with in an event-centric semantics. Negations of events are typically used to express priority among transitions; priority can here be expressed only through nesting, as transitions with outer scope take priority over transitions with inner scope. (2) Default states are generalized to default transitions. This allows a statement to be specified when a default state is entered that can establish a (local) invariant. (3) As there are no micro-steps, there is no need to take a fixpoint to determine the sequence of micro-steps; instead, all transitions on an event must be conflict-free so they can be taken simultaneously.

5 State Invariant Verification

An invariantchart $\mathscr{I}$ is a statechart structure with two additional components, *inv* and *Gobal*. The function *inv* maps every state to a Boolean expression, the *state invariant*. When I is attached to S, visually $S \mid I$, the state invariant $inv(S)$ is I, except with state tests $\mathsf{in}\, S_1, \ldots, S_n$ replaced by $in(parent^*[\{S_1, \ldots, S_n\}])$. If an invariant is missing in a chart, it is assumed to be true. Typically, we allow a richer set of Boolean expressions in invariants than in guards, though we do not make such a distinction here. The set *Global* is a non-empty subset of *Event*, the set of global events; all other events are local. The intention is that only transitions on global events need to establish the invariants. Transitions on local events can only occur as part of a transition on a global event, but not on their own. For the TV example, we define $Global = \{power, warm, down, up, mute\}$, which makes *soundOn* the only local event, and have:

$$\begin{array}{lcl} inv(Displaying) & = & \mathsf{true} \\ inv(Working) & = & \neg test(Waiting) \Rightarrow test(Displaying) \\ inv(Sound) & = & (1 \le lev) \wedge (lev \le 10) \\ inv(Root) & = & \mathsf{true} \end{array}$$

If a chart is in state s, then $inv(s)$ has to hold. If s is an XOR state, then $inv(c)$ for some child c of s has to hold as well, if s is an AND state, then $inv(c)$ for all children c of s has to hold, and so on. The child invariant $childinv(s)$ of state s is constructed recursively:

$$\begin{array}{l} childinv(s) \ \widehat{=} \\ \quad \mathsf{case}\, s\, \mathsf{of} \\ \quad\quad Basic : \mathsf{true} \\ \quad\quad XOR : \bigvee c \in children[\{s\}] \cdot test(c) \wedge inv(c) \wedge childinv(c) \\ \quad\quad AND : \bigwedge c \in children[\{s\}] \cdot inv(c) \wedge childinv(c) \\ \quad \mathsf{end} \end{array}$$

For example, with slight simplifications we have:

$$\begin{array}{l} childinv(Picture) \ = \\ \quad test(WarmingUp) \vee test(Displaying) \\ childinv(Working) \ = \\ \quad childinv(Picture) \wedge inv(Sound) \wedge childinv(Sound) \\ childinv(Root) \ = \\ \quad test(Standby) \vee \\ \quad (test(Working) \wedge inv(Working) \wedge childinv(Working)) \end{array}$$

The chart invariant of an invariantchart $\mathscr{I}$ is the state invariant of *Root* conjoined with the child invariant of *Root*:

$$chartinv \ \widehat{=} \ inv(Root) \wedge childinv(Root)$$

An invariantchart $\mathscr{I}$ is consistent if the operations of all global events preserve the chart invariant, more precisely if for all $E \in Global$:

$$\{chartinv\}\ op(E)\ \{chartinv\} \qquad (*)$$

In the TV example, this leads to five correctness conditions, one for each event $power, warm, down, up, mute$. We note that there are two transitions on *power* and *mute* and present now a scheme that allows these to be verified separately, thus splitting up a larger proof condition into two smaller ones. The observation underlying this is that if a chart is in state s then (1) it is in all ancestors of s, (2) the invariants $inv(a)$ of all ancestors a of s, including s, has to hold, (3) for all children of AND ancestors, except s itself, the invariant and the the child invariant have to hold, and (4) $childinv(s)$ has to hold; all other state invariants are irrelevant. The invariant constructed in this way is called the accumulated invariant $accinv(ss)$; it is generalized to a set ss of states where the closest common ancestor of any pair or states in ss is an AND state, but one may be the parent of the other. Note that $children[parent^+[ss] \cap AND]$ is the set of all children of the AND ancestors of ss. By subtracting $parent^*[ss]$ we get the set of all "AND uncles":

$$\begin{array}{l} accinv(ss) \ \widehat{=} \\ \quad in(parent^*[ss]) \wedge \\ \quad (\bigwedge s \in parent^*[ss] \cdot inv(s)) \wedge \\ \quad (\bigwedge s \in children[parent^+[ss] \cap AND] - parent^*[ss] \cdot \\ \quad\quad inv(s) \wedge childinv(s)) \wedge \\ \quad (\bigwedge s \in ss - parent^+[ss] \cdot childinv(s)) \end{array}$$

Following property illustrates the purpose of this definition: if the chart is in a set ss of states, then the chart invariant reduces to the accumulated invariant of this set:

$$in(parent^*[ss]) \wedge chartinv \equiv accinv(ss)$$

The source invariant of a transition t is the accumulated invariant of the source states; the target invariant of t requires that the accumulated invariant of the targets of the completion of t is taken, as a targeted states may have default transitions:

$$\begin{array}{lcl} sourceinv(t) & \widehat{=} & accinv(source(t)) \\ targetinv(t) & \widehat{=} & accinv(\bigcup u \in comp(t) \cdot target(u)) \end{array}$$

The main claim is an alternative way of checking (*) for $E \in Global$. The idea is to visit all transitions, starting those that the have the root state as their scope, and then to descend to all children, as in the definition of $op(E)$. The correctness condition of transition t on event E is in the simplest case:

$$\{sourceinv(t) \wedge guard'(t)\}\, effect'(t)\, \{targetinv(t)\}$$

In the case that an ancestor of s has other transitions on E in its scope, these transitions have priority. In the recursive definition below, the conjunction of the negations of all guards on E of one scope, expressed as $(\bigwedge t \in Trans(E,s) \cdot \neg guard'(t))$, is "assumed" when visiting the children:

$$\begin{array}{l} correct(E) \;\widehat{=}\; scopecorrect(E, Root) \\ scopecorrect(E,s) \;\widehat{=} \\ \quad (\bigwedge t \in Trans(E,s) \cdot \\ \qquad \{sourceinv(t) \wedge guard'(t)\}\, effect'(t)\, \{targetinv(t)\}) \\ \quad \wedge ((\bigwedge t \in Trans(E,s) \cdot \neg guard'(t)) \Rightarrow \\ \qquad childcorrect(E,s)) \\ childcorrect(E,s) \;\widehat{=} \\ \quad \mathsf{case}\, s\, \mathsf{of} \\ \qquad Basic : \mathsf{true} \\ \qquad XOR : \bigwedge c \in children[\{s\}] \cdot scopecorr(E,c) \\ \qquad AND : \{accinv(\{s\})\}\, childop(E,s)\, \{accinv(\{s\})\} \\ \quad \mathsf{end} \end{array}$$

The recursion stops when a Basic state or an AND state are encountered. The condition for an AND child (second last line) is equivalent to:

$$\begin{array}{l} \{accinv(\{s\})\} \\ \quad \| \; c \in children[\{s\}] \cdot scopeop(E,c) \\ \{accinv(\{s\})\} \end{array}$$

In general $\{p\}\, Q \parallel R\, \{r\}$ cannot be split into one condition for Q and one for R. However, further splitting of the condition is possible according to the structure of $scopeop(E,c)$. If the statement contains a nondeterministic choice with guards, we can apply:

$$\{p\}\, P \parallel (g \rightarrow Q \;[\!]\; h \rightarrow R)\, \{r\} \quad \equiv \quad \{p \wedge g\}\, P \parallel Q\, \{r\} \;\wedge\; \{p \wedge h\}\, P \parallel R\, \{r\}$$

This can be repeated until we arrive at a parallel composition of multiple assignment statements, which can be combined into a single assignment statement. The verification rule for assignments yields then a plain predicate than needs to be shown to hold.

For the TV chart, we abbreviate the accumulated invariant of *Working* by *WI* and get:

$$\begin{array}{l} WI \quad \equiv \\ \quad test(Working) \wedge (\neg test(Waiting) \Rightarrow test(Displaying)) \wedge \\ \quad 1 \leq lev \wedge lev \leq 10 \\ correct(power) \quad \equiv \\ \quad \{test(Standby)\} \\ \qquad assign(Working) \parallel assign(WarmingUp) \parallel \\ \qquad assign(Waiting) \parallel lev := 5 \\ \quad \{WI\} \\ \quad \wedge \\ \quad \{WI\}\, test(Working) \rightarrow assign(Standby)\, \{test(Standby)\} \\ correct(warm) \quad \equiv \\ \quad \{WI \wedge test(WarmingUp)\} \\ \qquad op(soundOn) \parallel assign(Displaying) \\ \quad \{WI\} \\ correct(mute) \quad \equiv \\ \quad \{WI\} \\ \qquad (\; test(On) \rightarrow assign(Off) \\ \qquad [\!]\; test(Off) \rightarrow assign(On)) \\ \qquad /\!/ \, \mathsf{skip} \\ \quad \{WI\} \end{array}$$

We conclude with formalizing our claim.

Theorem. *For invariantchart $\mathscr{I}$ and $E \in Global$:*

$$\{chartinv\}\, op(E)\, \{chartinv\} \quad \equiv \quad correct(E)$$

Proof. We show more generally that for any state s, if we consider only transition with scope a scope of s or a descendant of s, then $\{chartinv\}\, scopeop(E,s)\, \{chartinv\}$ and $scopecorrect(E,s)$ are equivalent:

$$\begin{array}{l} in(parent^*[\{s\}]) \wedge \\ (\bigwedge a \in parent^+[\{s\}] \cdot \bigwedge t \in Trans(E,a) \cdot \neg guard'(t)) \\ \Rightarrow \\ \{chartinv\}\, scopeop(E,s)\, \{chartinv\} \equiv scopecorrect(E,s) \end{array}$$

The theorem follows by taking $s = Root$. The proof then proceeds by induction over the distance between s and the scope of the enclosed transitions. We leave out the details.

6 Discussion

State invariants need to be inductive invariants rather than temporal invariants (safety properties). For example, *in On* $\Rightarrow$ *in Displaying* is a temporal invariant as it holds after every step, but is not strong enough to be used as the sole

state invariant of *Working*. By comparison, model-checking approaches require only temporal invariants to be stated. It remains to be seen which proves to be more helpful in the construction of complex systems.

The proof rule is still not as simple as one would wish. One reason is the need to explicitly accumulate (negations of) guards to express priority. One can question whether the occasional brevity in the chart that priorities allow is worth the constantly more complex verification conditions. The other reason is that children of AND states cannot be treated separately. In the TV example, the rule does not lead to separate verification conditions for transitions on *mute*. In this particular example, the result verification condition can be split into two using $\{p\}\, Q \,[]\, R\, \{r\} \equiv \{p\}\, Q\, \{r\} \wedge \{p\}\, T\, \{r\}$. In general, first the parallel composition would need to be distributed over the choice. If there are m concurrent states, each with n transitions on event E, then this results in n^m verification conditions for E, a reflection of the intricacy of concurrent states.

Acknowledgement. Dai Tri Man Le and Scott West were instrumental in adding state invariant verification to iState. The reviewers provided numerous constructive suggestions.

References

[1] J.-R. Abrial. *The B Book: Assigning Programs to Meaning*. Cambridge University Press, 1996.

[2] R.-J. Back. Invariant based programming. In S. Donatelli and P. S. Thiagarajan, editors, *27th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency*, Lecture Notes in Computer Science 4024, pages 1–18, Turku, Finland, 2006. Springer-Verlag.

[3] M. Balser, S. Bäumler, A. Knapp, W. Reif, and A. Thums. Interactive verification of UML state machines. In J. Davies, W. Schulte, and M. Barnett, editors, *International Conference on Formal Engineering Methods*, Lecture Notes in Computer Science 3308, pages 434–448, Seattle, WA, USA, 2004. Springer.

[4] M. v. d. Beeck. A comparison of statechart variants. In H. Langmaack, W.-P. deRoever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science 863, pages 128–148, Lübeck, Germany, 1994. Springer-Verlag.

[5] P. Bhaduri and S. Ramesh. Model checking of statechart models: Survey and research directions. Technical Report cs.SE/0407038, arXiv, July 2004.

[6] E. Clarke and W. Heinle. Modular translation of statecharts to SMV. Technical Report CMU-CS-00-XXX, School of Computer Science, Carnegie Mellon University, August 2000.

[7] W. Damm, B. Jasko, H. Hungar, and A. Pnueli. A compositional real-time semantics of STATEMATE designs. In W.-P. d. Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality: The Significant Difference*, Lecture Notes in Computer Science 1536, pages 186–238, Bad Malente, Germany, 1998. Springer-Verlag.

[8] N. Day and J. Joyce. The semantics of statecharts in hol. In J. Joyce and C.-J. Seger, editors, *Higher Order Logic Theorem Proving and its Applications*, Lecture Notes in Computer Science 780, pages 338–351, Vancouver, BC, Canada, 1994. Springer-Verlag.

[9] R. Eshuis, D. N. Jansen, and R. Wieringa. Requirements-level semantics and model checking of object-oriented statecharts. *Requirements Engineering*, 7(6):243–263, 2002.

[10] A. Galloway, T. Cockram, and J. McDermid. Experiences with the application of discrete formal methods to the development of engine control software. In *Distributed Computer Control Systems*. International Federation of Automatic Control, 1998.

[11] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

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

[13] A. Knapp, S. Merz, and C. Rauh. Model checking timed UML state machines and collaborations. In W. Damm and E.-R. Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science 2469, pages 395–416, Oldenburg, Germany, 2002. Springer.

[14] D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[15] J. Lilius and I. P. Paltor. Formalising UML state machines for model checking. In R. France and B. Rumpe, editors, *UML'99 – The Unified Modeling Language Beyond the Standard*, Lecture Notes in Computer Science 1723, pages 430–445, Fort Collins, Colorado, 1999. Springer-Verlag.

[16] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. Implementing statecharts in Promela / Spin. In *Second IEEE Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, 1998. IEEE Computer Society Press.

[17] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In T. Ito and A. Meyer, editors, *Proceedings of the 1st International Conference on Theoretical Aspects of Computer Software, TACS '91*, Lecture Notes in Computer Science 526, pages 244–264, Sendai, Japan, 1991. Springer-Verlag.

[18] E. Sekerinski. Graphical design of reactive systems. In D. Bert, editor, *2nd International B Conference*, Lecture Notes in Computer Science 1393, pages 182–197, Montpellier, France, 1998. Springer-Verlag.

[19] E. Sekerinski and R. Zurob. iState: A statechart translator. In M. Gogolla and C. Kobryn, editors, *UML 2001 – The Unified Modeling Language, 4th International Conference*, Lecture Notes in Computer Science 2185, pages 376–390, Toronto, Canada, 2001. Springer-Verlag.

[20] E. Sekerinski and R. Zurob. Translating statecharts to B. In M. Butler, L. Petre, and K. Sere, editors, *Third International Conference on Integrated Formal Methods*, Lecture Notes in Computer Science 2335, pages 128–144, Turku, Finland, 2002. Springer-Verlag.