CHAPTER 1

DESIGN VERIFICATION WITH STATE INVARIANTS

1.1 INTRODUCTION

This article is on statically verifying the design expressed by a statechart. Statecharts extend finite state machines with *clustering*, expressed by XOR states, with *concurrency*, expressed by AND states, and with *broadcast communication*. Both XOR and AND states structure the states hierarchically: if a chart is in an XOR state, it must be in exactly one of its children; if the chart is in an AND state, it must be in all of its children. Transitions between states can assign to and depend on global variables of arbitrary types, thus lifting the restriction to a finite number of states; the statechart states partition the combined state of the chart and the variables into *modes*. These extensions of finite state machines are meant to allow the requirements of embedded systems to be directly expressed [6].

Statecharts are appealing to practitioners as the underlying formalism of finite state machines is well-understood, as the visual designs are "easy to communicate to domain experts", and as statecharts can be directly executed through interpretation or compilation. With their inclusion in UML, statecharts are used for object-oriented design.

UML 2 Semantics and Applications. Edited by Kevin Lano
Copyright (c) 2009 John Wiley & Sons, Inc
PREPRINT

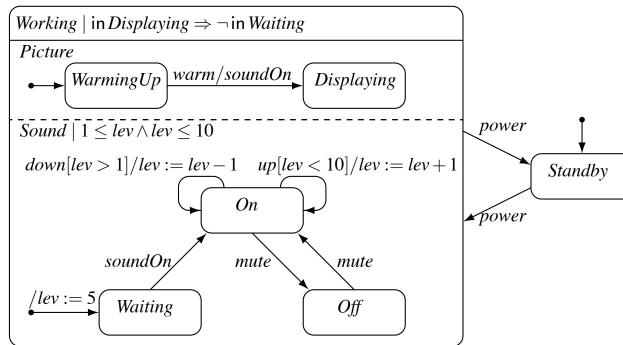


Figure 1.1 Statechart with invariants for a TV set.

Statecharts on their own do not immediately lead to opportunities for verifying the safety of the design: if an event is received and no transition on that event can take place, it is ignored. There is no intrinsic notion that an error occurs or an invalid state is reached for a given sequence of events. This reflects the requirement that embedded systems have to be robust and be prepared for arbitrary behaviour of their environment.

This article explores design verification through *state invariants*. These are conditions that are attached to individual states and specify what has to hold in that state. If a state S has invariant I attached to it, then every incoming transition must ensure that I holds. Dually, every outgoing transition can assume that I holds initially. This gives a method for checking a chart against an annotation consisting of invariants attached to states in the state hierarchy. Intuitively, state invariants document the "purpose" of states. In UML, state invariants can be attached to behavioural state machines and protocol state machines [18].

Consider the TV control example in Figure 1.1. The activity is partitioned into two states, the Basic state *Standby* and the AND state *Working*. When in *Working* state, the chart is in both *Picture* and *Sound* XOR states. Within *Picture* the chart is in one of the basic states *WarmingUp* and *Displaying*, within *Sound* the system is in one of the basic states *Waiting*, *On*, and *Off*. The invariant of *Working* is that whenever *Picture* is in *Displaying*, *Sound* must not be in *Waiting*, i.e. must be in *On* or *Off*. The invariant of *Sound* states that the sound level *lev* must be between 1 and 10. The event *power* causes the chart to flip between *Standby* and *Working*, no matter in which substates of *Working* the chart 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 can express safety of an embedded system or consistency of a software system. Compared to writing an equivalent combined invariant as a single global predicate, state invariants allow a potentially large invariant to be decomposed into parts that are in visual proximity to affected transitions, making complex invari-

ants more comprehensible. State invariants allow for shorter invariant expressions as *state tests* are implicit to the state to which an invariant is attached.

The interpretation of XOR and AND states carries over to state invariants: if a chart is in an XOR state, its invariant and the invariant of exactly one if its children have to hold; if the chart is in an AND state, its invariant and the invariants of all its children have to hold. As a consequence, the attached invariant of a state is *inherited* by all its children and thus all its descendants. Dually, the children of a state contribute to a *synthesized* invariant that is passed on to their parent and thus all their ancestors. The conjunction of the attached invariant, the inherited invariant, and the synthesized invariant of a state is called the *accumulated* invariant. A contribution of this article is to define accumulation formally and to justify it (Section 1.6).

State invariants can be used for verifying a design by testing or by static verification. The use for testing is conceptually simple: after each transition the accumulated invariant of the target states have to be checked; more precisely, for all leaf states in which the chart ends up, the attached invariant of those states as well as the attached invariant of all their ancestors have to hold. To check this at run-time, the evaluation of invariants has to be sufficiently efficient. Here we consider static verification and don't impose restrictions on the composition of invariants.

Static verification proceeds by generating a number of *verification conditions* from the annotated chart and then showing that these hold. This depends on the definition of a transition, which in presence of broadcasting can have different interpretations. The interpretation taken here is that all transitions resulting from broadcasts are to be taken simultaneously with the initiating transition, which we call *simultaneous broadcasting*. Thus, if in the chart of Figure 1.1 event *warm* is received when the chart is initially in *WarmingUp* and *Waiting*, the transitions on *warm* and *soundOn* are taken simultaneously and the invariant of *Working* is preserved.

Simultaneous broadcasting can be formalized using *parallel composition* of statements. The B Method subsumes an extension of guarded commands by parallel composition [1]. A number of approaches define statecharts by translation to the B Method [9, 10, 11, 13, 15, 17, 20, 24]. The (bounded) nondeterminism of guarded commands allow the nondeterminism in the choice of transitions to be reflected. As the B Method also supports proofs of invariants, such a translation leads to a method for proving preservation of accumulated invariants, to be precise with one verification condition per event. The second contribution of this article is a procedure that instead generates several smaller, "more local" verification conditions per event and justifies this in terms of the straightforward generation (Section 1.7). Automated theorem provers are more effective at proving or disproving many small conditions than few large ones. Thus the prospect is that state invariants not only make it easier to specify correctness conditions for statecharts but also make it easier to verify them.

The original interpretation of broadcasting leads to a sequence of internal *micro-steps*. In the above example this implies that first the transition on *warm* is taken, resulting in *Picture* being in *Displaying* and *Sound* remaining in *Waiting*, hence violating the invariant $\mathsf{in}\,\mathit{Displaying} \Rightarrow \neg\,\mathsf{in}\,\mathit{Waiting}$. Thus the transition on *soundOn* would be taken in a configuration when the accumulated invariant of its source state does not hold. As the transition on *soundOn* follows immediately, this violation is

not observable from outside. This interpretation necessitates that the invariant be relaxed to the following one, where $\mathsf{gen}\,E$ means that event E has been generated and is awaiting processing:

$$Working \mid \mathsf{in}\,Displaying \Rightarrow (\neg\,\mathsf{in}\,Waiting) \vee (\mathsf{in}\,Waiting \wedge \mathsf{gen}\,soundOn)$$

The set of generated events needs to be kept in a global variable and determines the next micro-step in a loop that is executed as long as the set is not empty. A verification method that attempts to be complete needs to allow this sequence to be referred to in invariants—like through the function gen above. Simultaneous broadcasting does not need such a set and allows events to be interpreted as operations (procedures). While micro-steps allow the same transition to be taken repeatedly within a macro-step, potentially leading to non-termination, simultaneous broadcasting forbids this. As intermediate states are not present with simultaneous broadcasting, it is more abstract than sequencing micro-steps. An implementation of simultaneous broadcasting would still need to introduce intermediate states following the refinement rules of parallel composition [1].

Nondeterminism arises in statecharts if more than one transition is enabled. Classical statecharts [4, 7, 19] and UML statecharts [18] resolve nondeterminism that can arise due to transitions on different levels differently: classical statecharts give priority to outer transitions, as this facilitates zooming in and out of complex states; UML statecharts give priority to inner transition, as an inner state can "override" the behaviour of an outer state. As a third contribution of this article, we study the consequences of resolving this nondeterminism either way for invariant verification and code generation (Section 1.8).

The final contribution of this article is a discussion on when and how to use state invariants (Section 1.9). After preliminaries (Section 1.2), we first define the (syntactic) statechart structure (Section 1.3), the meaning of statecharts in term of configurations and operations (Section 1.4), and the meaning of state invariants (Section 1.5).

Formal verification of statecharts has been studied extensively, e.g. in [3, 5, 8, 12, 14, 16], see [2] for a survey on model-checking approaches. These approaches specify invariants globally, rather than attaching them to states. However, they allow more general temporal properties than invariance that we consider here.

This line of work emerged from an attempt to generate comprehensible code from statecharts, as a way of cross-checking the statechart design [22, 23]. Compared to the approach there, a pre-processing step that leads to normalized statecharts is eliminated, as this step became awkward in an interactive tool. Also, the translation scheme is described more abstractly and the well-formedness criteria are revised and justified. This article revises and extends the verification condition generation of [21].

1.2 PRELIMINARIES

We use *generalized program statements* to define the meaning of an event. Generalized statements subsume those that may appear in a body of a transition. We are interested in models that are sufficiently abstract such that transition bodies do not

contain loops and recursion but may contain conditionals. To simplify matters, we assume that the evaluation of expressions is always defined.

A (generalized) statement P is defined by a pair, a *predicate* or *Boolean expression* $[P]$ relating the initial and final states, and a list αP of *variables* that are assigned to by P. The initial and final states are referred to by unprimed and primed variables. Let g be a Boolean expression, xv a list of unique variables, ev a list of expressions of the same length as xv, and Q, R statements:

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

The statement skip can always be executed and does not change any variables. The statement stop can never be executed, i.e. is always disabled. The multiple assignment $xv := ev$ assigns simultaneously the values of ev to the variables xv. The guarded statement $g \to Q$ blocks if g does not hold, otherwise is as Q. The *nondeterministic choice* $Q \mathbin{[\!]} R$ selects either operand that is enabled; if both are enabled, their choice is arbitrary, if neither is enabled, $Q \mathbin{[\!]} R$ blocks. The *parallel* or *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. Parallel composition is a generalization of multiple assignment, in the sense that $(x, y := e, f) = (x := e \parallel y := f)$. The *sequential composition* $Q \,;\, R$ joins the final variables of Q with the initial variables of R, formally expressed by renaming: $e[xv \backslash ev]$ stands for expression e with each occurrence of a variable of xv replaced by the corresponding expression in ev. Sequential composition is always well-defined. The *conditional statement* is defined in terms of the above:

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

The *enabledness domain* en P is the domain of the relation of statement P:

$$\mathsf{en}\ P = \exists xv' \,.\, [P] \quad \text{where} \quad xv = \alpha P$$

For example, en skip = true and en stop = false. The *prioritizing composition* $P \,/\!/\, Q$ is like P, if P is enabled, otherwise it is like Q:

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

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

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

The common verification rules for statements hold, for example:

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

1.3 STATECHART STRUCTURE

A statechart $\mathcal{S}$ 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 children of an AND state must be XOR states.

The finite set $Event$ is that of event names. The elements of the finite set $Transition$ are tuples t, represented as $ss \xrightarrow{t:E[g]/B} ts$, 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 chart expression, the transition guard, and $body(t) = B$ is a chart statement, the transition body. The state $Root$ must not be the source or target of any transition, $Root \notin source(t)$ and $Root \notin target(t)$ for any $t \in Transition$. All transitions must have at least one source state and one target state, $source(t) \neq \{\}$ 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, $source(default(s)) = \{s\}$. A fat dot inside the source state is used to visualize the source of a default transition. Certain XOR states are "required to have a default transition": 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 will be 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.

Chart 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.

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

Chart 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 the specifications of the transition name t:, the transition guard $[g]$, and the transition body $/B$ 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 and guard of a default transition do not play any role and are always left out.

The *closest common ancestor* $cca(ss)$ of a set ss of states is the state that is a proper 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.

$$c = cca(ss) \quad \equiv \quad c \in parent^+[ss] \wedge (\forall a \in State\ .\ a \in parent^+[ss] \Rightarrow a\, parent^*\, c)$$

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. 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 $t \in Transition$. This concludes the definition of the statechart structure.

For example, in Figure 1.2, states X and Z are orthogonal, as their closest common ancestor, V, is an AND state and neither is an ancestor of the other. States X and T are not orthogonal, as their closest common ancestor, S is and XOR state. States W and X are not orthogonal as W is an ancestor of X, though their closest common ancestor, V, is an XOR state.

We continue with several useful definitions. The *scope* of a transition is the state closest to the root through which the transition passes.

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

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.

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

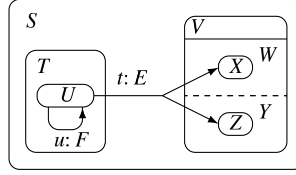


Figure 1.2 Self-transition and inter-level transition.

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.

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

Figure 1.2 defines $source(t) = \{U\}$ and $target(t) = \{X, Z\}$. The scope of t is the closest common ancestor of $\{U, X, Z\}$, which is S, thus $entered(t) = \{V, W, X, Y, Z\}$ and $exited(t) = \{U, T\}$. We also have that $source(u) = \{U\} = target(u)$. The scope of u is the closest common ancestor of $\{U\}$, which is T, thus $entered(u) = \{U\} = exited(u)$.

Given a state set ss, the *implicit children* are those children of AND states of ss that are not in ss. If a chart is in ss, it is also in all its implicit children.

$$imp(ss) \;\widehat{=}\; children[ss \cap AND] - ss$$

The *completion* 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.

$$comp(t) \;\widehat{=}\; \{t\} \cup (\bigcup s \in (target(t) \cap XOR) \cup imp(entered(t)) \,.\, comp(default(s)))$$

In Figure 1.3 (a) we have that $target(t) = \{U\}$, an XOR state, $default(U) = u$, and therefore $comp(t) = \{t, u\}$. In (b) we have that $entered(t) = \{T, U, V, W, X\}$ and $imp(entered(t)) = \{Y\}$. As $default(Y) = u$, we get $comp(t) = \{t, u\}$. In (c) we have that $entered(t) = \{T\}$ and $imp(entered(t)) = \{U, V\}$. Thus we get $comp(t) = \{t, u, v\}$.

We are now in a position to define formally when an XOR state is "required to have a default transition": a default transition has to be defined for the root state, $Root \in \mathsf{dom}\, default$, and for all XOR targets s of t and all implicit targets $imp(t)$, for all transitions t, formally:

$$\forall t \in Transition \,.\, (target(t) \cap XOR) \cup imp(entered(t)) \subseteq \mathsf{dom}\, default$$

With this restriction on statecharts, $comp(t)$ is well-defined for any transition t, as in the definition s in $default(s)$ ranges over XOR states are required to have a default transition. Furthermore, the recursion terminates 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 statechart is bounded.

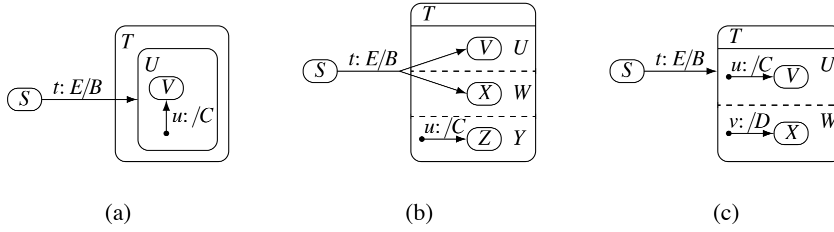


Figure 1.3 Examples for transition completion.

1.4 CONFIGURATIONS AND OPERATIONS

The "state" of a statechart $\mathcal{S}$ is given by its *configuration* of states and by the state of its global variables. A configuration can be 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 [7, 19]. We use here a different model that makes it easier to explain independent (concurrent) updates of a configuration [20]. For every XOR state s, including *Root*, a variable $lc(s)$, ranging over $uc(c)$ for every child c of s, is declared. We interpret $lc(s)$ and $uc(s)$ to be the state s starting with a lowercase or an uppercase letter. For the statechart of Figure 1.4 we get:

$$root : \{R, S\} \qquad t : \{U\} \qquad v : \{W, X\} \qquad x : \{Y, Z\}$$

Note the use of X as a value of variable v and the use of x as a variable. Formally, it is sufficient to assume that *lc* and *uc* are injective functions with disjoint ranges. The function *var* is defined to map the variable names to the set of possible values, e.g. $var(root) = \{R, S\}$. Thus *var* defines the set of possible configurations. We assume that these variables and their values are distinct from the global program variables. This model allows to define the *state test* and *state assignment* for any state s that is a child of an XOR state by inspecting and assigning the variable for that state:

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

In Figure 1.4, $test(s)$ and $assign(s)$ are defined for all states s except T and V:

$$\begin{array}{lcl@{\qquad}lcl} test(R) & \equiv & root = R & assign(R) & = & root := R \\ test(S) & \equiv & root = S & assign(S) & = & root := R \\ test(U) & \equiv & t = U & assign(U) & = & t := U \\ test(W) & \equiv & v = W & assign(W) & = & v := W \\ test(X) & \equiv & v = X & assign(X) & = & v := X \\ test(Y) & \equiv & x = Y & assign(Y) & = & x := Y \\ test(Z) & \equiv & x = Z & assign(Z) & = & x := Z \end{array}$$

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

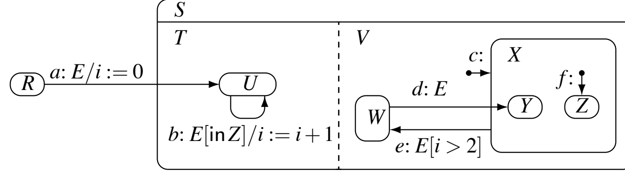


Figure 1.4 State hierarchy with transitions.

current state to *ss*.

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

The statement $goto(ss)$ is well-defined if the states of *ss* are not exclusive. For example, in Figure 1.4, $goto(\{U,X\})$ and $goto(\{X,Y\})$ are well-defined, but $goto(\{Y,Z\})$ is not.

The *trigger* of a transition t is a predicate that checks if the transition guard holds and if the system is in all source states; only all exited states are tested. The *effect* of a statement t is to execute the body of t, to go to the states entered by t, and to repeat this for all transitions of the completion of t.

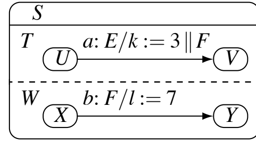
$$\begin{array}{lcl} trigger(t) & \widehat{=} & in(exited(t)) \wedge guard(t) \\ effect(t) & \widehat{=} & \| \; u \in comp(t) \,.\, body(u) \,\|\, goto(entered(u)) \end{array}$$

We allow ourselves to confuse the chart expression $guard(t)$ with its meaning as an expression and chart statement $body(u)$ with its meaning as a statement, whereby a broadcast of E occurring in a transition body is defined by $op(E)$, to be made precise further below, and a state test $\mathsf{in}\, S_1, \ldots, S_n$ occurring in the guard or body of transition t, written $\mathsf{in}_t\, S_1, \ldots, S_n$ is defined as testing being in $S_1, \ldots, S_n$ relative to being in $source(t)$:

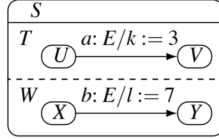
$$\mathsf{in}_t\, S_1, \ldots, S_n \quad \widehat{=} \quad in(parent^*[\{S_1, \ldots, S_n\}] - parent^*[source(t)])$$

The *goto* statement of $effect(t)$ is always well-defined as entered states are not exclusive. For Figure 1.4, noting that $comp(a) = \{a, c, f\}$, $comp(b) = \{b\}$, and $body(c) = \mathsf{skip} = body(f)$ we get:

$$\begin{array}{lcl} trigger(a) & \equiv & in(\{R\}) \wedge true \\ & \equiv & test(R) \\ effect(a) & = & body(a) \,\|\, goto(\{S,T,U\}) \,\|\, goto(\{X\}) \,\|\, goto(\{Z\}) \\ & = & i := 0 \,\|\, assign(S) \,\|\, assign(U) \,\|\, assign(X) \,\|\, assign(Z) \\ trigger(b) & \equiv & in(\{U\}) \wedge \mathsf{in}_b Z \\ & \equiv & test(U) \wedge test(X) \wedge test(Z) \\ effect(b) & = & body(b) \,\|\, goto(\{U\}) \\ & = & i := i+1 \,\|\, assign(U) \end{array}$$



$$\begin{aligned} op(E) = test(S) \rightarrow & \\ & (\ test(U) \rightarrow k := 3 \parallel op(F) \parallel assign(V) \\ & [\!]\ test(V) \rightarrow \mathsf{skip}) \\ op(F) = test(S) \rightarrow & \\ & (\ test(X) \rightarrow l := 7 \parallel assign(Y) \\ & [\!]\ test(Y) \rightarrow \mathsf{skip}) \end{aligned}$$



$$\begin{aligned} op(E) = test(S) \rightarrow & \\ & (\ (\ test(U) \rightarrow k := 3 \parallel assign(V) \\ & \quad [\!]\ test(V) \rightarrow \mathsf{skip}) \\ & \parallel (\ test(X) \rightarrow l := 7 \parallel assign(Y) \\ & \quad [\!]\ test(Y) \rightarrow \mathsf{skip})) \end{aligned}$$

Figure 1.5 Concurrent transitions and broadcasting.

The simplification carried out above is that $\mathsf{skip} \parallel P = P$ for any statement P.

The *operation* of an event E is a statement that captures the joint effect of all transitions in a chart on E. For brevity, let $Trans(E,s)$ stand for the set of transitions on event E with scope s:

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

The operation $op(E)$ is defined by recursively visiting all transition on E, starting with those on the outermost scope, *Root*. In case there is a choice between transitions with the same scope, one is selected arbitrarily. In case there is a choice between transitions on different scopes, transition on outer scopes are given priority. All transitions on the same event in concurrent states are taken in parallel. Of all transitions in an exclusive state, at most one can be taken.

$$\begin{array}{lcl} op(E) & \widehat{=} & scopeop(E, Root) \\ scopeop(E,s) & \widehat{=} & ([\!]\ t \in Trans(E,s)\ .\ trigger(t) \rightarrow effect(t)) \,/\!/\, childop(E,s) \\ childop(E,s) & \widehat{=} & \mathsf{case}\ s\ \mathsf{of} \\ & & \quad Basic : \mathsf{skip} \\ & & \quad XOR : [\!]\ c \in children[\{s\}]\ .\ test(c) \rightarrow scopeop(E,c) \\ & & \quad AND : \parallel c \in children[\{s\}]\ .\ scopeop(E,c) \\ & & \mathsf{end} \end{array}$$

Figure 1.5 gives two examples. In Figure 1.4 there is one event, E, with four transitions on it. With simplifications we get:

$$
\begin{aligned}
op(E) \quad = \quad & test(R) \rightarrow i := 0 \parallel assign(S) \parallel assign(U) \parallel assign(X) \parallel assign(Z) \\
& /\!/ (\ test(R) \rightarrow \mathsf{skip} \\
& \quad [\!]\ test(S) \rightarrow \\
& \quad\quad (\ (test(U) \wedge test(X) \wedge test(Z) \rightarrow i := i+1 \parallel assign(U)) \\
& \quad\quad\quad /\!/ \mathsf{skip} \\
& \quad\quad \parallel (\ test(W) \rightarrow assign(X) \parallel assign(Y) \\
& \quad\quad\quad [\!]\ test(X) \wedge i > 2 \rightarrow assign(W)) \\
& \quad\quad\quad /\!/ \mathsf{skip}))
\end{aligned}
$$

The simplifications are that choice over the empty range is stop, $[\!]\ i \in \{\} \,.\, P = \mathsf{stop}$, that parallel composition over the empty range is skip, $\parallel i \in \{\} \,.\, P = \mathsf{skip}$, that $\mathsf{skip} \parallel P = P$, that $\mathsf{stop}\ [\!]\ P = P$, that $\mathsf{stop} /\!/ P = P$, that $g \rightarrow P\ [\!]\ h \rightarrow P = g \vee h \rightarrow P$, and that $true \rightarrow P = P$.

The *semantics* of statechart $\mathcal{S}$ is defined by the pair of functions *var* and *op*, with *var* defining the possible configurations and *op* defining for each event a (possibly nondeterministic) statement operating on the configuration.

Well-Definedness

The definition of *op* restricts the statecharts to which a meaning can be given. These restrictions arise due to the use of parallel composition, which requires that operands assign to distinct variables, and due to possible recursion in the definition of *op*, which results from broadcasting. The following two conditions are sufficient and necessary:

1. $effect(t)$ must be well-defined, for all transitions t,
2. $effect(t) \parallel effect(u)$ must be well-defined, for all t, u such that $event(t) = event(u)$ and $scope(t) \perp scope(u)$.

The first condition excludes transition bodies like $k := 3 \parallel k := 7$ and the charts of Figure 1.6: In (a), the broadcast of F results in two parallel assignments to k. In (b), as

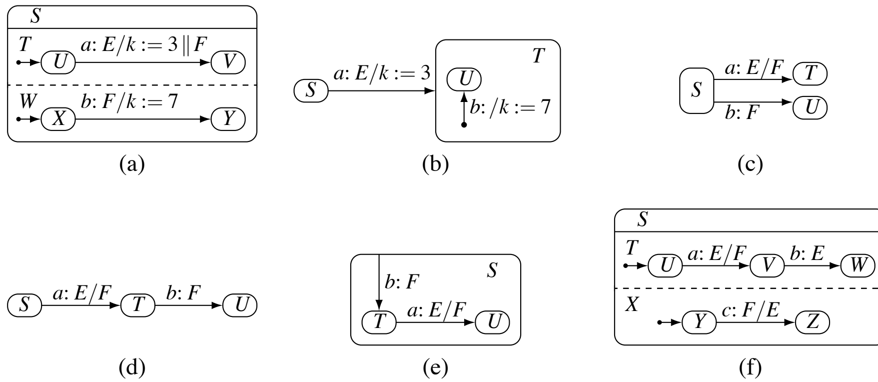


Figure 1.6 Violations of well-definedness condition 1.

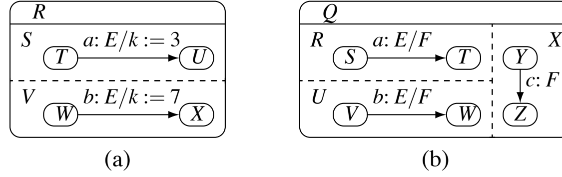


Figure 1.7 Violations of well-definedness condition 2.

the completion of a includes b, the effect of a includes again two parallel assignments to k. In (c), transition a leads to $assign(T) \parallel assign(U)$, which would result in parallel assignments to the same state variable, as is the case in (d) and (e). In (f), transition a leads to transitions c and b being taken, which results in $assign(V) \parallel assign(W)$. More generally, this condition prohibits any direct or indirect recursion among events, as these lead to parallel assignments to the same state variable.

The second condition excludes charts of Figure 1.7: In (a), on event E, both transitions a and b would be taken, resulting in parallel assignments to k. In (b), on event E, event F would be broadcast twice, resulting in $assign(Z) \parallel assign(Z)$.

Condition 1 ensures that *scopeop* is well-defined, provided that *childop* is well-defined. Condition 2 ensures that *childop* is well-defined, provided that *scopeop* is well-defined. Condition 1 also disallows any direct or indirect recursion among event operations. Hence these two conditions are sufficient and necessary. We nevertheless consider a third condition:

3. If the body of transition t contains a broadcast of event E and u is a transition on E, then t and u must be within concurrent states, i.e. $scope(t) \perp scope(u)$.

In Figure 1.8 (a), if the chart is in S and T, on event E both transitions a and b would be taken, as the effect of a is $assign(V) \parallel assign(U)$. Likewise, in (b) on F both transitions would be taken. In both cases the chart does not end up being in the targets of transitions taken due to broadcasting of events with transitions at outer levels. The above condition restricts broadcasting to events with transitions only in concurrent states.

Code Generation

The semantics of a chart can be directly expressed as a single MACHINE in the B Method. The VARIABLES of the machine are derived from the function *var* and the OPERATIONS define each event E by $op(E)$ as follows. The code for $scopeop(E, s)$ is a SELECT statement:

$$\begin{array}{l} (\ trigger(t_1) \rightarrow effect(t_1) \\ [\!]\ \ldots \\ [\!]\ trigger(t_n) \rightarrow effect(t_n)) \\ /\!/\ childop(E, s) \end{array}$$

```
SELECT trigger(t1) THEN effect(t1)
WHEN ...
WHEN trigger(tn) THEN effect(tn)
ELSE childop(E, s)
END
```

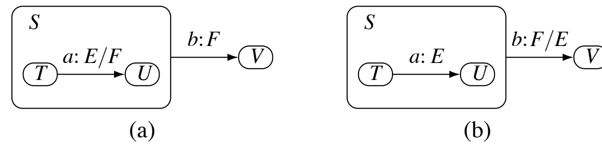


Figure 1.8 Violations of well-definedness condition 3.

The code for $childop(E,s)$, for an XOR state s, is a CASE statement:

$$
\begin{array}{l}
(\ r = S_1 \rightarrow scopeop(E,S_1) \\
[]\ \ldots \\
[]\ r = S_n \rightarrow scopeop(E,S_n))
\end{array}
$$

```
CASE r OF
   EITHER S1 THEN scopeop(E, S1)
   OR ...
   OR Sn THEN scopeop(E, Sn)
   END
END
```

The generated code can be further simplified. If there is only a single transition on a level for an event, the generated code is of the form SELECT g THEN Q ELSE R END and can be written as IF g THEN Q ELSE R END instead. CASE statements can be simplified by leaving out all alternatives with body *skip* and adding ELSE *skip* instead. CASE statements with a single alternative can be rewritten as IF statements. An IF statement of the form IF b THEN Q ELSE *skip* END can be simplified to IF g THEN Q END.

Figure 1.9 gives the code of the TV example as generated by the iState tool [23]. The generated code preserves the broadcasting structure by calling the operating of the broadcast event rather than inlining it. As the B Method does not allow calls of operations within the same machine, this is expressed in terms of auxiliary DEFINITIONS. The B Method also requires that all variables are initialized. In case the value of a state variable is initially irrelevant, a nondeterministic assignment is generated.

For generating an executable implementation, the SELECT statement needs to be refined by an IF statement in which the guards are evaluated in some arbitrary order. An implementation of parallel composition by sequential composition requires in general that copies of the involved state variables and global variables are made such that their initial values are available to all statements of the parallel composition. If there is no dependency on the initial values, these copies are not needed. For example, in Figure 1.9 all parallel compositions can be implemented by sequential compositions in any order. In principle the elimination of parallel composition can be automated.

1.5 STATE INVARIANT VERIFICATION

A statechart with invariants I, or *invariantchart* for short, is a statechart structure with two additional components, *inv* and *Gobal*. The function *inv* maps every state to a Boolean chart expression, the *state invariant*. Attaching chart expression I to state

```
MACHINE TV

SETS
    ROOT = {Standby, Working};
    PICTURE = {Displaying, WarmingUp};
    SOUND = {Waiting, On, Off}

VARIABLES
    root,
    picture,
    sound,
    lev

INVARIANT
    root : ROOT ∧
    picture : PICTURE ∧
    sound : SOUND ∧
    lev : INTEGER

INITIALISATION
    root := Standby
     ||
    picture :∈ PICTURE
     ||
    sound :∈ SOUND
     ||
    lev :∈ INTEGER


DEFINITIONS
DEF_soundOn ==
    IF (root = Working) THEN
        IF (sound = Waiting) THEN
            sound := On
        END
    END


OPERATIONS
mute =
    IF (root = Working) THEN
        CASE sound OF
            EITHER Off THEN
                sound := On
            OR On THEN
                sound := Off
            OR Waiting THEN
                skip
            END
        END
    END
;
power =
    CASE root OF
        EITHER Working THEN
            root := Standby
        OR Standby THEN
            lev := 5
             ||
            root := Working
             ||
            picture := WarmingUp
             ||
            sound := Waiting
        END
    END
;
up =
    IF (root = Working) THEN
        IF (sound = On) THEN
            IF (lev < 10) THEN
                lev := (lev + 1)
                 ||
                sound := On
            END
        END
    END
;
down =
    IF (root = Working) THEN
        IF (sound = On) THEN
            IF (lev > 1) THEN
                lev := (lev − 1)
                 ||
                sound := On
            END
        END
    END
;
soundOn =
    DEF_soundOn
;
warm =
    IF (root = Working) THEN
        IF (picture = WarmingUp) THEN
            DEF_soundOn
             ||
            picture := Displaying
        END
    END

END
```

Figure 1.9 B code of TV example.

S, visually $S \mid I$, defines $inv(S)$ to be I. If no invariant is attached, $inv(S)$ 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. The global events are the interface through which the environment asks the system for a response.

We allow ourselves to confuse a chart expression attached to a state with its meaning as an expression, whereby a state test in $S_1, \ldots, S_n$ occurring in I attached to S, indicated by writing $\mathsf{in}_S\, S_1, \ldots, S_n$ is defined as testing being in $S_1, \ldots, S_n$ relative to being in S:

$$\mathsf{in}_S\, S_1, \ldots, S_n \quad \widehat{=} \quad in(parent^*[\{S_1, \ldots, S_n\}] - parent^*[\{S\}])$$

The *chart invariant* is defined by recursively visiting all attached invariants, starting with that attached to *Root*. In case a state is an XOR state, the invariant attached to some child has to hold as well. In case the state is an AND state, the invariant attached to each child has to hold as well.

$$\begin{array}{lcl}
chartinv & \widehat{=} & scopeinv(Root) \\
scopeinv(s) & \widehat{=} & inv(s) \wedge childinv(s) \\
childinv(s) & \widehat{=} & \mathsf{case}\; s\; \mathsf{of} \\
 & & \quad Basic : \mathsf{true} \\
 & & \quad XOR : \bigvee c \in children[\{s\}] \,.\, test(c) \wedge scopeinv(c) \\
 & & \quad AND : \bigwedge c \in children[\{s\}] \,.\, scopeinv(c) \\
 & & \mathsf{end}
\end{array}$$

Chart $\mathcal{S}$ is *correct* if the default transition of *Root* establishes the chart invariant and all operations of global events preserve the chart invariant:

$$\begin{array}{ll}
(a) & \{\mathsf{true}\}\, default(Root)\, \{chartinv\} \\
(b) & \forall E \in Global \,.\, \{chartinv\}\, op(E)\, \{chartinv\}
\end{array}$$

For the TV example, we define $Global = \{power, warm, down, up, mute\}$, which makes *soundOn* the only local event, and have:

$$\begin{array}{lcl}
inv(Working) & \equiv & test(Displaing) \Rightarrow \neg test(Waiting) \\
inv(Sound) & \equiv & 1 \leq lev \wedge lev \leq 10
\end{array}$$

For all other states, including *Root*, the attached invariant is true. It follows that $scopeinv(s)$ for all Basic states s of the chart is true; for the other states we get:

$$\begin{array}{lcl}
scopeinv(Picture) & \equiv & test(WarmingUp) \vee test(Displaying) \\
scopeinv(Sound) & \equiv & inv(Sound) \wedge (test(Waiting) \vee test(On) \vee test(Off)) \\
scopeinv(Working) & \equiv & inv(Working) \wedge scopeinv(Picture) \wedge scopeinv(Sound) \\
scopeinv(Root) & \equiv & test(Standby) \vee (test(Working) \wedge scopeinv(Working))
\end{array}$$

The last line defines the chart invariant. The B Method allows this invariant to be expressed in the INVARIANT section:

INVARIANT

$$
\begin{array}{l}
(root\ = Standby) \lor \\
(root\ = Working \land \\
\quad (picture\ = Displaying \Rightarrow \neg(sound\ = Waiting)) \land \\
\quad (1 \leq lev \land lev \leq 10))
\end{array}
$$

This leads to six correctness conditions, one for each event *power*, *warm*, *down*, *up*, *mute* and one for the initialization. The B tools generate these five conditions and allow them to be proven automatically or interactively.

Above invariant has been simplified. The definition of *chartinv* would generate predicates like $picture\ = WarmingUp \lor picture\ = Displaying$ that arise from the XOR case in $childinv(Picture)$. Such tautologies can be eliminated during generation with following reformulation:

$$
\begin{array}{lll}
childinv(s) & \equiv & \mathsf{case}\, s\, \mathsf{of} \\
 & & \quad Basic : \mathsf{true} \\
 & & \quad XOR : \bigwedge c \in children[\{s\}] \,.\, test(c) \Rightarrow scopeinv(c) \\
 & & \quad AND : \bigwedge c \in children[\{s\}] \,.\, scopeinv(c) \\
 & & \mathsf{end}
\end{array}
$$

Now, if $scopeinv(c)$ is true (which it is for every Basic state c without attached invariant), $test(c) \Rightarrow scopeinv(c)$ is immediately true. If this is the case for all children c of s, $childinv(s)$ is immediately true.

1.6 ACCUMULATED INVARIANTS

The observation underlying a more targeted verification condition generation is that sometimes it is sufficient to consider correctness of individual transitions, rather than that of an event operation, and that parts of the chart invariant may be irrelevant for the correctness of transitions. To start with, let the *base* of a state set *ss* be *ss* together with the implicit children of all ancestors of *ss*. That is, the base of *ss* adds to *ss* all children of AND ancestors that are not ancestors of *ss*, i.e. the "AND uncles". The *(upward) closure* of state set *ss* is the set of all ancestors of the base of *ss*, including *ss*. That is, it is the set of states in which a chart must be if it is in *ss*.

$$
\begin{array}{lll}
base(ss) & \widehat{=} & ss \cup imp(parent^{+}[ss]) \\
closure(ss) & \widehat{=} & parent^{*}[base(ss)]
\end{array}
$$

If a chart moves to state set *ss* then (1) it has to be in all ancestors of *ss*, (2) the attached invariants of all states of the closure of *ss* have to hold, and (3) the child invariants for all states of the base of *ss*, have to hold. The invariant constructed in this way is called the *accumulated invariant* of *ss*.

$$
\begin{array}{lll}
accinv(ss) & \widehat{=} & in(parent^{*}[ss]) \land \\
 & & (\bigwedge s \in closure(ss) \,.\, inv(s)) \land \\
 & & (\bigwedge s \in base(ss) \,.\, childinv(s))
\end{array}
$$

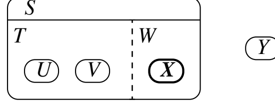


Figure 1.10 State hierarchy.

The invariants that originate from the descendants of the base are the *synthesized* invariants; those that originate from ancestors of the base are the *inherited* invariants. For example, in Figure 1.10 we have:

$$\begin{array}{lcl} base(\{X\}) & = & \{T, X\} \\ closure(\{X\}) & = & \{Root, S, T, W, X\} \\ accinv(\{X\}) & = & test(S) \wedge test(X) \wedge \\ & & inv(Root) \wedge inv(S) \wedge inv(T) \wedge inv(W) \wedge inv(X) \wedge \\ & & ((test(U) \wedge inv(U)) \vee (test(V) \wedge inv(V))) \end{array}$$

That is, the invariants of $Root$, S, T are inherited in S and the invariants of U, V are synthesized for X. Following property justifies accumulation: if a chart is in state set ss, then the chart invariant reduces to the accumulated invariant of ss.

Theorem 1 *For any non-empty state set ss:*

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

Rather than proving this theorem directly, we prove a more general one, but first state a lemma about how the accumulated invariant of a state relates to the accumulated invariant of its parent.

Lemma 1 *For any state s except Root:*

$$accinv(\{parent(s)\}) \wedge test(s) \quad \equiv \quad accinv(\{s\}) \quad \textit{if} \quad parent(s) \in XOR \qquad \text{(a)}$$

$$accinv(\{parent(s)\}) \quad \equiv \quad accinv(\{s\}) \quad \textit{if} \quad parent(s) \in AND \qquad \text{(b)}$$

We omit the proof. The following theorem states how the accumulated invariant of a state relates to the accumulated invariant of a set of descendants.

Theorem 2 *For any state s and any non-empty state set ss with* $ss \subseteq children^*[\{s\}]$*:*

$$accinv(\{s\}) \wedge in(path(s, ss)) \quad \equiv \quad accinv(ss)$$

Proof: The proof proceeds by induction over the maximal distance between s and ss, under the assumption that $ss \subseteq children^*[\{s\}]$. Let r^n be relation r composed n times, formally $r^0[p] = p$ and $r^{n+1}[p] = r[r^n[p]]$. Defining

$$p(s, ss) \quad \widehat{=} \quad accinv(\{s\}) \wedge in(path(s, ss)) \equiv accinv(ss)$$

we show that $p(s, ss)$ holds for $s \in \bigcup i \in [0..n] \,.\, parent^i[ss]$ by induction over n. In the base case, $n = 0$ implies $ss = \{s\}$, hence $p(s, ss)$ follows immediately. For the

induction step, suppose $p(s, ss)$ holds for all $s \in \bigcup i \in [0..n] \ . \ parent^i[ss]$. We show that $p(parent(s), ss)$ holds:

$$\begin{array}{ll} & accinv(\{parent(s)\}) \wedge in(path(parent(s), ss)) \equiv accinv(ss) \\ \equiv & \quad \langle \text{from the definitions of } in \text{ and } path \rangle \\ & accinv(\{parent(s)\}) \wedge in(s) \wedge in(path(s, ss)) \equiv accinv(ss) \\ \equiv & \quad \langle \text{case } parent(s) \in XOR \text{ and Lemma 1 (a), case } parent(s) \in AND \text{ and (b)} \rangle \\ & accinv(\{s\}) \wedge in(path(s, ss)) \equiv accinv(ss) \end{array}$$

Hence $p(s, ss)$ holds for $s \in parent[parent^n[ss]] = parent^{n+1}[ss]$. With the induction assumption it follows that $p(s, ss)$ holds for $s \in \bigcup i \in [0..n+1] \ . \ parent^i[ss]$, which completes the induction step and allows to conclude that $p(s, ss)$ holds for $s \in \bigcup i \in nat \ . \ parent^i[ss]$. The theorem follow by noting that $parent^*[ss] = \bigcup i \in nat \ . \ parent^i[ss]$ and that $s \in parent^*[ss]$ follows from the assumption $ss \subseteq children^*[\{s\}]$. ■

Theorem 1 follows from Theorem 2 by taking $s = Root$ and observing that $chartinv \equiv accinv(Root)$.

For the TV chart we note that for example

$$\begin{array}{ll} base(Standby) = \{Standby\} & closure(Standby) = \{Root, Standby\} \\ base(Working) = \{Working\} & closure(Working) = \{Working, Standby\} \\ base(On) = \{Picture, On\} & closure(On) = \{Root, Working, Picture, Sound, On\} \\ base(Off) = \{Picture, Off\} & closure(Off) = \{Root, Working, Picture, Sound, Off\} \end{array}$$

and get following accumulated invariants:

$$\begin{array}{lll} accinv(\{Standby\}) & \equiv & test(Standby) \\ accinv(\{Working\}) & \equiv & test(Working) \wedge (test(Displaying) \Rightarrow \neg test(Waiting)) \wedge \\ & & 1 \leq lev \wedge lev \leq 10 \\ accinv(\{On\}) & \equiv & test(Working) \wedge test(On) \wedge \\ & & (test(Displaying) \Rightarrow \neg test(Waiting)) \wedge 1 \leq lev \wedge lev \leq 10 \\ accinv(\{Off\}) & \equiv & test(Working) \wedge test(Off) \wedge \\ & & (test(Displaying) \Rightarrow \neg test(Waiting)) \wedge 1 \leq lev \wedge lev \leq 10 \end{array}$$

1.7 VERIFICATION CONDITION GENERATION

The *source invariant* of a transition is the accumulated invariant of its source states. The *target invariant* of transition t is consists of the accumulated invariant of its target states; if target states are composite states or if states are implicitly entered by t, then the accumulated invariant of the targets of the completion of t have to be added:

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

We are now prepared to present an alternative way of checking the correctness of a chart. The idea is to visit all transitions, starting those that the have the root state as

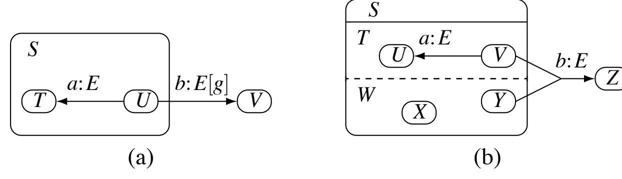


Figure 1.11 Transitions with different priorities.

their scope, and then to descend to all children. The correctness condition of transition t is in the simplest case:

$$\{sourceinv(t) \wedge guard(t)\}\ effect(t)\ \{targetinv(t)\}$$

In two cases this correctness assertion is not adequate. In the case that t is taken simultaneously with other transitions, other target invariants have to be established and other source invariants can be assumed. In the case that an ancestor of $scope(t)$ has other transitions on $event(t)$, these transitions have priority. In the recursive definition below, the conjunction of the negations of all triggers on E of one scope, expressed as $\bigwedge t \in Trans(E,s)\ .\ \neg trigger(t)$, is "assumed" when visiting the children:

$$\begin{array}{lcl}
correct(E) & \widehat{=} & scopecorrect(E, Root) \\
scopecorrect(E,s) & \widehat{=} & (\bigwedge t \in Trans(E,s)\ . \\
 & & \quad \{sourceinv(t) \wedge guard(t)\}\ effect(t)\ \{targetinv(t)\}) \wedge \\
 & & ((\bigwedge t \in Trans(E,s)\ .\ \neg trigger(t)) \Rightarrow childcorrect(E,s)) \\
childcorrect(E,s) & \widehat{=} & \mathsf{case}\ s\ \mathsf{of} \\
 & & \quad Basic : \mathsf{true} \\
 & & \quad XOR : \bigwedge c \in children[\{s\}]\ .\ scopecorr(E,c) \\
 & & \quad AND : \{accinv(\{s\})\}\ childop(E,s)\ \{accinv(\{s\})\} \\
 & & \mathsf{end}
\end{array}$$

Figure 1.11 illustrates the consequence of priorities on preconditions. We note that $\neg trigger(t) \equiv \neg in(exited(t)) \vee \neg guard(t)$. In (a), transition b has priority over a, hence a is taken only if g does not hold as $\neg guard(b)$ is part of the precondition of the correctness assertion for a. In general, for any predicates q,p,r and statement Q:

$$g \Rightarrow \{p\}\ Q\ \{r\} \quad \equiv \quad \{g \wedge p\}\ Q\ \{r\}$$

In (b), transition a is taken only if T is in V and W is not in Y as $\neg in(exited(b))$ is part of the precondition of the correctness assertion for a.

Theorem 3 *For any $E \in Global$:*

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

Rather than proving this theorem directly, we prove a more general one: if we consider only transitions at scope s or below, then $\{accinv(\{s\})\}\ scopeop(E,s)\ \{accinv(\{s\})\}$ and $scopecorrect(E,s)$ are equivalent:

Theorem 4 *For any state s:*

$$\{accinv(\{s\})\}\, scopeop(E,s)\, \{accinv(\{s\})\} \quad \equiv \quad scopecorrect(E,s)$$

Proof : The proof proceeds by induction over the structure of charts. Defining

$$p(s) \quad \widehat{=} \quad \{accinv(\{s\})\}\, scopeop(E,s)\, \{accinv(\{s\})\} \equiv scopecorrect(E,s)$$

the base case is that $p(s)$ holds for Basic or AND state s and the induction step is that $p(s)$ holds for XOR state s provided that $p(c)$ holds for all children c of s. To start with, we assume $\bigwedge a \in parent^{+}[\{s\}] \,.\, \bigwedge t \in Trans(E,a) \,.\, \neg trigger(t)$ and simplify:

$$\begin{array}{ll}
 & p(s) \\
\equiv & \quad \langle \text{definition of } scopeop, /\!/ \rangle \\
 & \{accinv(\{s\})\} \\
 & \quad ([\!]\ t \in Trans(E,s) \,.\, trigger(t) \rightarrow effect(t))\ [\!] \\
 & \quad ((\bigwedge t \in Trans(E,s) \,.\, \neg trigger(t)) \rightarrow childop(E,s)) \\
 & \{accinv(\{s\})\} \\
 & \equiv \\
 & scopecorrect(E,s) \\
\equiv & \quad \langle \text{verification rules for } [\!], \rightarrow, \text{definition of } trigger, scopecorrect \rangle \\
 & (\bigwedge t \in Trans(E,s) \,. \\
 & \quad \{accinv(\{s\}) \wedge in(exited(t)) \wedge guard(t)\}\, effect(t)\, \{accinv(\{s\})\}) \wedge \\
 & ((\bigwedge t \in Trans(E,s) \,.\, \neg trigger(t)) \Rightarrow \\
 & \quad \{accinv(\{s\})\}\, childop(E,s)\, \{accinv(\{s\})\} \\
 & \equiv \\
 & (\bigwedge t \in Trans(E,s) \,. \\
 & \quad \{sourceinv(t) \wedge guard(t)\}\, effect(t)\, \{targetinv(t)\}) \wedge \\
 & ((\bigwedge t \in Trans(E,s) \,.\, \neg trigger(t)) \Rightarrow childcorrect(E,s)) \\
\Leftarrow & \quad \langle \text{by Theorem 2: } accinv(\{s\}) \wedge in(exited(t)) \equiv sourceinv(t), (*) \rangle \\
 & ((\bigwedge t \in Trans(E,s) \,.\, \neg trigger(t)) \Rightarrow \\
 & \quad \{accinv(\{s\})\}\, childop(E,s)\, \{accinv(\{s\})\} \\
 & \equiv \\
 & ((\bigwedge t \in Trans(E,s) \,.\, \neg trigger(t)) \Rightarrow childcorrect(E,s)) \\
\Leftarrow & \quad \langle \text{logic} \rangle \\
 & \{accinv(\{s\})\}\, childop(E,s)\, \{accinv(\{s\})\} \ \equiv\ childcorrect(E,s)
\end{array}$$

In the step (*) we use that $effect(t)$ does indeed establish $\bigwedge u \in comp(t) \,.\, in(entered(u))$, which is given by the definition of $effect(t)$, and does preserve $accinv(\{s\})$, which is guaranteed by well-formedness condition 3. Hence $\bigwedge u \in comp(t) \,.\, in(entered(u))$ can be conjoined to the postcondition $accinv(\{s\})$. It is then straightforward to show that by Theorem 2 and the definition of $comp(t)$:

$$accinv(\{s\}) \wedge (\bigwedge u \in comp(t) \,.\, in(entered(u))) \quad \equiv \quad targetinv(t)$$

We continue the proof with a case analysis. If $s \in Basic$, $childop(E,s)$ simplifies to skip and $childcorrect(E,s)$ simplifies to true, hence the last line follows immediately. If $s \in$

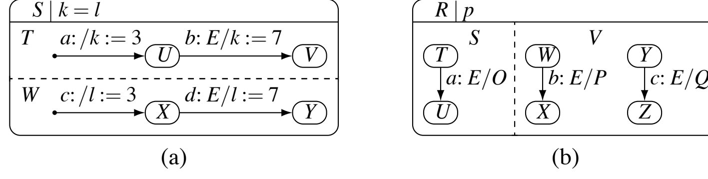


Figure 1.12 Concurrent transitions and invariants

AND, $childcorrect(E,s)$ is equivalent to $\{accinv(\{s\})\}\, childop(E,s)\, \{accinv(\{s\})\}$, hence the last line follows immediately. If $s \in XOR$, we continue:

$$
\begin{array}{ll}
 & \{accinv(\{s\})\}\ [\!]\ c \in children[\{s\}] \,.\, test(c) \rightarrow scopeop(E,c)\, \{accinv(\{s\})\} \\
 & \equiv \\
 & (\bigwedge c \in children[\{s\}] \,.\, scopecorr(E,c)) \\
\Leftarrow & \quad \langle \text{verification rules for } [\!], \rightarrow \rangle \\
 & (\bigwedge c \in children[\{s\}] \,.\, \{accinv(\{s\}) \wedge test(c)\}\, scopeop(E,c)\, \{accinv(\{s\})\}) \\
 & \equiv \\
 & (\bigwedge c \in children[\{s\}] \,.\, scopecorr(E,c)) \\
\Leftarrow & \quad \langle \text{logic, by Theorem 2: } accinv(\{s\}) \wedge test(c) \equiv accinv(\{c\}), (^{**}) \rangle \\
 & \bigwedge c \in children[\{s\}] \,. \\
 & \quad \{accinv(\{c\})\}\, scopeop(E,c)\, \{accinv(\{c\})\}) \ \equiv\ scopecorr(E,c)
\end{array}
$$

In the step (**) we use that $scopeop(E,c)$ preserves $accinv(\{c\})$, which is guaranteed by well-formedness condition 3. The last line is exactly the induction assumption. This concludes the induction step and the case analysis. ■

Theorem 3 follows by taking $s = Root$ and observing that $chartinv \equiv accinv(Root)$.

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

$$\{accinv(\{s\})\} \parallel c \in children[\{s\}] \,.\, scopeop(E,c)\, \{accinv(\{s\})\} \qquad (*)$$

In general $\{p\}\, Q \parallel R\, \{r\}$ cannot be split into one condition for Q and one for R, as can be seen for $\{k = l\}\, k := 7 \parallel l := 7\, \{k = l\}$. Figure 1.12 (a) illustrates that the correctness of b and d cannot be shown individually.

For the TV chart we have that

$$
\begin{array}{lcl}
op(soundOn) & = & (test(Working) \rightarrow \\
 & & \quad (test(Waiting) \rightarrow assign(On) \\
 & & \quad [\!]\ test(On) \rightarrow \mathsf{skip} \\
 & & \quad [\!]\ test(Off) \rightarrow \mathsf{skip}) \\
 & & [\!]\ test(Standby) \rightarrow \mathsf{skip})
\end{array}
$$

and get following correctness assertions, with some simplifications:

$$\begin{array}{lll}
correct(power) & \equiv & \{accinv(\{Standby\})\} \\
& & \quad assign(Working) \parallel assign(WarmingUp) \parallel \\
& & \quad assign(Waiting) \parallel lev := 5 \\
& & \{accinv(\{Working\})\} \\
& & \wedge \\
& & \{accinv(\{Working\})\} \\
& & \quad assign(Standby) \\
& & \{accinv(\{Standby\})\} \\
correct(warm) & \equiv & \{accinv(\{Working\})\} \\
& & \quad test(WarmingUp) \rightarrow op(soundOn) \parallel assign(Displaying) \\
& & \{accinv(\{Working\})\} \\
correct(down) & \equiv & \{accinv(\{Working\})\} \\
& & \quad test(On) \rightarrow lev := lev - 1 \parallel assign(On) \\
& & \{accinv(\{Working\})\} \\
correct(up) & \equiv & \{accinv(\{Working\})\} \\
& & \quad test(On) \rightarrow lev := lev + 1 \parallel assign(On) \\
& & \{accinv(\{Working\})\} \\
correct(mute) & \equiv & \{accinv(\{Working\})\} \\
& & \quad (\ test(On) \rightarrow assign(Off) \\
& & \quad [\!]\ test(Off) \rightarrow assign(On)) \\
& & \{accinv(\{Working\})\}
\end{array}$$

The simplifications carried out are that verification conditions of the form $\{p\}\,Q \,/\!/\, \mathsf{skip}$ $\{p\}$ are replaced by $\{p\}\,Q\,\{p\}$.

In the design of embedded systems, physical components are typically modelled by concurrent states on outer levels. For such designs, the possibility for generating targeted verification conditions by *scopecorrect* is limited, as the recursion stops as soon as an AND state is encountered. Still, special cases exist:

1. If only one concurrent state contains transitions on event E, then the parallel composition in (*) disappears, resulting in

   $$\{accinv(\{s\})\}\, scopeop(E,s)\, \{accinv(\{s\})\}$$

   Theorem 4 can now be used to continue decomposing the verification conditions according to *scopecorrect*.

2. Further splitting of the verification condition is possible according to the structure of $scopeop(E,c)$. If an operand of the parallel composition contains a nondeterministic choice with guards, we can use that $\parallel$ distributes over $[\!]$:

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

   Figure 1.12 (b) illustrates such a case: the operation of E in V contains a choice over all children of V. Applying above rule results in two verification

conditions, one with a parallel composition of a and b and one with a and c. In general, if there are m concurrent states and each has n transitions on event E, then this results in $m \times n$ verification conditions. Hence this approach has the potential of generating a possibly large number of smaller conditions.

3. The distributivity of $\|$ over $[]$ can also be applied is for bodies containing conditional statements, as if g then Q else $R = (g \rightarrow Q) \;[]\; (\neg g \rightarrow R)$. Hence, for each transition the number of proof conditions involving that transition double with each conditional statement that it contains.

For the TV example we note transitions on *warm*, *down*, *up*, *mute* occur only in one concurrent state and apply rule 1 above. As *warm* broadcasts *soundOn*, we apply rule 2 as well.

$$
\begin{array}{lll}
correct(warm) & \equiv & \{accinv(\{WarmingUp\})\} \\
& & \quad test(WarmingUp) \rightarrow test(Working) \rightarrow \\
& & \quad\quad test(Waiting) \rightarrow assign(On) \parallel assign(Displaying) \\
& & \{accinv(\{Displaying\})\} \\
& & \wedge \\
& & \{accinv(\{WarmingUp\})\} \\
& & \quad test(WarmingUp) \rightarrow test(Working) \rightarrow test(On) \rightarrow \\
& & \quad\quad assign(Displaying) \\
& & \{accinv(\{Displaying\})\} \\
& & \wedge \\
& & \{accinv(\{WarmingUp\})\} \\
& & \quad test(WarmingUp) \rightarrow test(Working) \rightarrow test(Off) \rightarrow \\
& & \quad\quad assign(Displaying) \\
& & \{accinv(\{Displaying\})\} \\
correct(down) & \equiv & \{accinv(\{On\})\}\; lev := lev - 1 \parallel assign(On)\; \{accinv(\{On\})\} \\
correct(up) & \equiv & \{accinv(\{On\})\}\; lev := lev + 1 \parallel assign(On)\; \{accinv(\{On\})\} \\
correct(mute) & \equiv & \{accinv(\{On\})\}\; assign(Off)\; \{accinv(\{Off\})\} \\
& & \wedge \\
& & \{accinv(\{Off\})\}\; assign(On))\; \{accinv(\{On\})\}
\end{array}
$$

The two verification conditions for *power* are unchanged. Thus this results in nine verification conditions, compared to the original five, plus one for the initialization.

The proof conditions are now of the from $\{p\}\, Q_1 \| \ldots \| Q_n\, \{r\}$, where each Q_i is a multiple assignment statement, assigning to state variables or to global variables. Using that $(x := e \parallel y := f) = (x, y := e, f)$ these can be merged into a single multiple assignment. The verification rule for assignments yields then a plain predicate that can be passed to a theorem prover.

1.8 PRIORITY AMONG TRANSITIONS

UML statecharts differ from above interpretation in giving transitions with inner scope priority over transitions with outer scope [18]. Thus in Figure 1.13 (a) transition a has

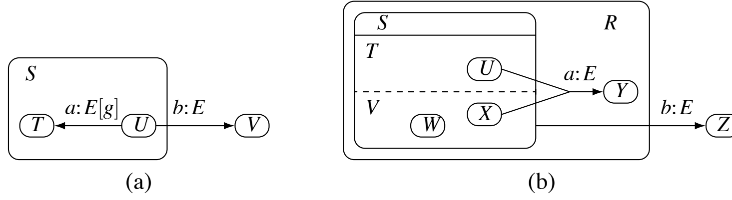


Figure 1.13 Transitions with different priorities.

priority over transition b as $\neg guard(a)$ is part of the precondition of the correctness assertion for b. In (b), transition b is taken only if T is not in U or V is not in X as $\neg in(exited(a))$ is part of the precondition of the correctness assertion for b. In this interpretation, the notion of a chart invariant remains the same, but *op* and *correct* have to be adapted. Let $\underline{Trans}(E,s)$ be the set of all transition on E with scope below s:

$$\underline{Trans}(E,s) \quad \widehat{=} \quad \{t \in Transition \mid event(t) = E \wedge scope(t) \in children^{+}[\{s\}]\}$$

The operation $\underline{op}(E)$ allows a transition to be taken only if no other transition with lower scope is enabled. Formally, all transitions on scope s are guarded by $\bigwedge t \in \underline{Trans}(E,s) \,.\, \neg trigger(t)$. The choice among transitions with the same scope is arbitrary.

$$\begin{array}{lcl}
\underline{op}(E) & \widehat{=} & \underline{scopeop}(E, Root) \\
\underline{scopeop}(E,s) & \widehat{=} & ((\bigwedge t \in \underline{Trans}(E,s) \,.\, \neg trigger(t)) \rightarrow \\
 & & \quad [\!]\; t \in Trans(E,s) \,.\, trigger(t) \rightarrow effect(t)) \\
 & & /\!/\, \underline{childop}(E,s) \\
\underline{childop}(E,s) & \widehat{=} & \mathsf{case}\; s\; \mathsf{of} \\
 & & \quad Basic : \mathsf{skip} \\
 & & \quad XOR : [\!]\; c \in children[\{s\}] \,.\, test(c) \rightarrow \underline{scopeop}(E,c) \\
 & & \quad AND : \|\; c \in children[\{s\}] \,.\, \underline{scopeop}(E,c) \\
 & & \mathsf{end}
\end{array}$$

The verification conditions reflect this by assuming that $\bigwedge t \in \underline{Trans}(E,s) \,.\, \neg trigger(t)$ holds for transitions with scope s:

$$\begin{array}{lcl}
\underline{correct}(E) & \widehat{=} & \underline{scopecorrect}(E, Root) \\
\underline{scopecorrect}(E,s) & \widehat{=} & ((\bigwedge t \in \underline{Trans}(E,s) \,.\, \neg trigger(t)) \Rightarrow \\
 & & \quad (\bigwedge t \in Trans(E,s) \,. \\
 & & \qquad \{sourceinv(t) \wedge guard(t)\}\; effect(t)\; \{targetinv(t)\})) \wedge \\
 & & \underline{childcorrect}(E,s)) \\
\underline{childcorrect}(E,s) & \widehat{=} & \mathsf{case}\; s\; \mathsf{of} \\
 & & \quad Basic : \mathsf{true} \\
 & & \quad XOR : \bigwedge c \in children[\{s\}] \,.\, \underline{scopecorr}(E,c) \\
 & & \quad AND : \{accinv(\{s\})\}\; \underline{childop}(E,s)\; \{accinv(\{s\})\} \\
 & & \mathsf{end}
\end{array}$$

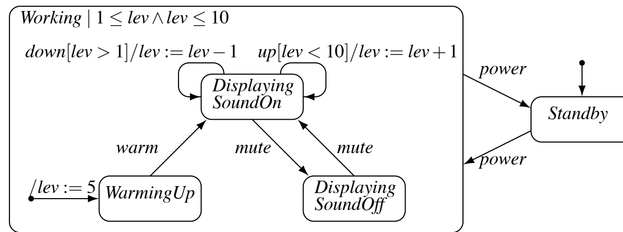


Figure 1.14 Alternative state structure for TV set.

Without proof we claim that the preservation of the chart invariant by $\underline{op}(E)$ can be verified by $\underline{correct}(E)$:

Theorem 5 *For any $E \in Global$:*

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

The verification conditions from $\underline{correct}(E)$ are of similar complexity as those from $correct(E)$. However, if $\underline{op}(E)$ were directly used for code generation, the resulting code would be more complex: as in the recursive definition transitions on outer scopes are visited first, the triggers of all transitions of lower scopes on that event need to be evaluated before these these are visited, where they are evaluated again.

1.9 CONCLUSIONS

Having an effective mechanism for verifying invariants begs the question of when and how to use invariants. Sometimes the need for an invariant can be avoided altogether. Figure 1.14 gives a chart that is equivalent to that of Figure 1.1 but avoids the invariant originally attached to *Working* by restructuring the states of *Working*. If one were not able to express and check invariants one might prefer the restructured one, on the grounds that by its mere structure it cannot lead to an invalid configuration. However, the structure of concurrent states of the original chart reflects the structure of the components of the application better and one would believe that it is easier to design, comprehend, and maintain. We could also avoid the invariant $1 \leq lev \wedge lev \leq 10$ by having ten distinct *On* states, one for each level. Such a design would be awkward at best and impossible if the range of variables is unbounded. In presence of global variables, invariants cannot be avoided through restructuring. After all, we get confidence in a design by having descriptions with some redundancy–here by state transitions and by invariants–and checking their consistency. By removing the possibility for these checks through a “clever” design, the design will not be more trustworthy.

We define two chart annotations to be *equivalent* if the resulting chart invariants are equivalent, meaning that they lead to the same proof conditions. Figure 1.15 illustrates two sets of equivalent chart annotations. Used as transformation rules,

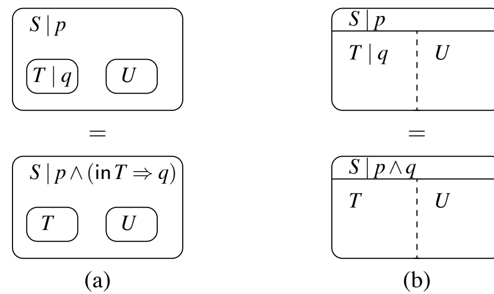


Figure 1.15 Equivalent annotations.

these two equivalencies allow all invariants to be moved up to *Root*. This design freedom leads to the question where to attach invariants best. Figure 1.16 gives an annotation that is equivalent to that of Figure 1.1. The original invariant of *Sound* is now attached to *Working*. However, only transitions within *Sound* are relevant for this invariant: the invariant is above the scope of all affected transitions. In Figure 1.16 the original invariant of *Working* has been moved to *Displaying*. While this shortens the invariant expression by leaving out the state tests, some relevant transitions are now in a concurrent state, making the dependency less visual. These two observations motivate following rule:

- Invariants should be attached exactly to the scope of all relevant transitions.

The chart of Figure 1.1 follows this rule. We summarize the main points of the approach:

1. Configurations are defined by state variables and each event is defined as one operation for all transitions on that event. This disallows Boolean combinations of events as in classical statecharts but is in line with UML statecharts.

2. An operation of an event is defined by a "recursive descent" of the state hierarchy. This favours giving priority to transitions on outer level over transitions on inner levels. This definition also serves as a scheme for code generation.

3. The state variables and event operations are mapped to one module (MACHINE in the B Method).

4. All transitions on an event are taken simultaneously, rather than in a sequence of micro-steps. For this, all simultaneously taken transitions must be conflict-free. In our experience that excludes some statecharts that would be of questionable design.

5. Invariants can be attached to basic and composite states. The chart invariant is derived from the attached invariants. All event operations have to preserve the chart invariant.

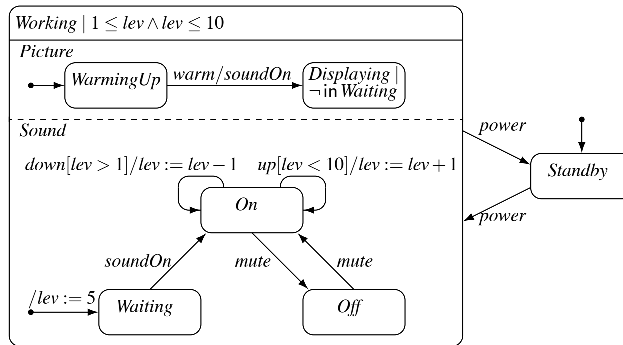


Figure 1.16 Alternative annotation for TV set.

6. The default transition of the root state has to establish the chart invariant; default transitions are also used for establishing a local invariant. For this, default transitions need to have a body.

7. Local verification conditions are computed from the accumulated invariant of the source and target states. The justification of the local verification conditions is in terms of the chart invariant.

An alternative to mapping the state variables and event operations to a single module is to distribute them by certain design criteria among several modules with an acyclic or tree dependency structure [10]. With invariants distributed among modules as well, this also leads to more local verification conditions, but in a different way than through accumulated invariants. Entry and exit actions, history states, and transitions with segments remain future work.

Acknowledgements

The author is indebted to Kevin Lano for his patience and his help. Dai Tri Man Le suggested the term accumulated invariant. Daniel Zingaro pointed out several errors.

REFERENCES

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

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

3. 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.

4. W. Damm, B. Jasko, H. Hungar, and A. Pnueli. A compositional real-time semantics of STATEMATE designs. In W.-P. de 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.

5. 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.

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

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

8. 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.

9. R. Laleau and A. Mammar. An overview of a method and its support tool for generating b specifications from uml notations. In *15th IEEE International Conference on Automated Software Engineering, ASE 2000*, Grenoble, France, 2000. IEEE Computer Socitey Press.

10. K. Lano, K. Androutsopoulos, and P. Kan. Structuring reactive systems in B AMN. In *3rd IEEE International Conference on Formal Engineering Methods*, York, England, 2000. IEEE Computer Socitey Press.

11. K. Lano and D. Clark. Direct semantics of extended state machines. *Journal of Object Technology*, 6(9):35–51, 2007.

12. 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.

13. H. Ledang and J. Souquières. Contributions for modelling UML state-charts in B. In *Integrated Formal Methods, Third International Conference, IFM 2002*, Lecture Notes in Computer Science 2335, pages 109–127, Turku, Finland, May 2002. Springer.

14. J. Lilius and Ivan 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.

15. E. Meyer and J. Souquières. A systematic approach to transform OMT diagrams to a B specification. In J. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems, FM'99*, Lecture Notes in Computer Science 1708, pages 875—895, Toulouse, France, September 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. H. P. Nguyen. *Dérivation De Spécifications Formelles B à Partir De Spécifications Semi-Formelles*. Doctoral thesis, INIST-CNRS, 1998.

18. OMG. Unified modeling language, superstructure, v2.1.2. Technical report, OMG, http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF, 2007.

19. 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.

20. 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.

21. E. Sekerinski. Verifying statecharts with state invariants. In K. Breitman, J. Woodcock, R. Sterritt, and M. Hinchey, editors, *ICECCS '08—13th IEEE International Conference on Engineering of Complex Computer Systems*, pages 7–14, Belfast, Northern Ireland, March 2008. IEEE Computer Society.

22. 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.

23. 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.

24. D. Snook and M. Butler. UML-B: Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 15(1):92–122, January 2006.