The Semantics of Statecharts in HOL

Nancy Day and Jeffrey J. Joyce

Integrated Systems Design Laboratory, Department of Computer Science,
University of British Columbia, Vancouver, B.C., V6T 1Z2, Canada

Abstract. Statecharts are used to produce operational specifications in the CASE tool STATEMATE. This tool provides some analysis capabilities such as reachability of states, but formal methods offer the potential of linking more powerful requirements analysis with CASE tools. To provide this link, it is necessary to have a rigorous semantics for the specification notation. In this paper we present an operational semantics for statecharts in quantifier free higher order logic, embedded in the theorem prover HOL.

1 Introduction

Statecharts are an extended finite state machine, graphical formalism for real-time systems which alleviates many of the problems such as state explosion, encountered with other state machine notations [4]. They are the notation used to give operational specifications in the commercial CASE tool STATEMATE. This tool provides some analysis capabilities such as reachability of states, but formal methods offer the potential of linking more powerful requirements analysis with CASE tools.

To use these methods, it is necessary to have a rigorous semantics for the notation. Previous work has given semantic interpretations for statecharts[2][7][8], but these did not completely agree with our intuitive idea of their behaviour. In this paper we present an operational semantics for statecharts in quantifier free higher order logic, embedded in the theorem prover HOL. The type-checking facilities of HOL and the expressiveness of higher order logic were very useful in writing the semantics.

The first section informally describes the operation of statecharts and the remaining parts of the paper formalize these ideas, pointing out situations where it is not obvious what the behaviour of the statechart will be. Particular attention is given to issues such as what it means to take a step, race conditions, and multiple actions associated with one transition. The semantics are given by a next configuration predicate which holds true if one complete system configuration is a successor to another.

These semantics have been used to create a model checker for statecharts in the hybrid verification tool HOL-VOSS[9].

2 An Introduction to Statecharts

There is a great deal of interest from both academia and industry in the statecharts formalism. It is an extended state transition notation for expressing the concurrent operation of real-time systems. It is often described as:

state-diagrams + depth + orthogonality + broadcast-communication[2]

In statecharts, the diagrammatic layout of the notation has meaning beyond just the labels on states and transitions. A hierarchy of states is portrayed in a style similar to set inclusion in Venn diagrams to reduce the complexity of the model and therefore make it more readable. In light of this, we rely on the graphical notation to introduce statecharts through an example. The reader is referred to Harel[3] for an explanation of the origins of statecharts as a type of higraph that combines the elements of graphs and Venn diagrams.

The STATEMATE manual describes a traffic light system controlling a two-way intersection which is a simple but effective example of the expressiveness of statecharts[5]. The statechart for this controller is given in Fig.1 and will be referred to throughout this section.

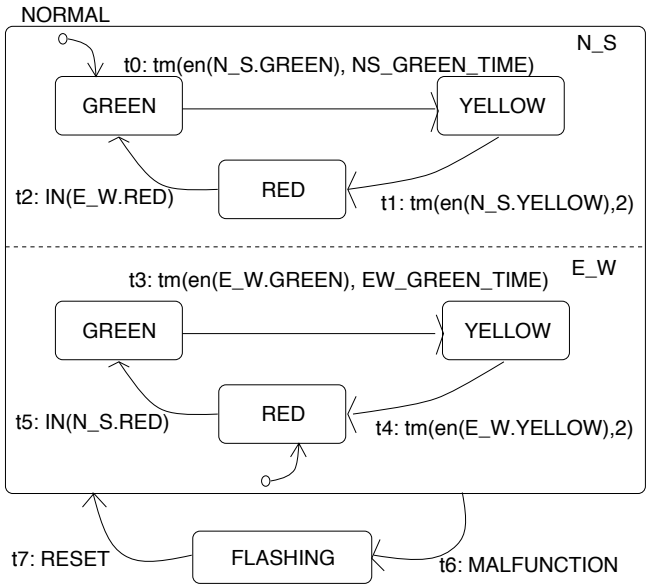

Fig. 1. Traffic light statechart

A statechart models the system as being in a number of *states*, depicted by rounded boxes, which describe its operation. For example, the state labelled NORMAL, at the top of the page, represents the normal operation of the lights in both directions. The dashed line through its middle splits it into two substates,

north-south(N_S) and east-west(E_W), which operate concurrently (orthogonality), making NORMAL an AND-state. N_S is an OR-state since it has substates, labelled red, yellow, and green, and the model can only be in one of them at any time (exclusive OR). The representation of these substates within the larger rounded box creates a hierarchy of states (depth). Within this hierarchy, the state NORMAL is an *ancestor* of N_S and E_W. Similarly, N_S.GREEN is a *descendant* of N_S. When a state is not decomposed into AND- or OR-states, it is called a basic state.

At a given moment, the *configuration* of the model includes the states the system is currently in and the values for all data-items. The current set of states alone is called the *state configuration*. A set of basic states is a *legal state configuration* if it satisfies the constraints of the hierarchy. A discrete notion of time is used where the system moves between configurations as a result of stimulus generated both from within the system and externally.

States are connected by *transitions* with labels of the form *event* [*condition*] / *action*. The tags **t0**, **t1**, etc. are used for reference only. If the system is currently in the source state of a certain transition, labelled *e[c]/a*, and the event e occurs when the condition c is satisfied then the transition is *enabled*. *Broadcast communication* is used which means that all events and the values of any data-items can be referenced anywhere in the system. The event and condition are together referred to as the *trigger* of the transition.

A *condition* is a boolean expression which can include statements like IN(x) to check whether the system is currently in state x. These are often used to synchronize components as in transition **t5** in the E_W state.

An *event* is a change in a condition which occurs in the previous step. Entering a state x causes the event **en**(x) to occur. A timeout event, **tm**(ev, x), occurs x steps after the event ev. The event ev is called the *timeout event* and x is the *timeout step number*. The configuration of the system includes the relevant events which occur in the previous step.

Enabled transitions move the system from one configuration to another. Following, or *taking* a transition means exiting its source state, carrying out the actions on its label, and entering its destination state. Informally, following a set of these transitions constitutes a *step* or one time unit.

Transitions can be taken in the substates of an AND-state simultaneously. A transition can be enabled if it originates in any ancestor of the current set of basic states in the configuration. Transitions can also terminate at the outer boundary of a state with substates. *Default arrows*, given diagrammatically as open circles pointing at a state, lead the system into a configuration of basic states. For example, when transition **t7** is followed, it terminates at the state NORMAL which is made up of parallel components. The default arrows for each of its substates point at E_W.GREEN and N_S.RED.

If a transition is followed, the action part of the label is carried out and the system moves into the destination state. Actions include generating events or modifying values of variables through assignment statements.

Statecharts often include elements like history states, connectors for compound transitions, static reactions, and transitions with multiple source and destination states. For simplicity, these are not considered here but they are discussed in [1].

3 Ambiguities in Statecharts

The notation described above may seem very straightforward, however, statecharts can be created where their intended meaning is not so obvious.

3.1 What is a Step?

There is no inherent model of timing associated with statecharts other than the movement between states by following transitions. Following a set of transitions and carrying out their actions is considered one time unit or *step*. There are different interpretations of what constitutes this set of transitions.

Briefly, the factors to consider are:

- How are conflicts among enabled transitions resolved ? (i.e. when they can not all be taken)
- Can events generated by the actions of transitions followed in this step trigger transitions which are also followed in this step? In Fig.2a, if the system starts in the states **A** and **C**, and transition **t0** is followed generating the event **f**, is **t1** then enabled and followed in the same step?
- are transitions only from the current set of source states considered or can we move through multiple states in a path in one step? From Fig.2b, we can see that this could lead to infinite loops within a step[5].

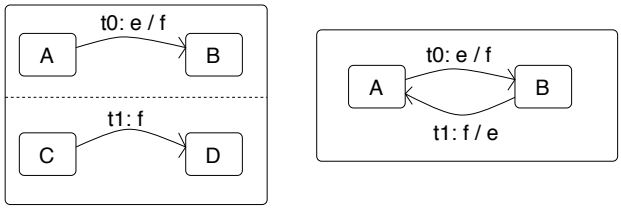

Fig. 2. What is a step?

3.2 Multiple Actions on a Transition

A transition may have multiple actions. If more than one of these actions modify the same variable, what will the value of the variable be at the end of the transition? For example, the action $/x := 1; x := x + 2$, with 0 as the value of x in the current configuration, has the following possible interpretations:

1. The actions are taken sequentially so that after following the transition x has the value 3.
2. The actions are taken relative to the beginning of the step but their effects are evaluated sequentially, therefore the second action takes precedence and the result is that x becomes 2.
3. The actions are evaluated relative to the beginning of the step, and they are not assumed to happen in any particular order, however, the actions are atomic and do not conflict. With this interpretation, the result is that x could be 1 or 2 after the transition is taken.

3.3 Race Conditions

A race condition occurs if transitions followed simultaneously in parallel components modify the same variable in a step. This is similar to the situation described in the previous section, but has the added possible interpretation that the actions could conflict with each other (i.e. they are not atomic), and the value for x would then be indeterminate.

3.4 Non-determinism

Statecharts have a hierarchy of states and transitions can originate from states at any level in the hierarchy. If multiple transitions are enabled from states which are descendants or ancestors of each other in the hierarchy, as in Fig.3 where **A** is the parent of **B**, which transition is taken or should both be followed?

Transitions may also originate at exactly the same state and if both are enabled either could be followed.

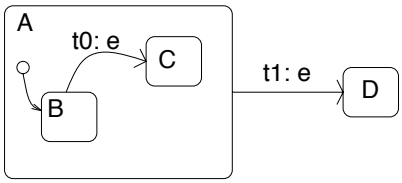

Fig. 3. Structural non-determinism

3.5 Timeouts

The statechart for the traffic light in Fig.1 uses several timeouts to trigger different transitions, such as **t0** or **t1** . When should the system begin to consider the event upon which the timeout is based? Is it the last time the timeout event occurs? Or must the timeout event occur after we have entered the source state and then the system waits the appropriate number of steps before following the transition?

When the timeout step number is symbolic, there is the further question of when to evaluate it. Is it evaluated when the system arrives in the transition's source state (i.e. the first time the transition could be enabled)? Or can the value change between steps? An example of a situation where this might occur in the traffic light is if the **NS_GREEN_TIME** is affected by a pedestrian button which indicates someone wants to cross the street.

3.6 Transitions Among AND Component States

The orthogonal components of AND-states operate concurrently, so it is difficult to see the need for transitions which go between them. However, it is possible, depending on the definition of a legal statechart. In Fig.4a, we can see that following transition **t0** leads into the state **C**, but the system must remain in some state of **X** at all times. At this point, should it follow the default transition of **X** into **A** to reach a legal state configuration?

The situation could occur where two transitions cross AND-state boundaries at the same time. In Fig.4b, if **t0** and **t3** are followed at the same time, the system will arrive in states **B** and **C**, which is a legal state configuration.

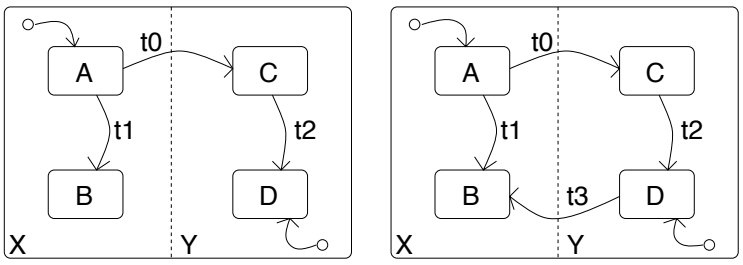

Fig. 4. Transitions among AND component states

4 Textual Representation of Statecharts

The graphical notation described in the previous section can be directly translated to a textual representation. A translation program extracts information about the statechart from STATEMATE and outputs ML code to create a HOL definition for the statechart. This definition can be given directly as an argument (referred to as the variable *sc*) to the semantic functions. The semantics use functions which extract information from the HOL definition.

5 A Semantics for Statecharts

Meaning is given to the syntax describing the statechart by translating it into a relation over the current configuration and next configuration. Other semantics for statecharts, given previously by Harel[2], Pnueli[8], and Leveson[7], differ from the semantics used here both in content and form. This paper presents only how we resolved the major difficulties in writing the semantics. The full semantics and a comparison to other approaches can be found in [1].

Our semantic functions were written with the intention of using them with a model checker, where they are executed in a Binary Decision Diagram(BDD) package which understands higher order logic with quantification only over Boolean variables. Limited quantification is not a major restriction since we are dealing with a finite domain. However, it is natural to express conditions over all transitions or all variables. In order to create the abstraction of existential and universal quantification over these sets, the semantics use definitions like EVERY, which takes the conjunction of applying a predicate p, to all elements of a list x:

$$\mathsf{EVERY}\, p\, x = (x = [\,]) \Rightarrow \mathsf{T} \mid p\,(\mathsf{HD}\, x) \wedge \mathsf{EVERY}\, p\,(\mathsf{TL}\ x)$$

Analogous definitions over lists can be given for:

EXISTS p x: return the disjunction of applying p to the elements of the list x
X_EXISTS p x: exactly one element of the list x returns true when p is applied to all elements in the list
PAIR_EVERY p x y: given two lists, x and y, return the conjunction of applying p to the pair made up of the first elements of each list

Modelling the configuration. The configuration of the system is completely represented by the values of a set of variables which include elements for the basic states, data-items, and events.

We can describe a configuration as a function mapping variables to values:

$$\text{Config} \equiv \text{Variable} \rightarrow \text{Value}$$

The variable cf will be used in functions to represent a configuration.

The meaning of expressions can be given compositionally using functions which take a configuration as an argument and return the result of evaluating the expression in that configuration. For example, the meaning of a variable v is given by the function: $\mathsf{SemVAR}\, v = \lambda cf.\, cf\ v$. This style of function has been used previously to give the semantics of a small imperative language where all of its elements are compositional[6]. Expressions are evaluated relative to the current configuration and actions assign the values of these expressions to the variables in the next configuration.

Hierarchy of states. Each basic state is represented by one Boolean variable which indicates whether or not the system is currently in that state. Higher level states (stn) are given meaning through the values of the basic states:[1]

$$\begin{aligned}\mathsf{INSTATE}\ sc\ cf\ stn = ((\mathsf{TYP}\ sc\ stn = \mathsf{B}) \wedge \mathsf{SemVAR}\ stn\ cf) \vee \\ ((\mathsf{TYP}\ sc\ stn = \mathsf{A}) \wedge \mathsf{EVERY}\,(\mathsf{INSTATE}\ sc\ cf)\,(\mathsf{SUBSTATES}\ sc\ stn)) \vee \\ ((\mathsf{TYP}\ sc\ stn = \mathsf{O}) \wedge \mathsf{EXISTS}\,(\mathsf{INSTATE}\ sc\ cf)\,(\mathsf{SUBSTATES}\ sc\ stn))\end{aligned}$$

The difficulty in giving the semantics is in expressing both which transitions can be taken and what the result is of following these transitions. The remaining semantic functions can be grouped into three areas:

- determining if events occur in a step (EVENT_COND)
- conditions on the set of transitions which can be taken, including hierarchy, priority and triggers (TRANS_COND)
- conditions on the variables in the next configuration modified in this step (VAR_COND)

The validity of these semantics, both in our interpretation of the operation of statecharts, and in the correctness of expressing this interpretation in higher order logic, has been checked using the theorem prover HOL by reducing the semantic functions to Boolean expressions over the variables for particular problems. Through this process, errors were discovered and fixed, and we have increased confidence in the result.

5.1 Events

Events are interpreted as Boolean expressions which depend on whether changes in values occur between steps. For events other than timeouts, this is relative to the previous step, but for timeouts, it can involve checking several time units earlier. In order to minimize the number of values used in the overall expression, we must determine the truth value of an event relative to the current configuration only. A counter for each relevant event is created which gets reset to zero when the event occurs and otherwise is incremented in each step. For a timeout, we test if the counter is equal to the timeout step number. This allows us to determine the truth value of events relative to the current configuration only.

Since each counter has a maximum value, we have to ensure that it does not falsely indicate that the event occurs when it overflows. This is done by incrementing it only up to its maximum value. This maximum value can never be used to indicate an event occurring, therefore a counter must be larger than its associated timeout step number.

To determine if a timeout event occurs, the function SemTM checks if the counter(c) for the event equals the expressions (e) for the timeout step number:

$$\mathsf{SemTM}\,(c,e) = \lambda cf.\,\neg\mathsf{MAXVALUE}\,(\mathsf{SemVAR}\ c\ cf) \wedge \mathsf{SemEQUAL}\,(c,e)\,cf$$

[1] States can have type A (AND), O(OR), or B(basic).

The timeout step number is evaluated in the current configuration resolving the questions raised in Sect.3.5.

The counter is reset to zero in the next configuration if the event occurs in this step. For example, to determine if the system enters a state (stn) in this step, we use the function:

$$\mathsf{SemEN}\ sc\ stn = \lambda(cf, cf').\neg\mathsf{INSTATE}\ sc\ cf\ stn \wedge \mathsf{INSTATE}\ sc\ cf'\ stn$$

The next configuration relation must include a condition which updates the events for each transition:

$$\mathsf{EVENT\_COND}\, sc\ transset\ cf\ cf' = \qquad (1)$$
$$\mathsf{EVERY}\,(\lambda tlabel.\,\mathsf{UpdateEvent}\,(\mathsf{EVENT}\,(\mathsf{TRANS}\ sc\ tlabel))\,(cf, cf'))\,transset$$

where $transset$ is the set of numeric transition labels.

5.2 Transition Condition

A *step* means following a set of transitions which satisfy a number of conditions. Each transition is represented by a Boolean flag indicating if the transition is taken in this step. Because statecharts can describe non-deterministic operation, there will be several different sets which are eligible. If a vector of transition flags satisfies the transition condition then it represents a legal set. We resolve the issues raised in Sects.3.1 and 3.4 by giving the conditions which this set must satisfy:

1. A transition is enabled if the system is in its source state and its trigger is true. Any transitions which are followed must be enabled.
2. If two or more transitions are enabled and have the same source state, only one will be taken but it is indeterminate as to which will be chosen.
3. Within an OR-state, only one transition can be followed.
4. Transitions may be followed within each of the substates of an AND-state.
5. If a transition from a parent state is enabled, it has precedence over one from a descendant.
6. If one or more transitions are enabled then some set of transitions will be followed.

The conditions on transitions relative to the hierarchy of states is determined by their source state.

We make the assumption that transitions do not go between components of an AND-state (Sect.3.6) and at the present time, we will also assume that destination states for transitions which are chosen do not conflict.[2] This second assumption should be relaxed in the final version of the semantics. With these assumptions, the above conditions ensure that more than one chosen transition does not modify the same basic state. Provided that the system is currently

[2] Destination states for chosen transitions will never conflict if transitions only go between substates of the same parent state.

in a legal state configuration, the next configuration will be legal if the set of transitions taken satisfy these conditions. For our purposes, a step does not include transitions triggered by events occurring in this step and therefore only transitions out of the set of states at the beginning of the step are considered (Sect.3.1).

To satisfy the first three conditions, we can consider the transitions among the substates of a given OR state within the hierarchy. At this one level, we can take exactly one of the transitions providing it is enabled:

$$\begin{aligned}&\mathsf{ONE\_LEVEL}\ cf\ sc\ index\ bvtrs = \mathsf{X\_EXISTS}\,(\lambda y.\, y)\ bvtrs\ \wedge\\ &\mathsf{PAIR\_EVERY}\,(\lambda(flag, i).\ flag \Longrightarrow \mathsf{TRIGGER}\ sc\ i\ cf)\ bvtrs\ index\end{aligned}$$

where *index* is the list of transition labels for this level and *bvtrs* is the list of associated flags. The fourth condition is given by saying that the function ONE_LEVEL must be true in all components of an AND-state.

We now have to consider multiple levels in the hierarchy and the priority among these levels. The priority of transitions is given by checking if there is any way to satisfy the function ONE_LEVEL for a given level by existentially quantifying over its transition flags. Only if there is no way to satisfy this function, do we consider transitions originating at lower levels. The function EXISTSN creates a number of existentially quantified Boolean variables in a bit vector (*bvtrans*) which is given as a parameter to the second argument of EXISTSN[9]. In pseudo-code, the transition condition can be given as:

$$\begin{aligned}&\mathsf{TRANS\_COND}\ cf\ sc\ bvtrans\ stn = \qquad (2)\\ &(\mathsf{TYP}\ sc\ stn = \mathsf{B}) \Rightarrow \mathsf{T}\ |\\ &(\mathsf{TYP}\ sc\ stn = \mathsf{A}) \Rightarrow\\ &\qquad \mathsf{EVERY}\,(\mathsf{TRANS\_COND}\ cf\ sc\ bvtrans)\,(\mathsf{SUBSTATES}\ sc\ stn)\ |\\ &\textbf{let}\ here = \text{set of transitions at this level}\ \textbf{in}\\ &\textbf{let}\ priioritytest = \mathsf{EXISTSN}\,(\mathsf{LENGTH}\ here)\,(\mathsf{ONE\_LEVEL}\ cf\ sc\ here)\ \textbf{in}\\ &\quad priioritytest \Rightarrow (\mathsf{ONE\_LEVEL}\ cf\ sc\ here\,(\text{transition flags for this level}) \wedge\\ &\qquad\qquad (\text{all flags for lower levels set to false}))\ \ |\\ &\qquad\qquad\qquad ((\text{all flags for this level set to false}) \wedge\\ &\qquad\qquad \mathsf{EVERY}\,(\mathsf{TRANS\_COND}\ cf\ sc\ bvtrans)\,(\mathsf{SUBSTATES}\ sc\ stn))\end{aligned}$$

5.3 Variable Condition

Given the set of transitions which can be taken, we now have to determine the effects of these transitions on the whole system. The function RESULT returns the set of modifications for exiting the source state, carrying out the actions, and entering the destination state.

Results of a transition. The transitions are labelled with actions which modify the data items in the system. These actions can all be defined in terms of an assignment statement. The semantic function for an assignment statement returns a pair, (v, e), which indicates that the expression e, evaluated in the current configuration, should be assigned to the variable v in the next configuration.

Executing a transition modifies the configuration not only by the actions but also by leaving the source state and entering the destination state. The variables for the basic states of the source state must be set to false and the ones for the destination should be be set to true by following default entrances. If the destination modifications overlap with the ones for the source (for example if a transition loops), then the changes for the destination have precedence.

Combining results of executing transitions. The values of the variables in the next configuration must satisfy the following three properties:

1. If a given transition is taken, at the end of the step the system will be in a configuration which includes the destination state of the transition and all its actions will be carried out except where conflicts occur among the actions of all transitions.
2. If more than one modification is made to the same variable (i.e. a conflict occurs) then exactly one of these modifications will be true in the next configuration.
3. If a variable is not modified by any transition in a step, then it retains its previous value.

Modifications from all transitions are considered together, whether they came from the same or different transitions. The variable condition resolves conflicts among assignments (CH) as discussed in Sects.3.2 and 3.3 and ensures the last property for variables which are not changed (UNCH):

$$\begin{aligned} &\mathsf{VAR\_COND}\, sc\ cf\ cf'\ transset\ bvtrans\ varlist = \\ &\mathsf{EVERY}(\lambda v.\, \mathsf{UNCH}\ sc\ v\ cf\ cf'\ transset\ bvtrans \vee \\ &\quad \mathsf{CH}\ sc\ v\ cf\ cf'\ transset\ bvtrans)\, varlist \end{aligned} \tag{3}$$

where *sc* is the textual representation for the statechart, *varlist* is the set of variables, *cf* is the current configuration, *cf'* is the next configuration, *transset* is the set of labels for the transitions, and *bvtrans* is a bit vector containing the flags for the transitions.

Classification of variables. Only variables under this system's control should necessarily keep their previous value if they are not modified, i.e. internal variables. External data-items and events may not retain their previous values between steps. The variables for the basic states are all internal, but the classification of events and data items as external or internal must be given. We assume that *varlist* includes only the internal variables.

Unchanged variables. The function UNCH uses the set of modifications returned by RESULT and the transition flags to determine if a variable, v, has not been changed in a step, and therefore should keep its previous value[3]:

$$\begin{aligned}&\mathsf{UNCH}\ sc\ v\ cf\ cf'\ transset\ bvtrans = \\ &\qquad \mathsf{EVERY}\,(\lambda tlabel.\,\neg\mathsf{EL}\ tlabel\ bvtrans \vee \\ &\neg\mathsf{MEMBER}\ v\,(\mathsf{CHANGEDVAR}(\mathsf{RESULT}\ sc\ tlabel)))\ transset\ \wedge \\ &\qquad \mathsf{EQUAL}\,(\mathsf{SemVAR}\ v\ cf)(\mathsf{SemVAR}\ v\ cf')\end{aligned}$$

Resolving conflicts. When more than one assignment is made to the same variable, the actions are treated atomically and exactly one of the possible modifications occurs. The function ACT looks at the list of modifications (*modlist*) and forms the disjunction of all possible modifications to a variable (v), evaluating the expressions relative to the current configuration:

$$\begin{aligned}&\mathsf{ACT}\ v\ modlist\ cf\ cf' = \\ &\mathsf{EXISTS}\,(\lambda asn.\,(\mathsf{FST}\ asn = v) \wedge \mathsf{EQUAL}\,(cf'\ v)\ ((\mathsf{SND}\ asn)\ cf))\ modlist\end{aligned}$$

Applying the ACT function to the results of each transition and then taking the disjunction of these clauses for all chosen transitions produces the effect of taking the disjunction of all possible modifications to a variable (v) in a step:

$$\begin{aligned}&\mathsf{CH}\ sc\ v\ cf\ cf'\ transset\ bvtrans = \\ &\mathsf{EXISTS}\,(\lambda tlabel.\,\mathsf{EL}\ tlabel\ bvtrans \wedge \mathsf{ACT}\ v\,(\mathsf{RESULT}\ sc\ tlabel)\ cf\ cf')\ transset\end{aligned}$$

5.4 Next Configuration Relation

The next configuration predicate combines all of the restrictions given above to produce a relation between the previous configuration (cf) and the next configuration (cf') for a particular statechart (sc). The set of internal variables is given as a parameter (*varlist*). The variable *root* is the ancestor of all states.

$$\begin{aligned}&\mathsf{NC}\ sc\ varlist\ cf\ cf' = \\ &\quad \textbf{let}\ root = \mathsf{ROOT}\ sc\ \textbf{and}\ transset = \mathsf{GET\_TRANS\_LABELS}\ sc\ \textbf{in} \\ &\qquad \textbf{let}\ transnum = \mathsf{SUC}\,(\mathsf{MAX\_TRANS}\ transset)\ \textbf{in} \\ &\qquad\quad \mathsf{EXISTSN}\ transnum\,(\lambda bvtrans. \\ &\qquad\quad \mathsf{EVENT\_COND}\ sc\ transset\ cf\ cf'\ \wedge \qquad (1) \\ &\qquad\quad \mathsf{TRANS\_COND}\ cf\ sc\ bvtrans\ root\ \wedge \qquad (2) \\ &\qquad\quad \mathsf{VAR\_COND}\ sc\ cf\ cf'\ transset\ bvtrans\ varlist) \qquad (3)\end{aligned}$$

[3] EL n l returns the nth element of l.

6 Embedding the Semantics in HOL

Most of the preceding definitions can be input directly into HOL, but in a few cases recursive definitions over the hierarchy of the statechart are used. Since HOL does not provide general recursion, these need to be phrased in terms of primitive recursion. We do this by giving the length of the list of states in the statechart as an extra argument to recurse over. This is an upper bound on the recursion since the statechart hierarchy would have to be a degenerate tree to reach this bound. Definitions like **INSTATE** become:

$$
\begin{aligned}
&(\mathsf{INSTATE}\ \mathsf{0}\ sc\ cf\ stn = \mathsf{F}) \wedge \\
&(\mathsf{INSTATE}\,(\mathsf{SUC}\,n)\ sc\ cf\ stn = \\
&\qquad ((\mathsf{TYP}\ sc\ stn = \mathsf{B}) \wedge (\mathsf{SemVAR}\ stn\ cf)) \vee \\
&((\mathsf{TYP}\ sc\ stn = \mathsf{A}) \wedge \mathsf{EVERY}\,(\mathsf{INSTATE}\ n\ sc\ cf)\,(\mathsf{SUBSTATES}\ sc\ stn)) \vee \\
&((\mathsf{TYP}\ sc\ stn = \mathsf{O}) \wedge \mathsf{EXISTS}\,(\mathsf{INSTATE}\ n\ sc\ cf)\,(\mathsf{SUBSTATES}\ sc\ stn)))
\end{aligned}
$$

7 Model Checking

The next configuration relation forms the basis for a model checking function written in HOL, and executed in VOSS using the efficient representation of its BDD package. This model checker tests boolean expressions relative to the current configuration. These may be invariants to prove safety properties, functional or timing requirements.

The model checking function quantifies over the bits used to represent the configuration and iteratively checks that the expression holds through a limited number of steps. By starting in any possible system configuration, we can show that the property holds for all time. A full description of the model checker and examples of its use can be found in [1].

The end result is that an operational specification can be created in the CASE tool STATEMATE and then analyzed using a BDD-based model checker. We expect it is possible to make the semantic functions more efficient to speed up this analysis.

8 Conclusion

This paper presents a high-level view of an operational semantics for a working subset of statecharts in quantifier free higher order logic as a next configuration relation. It is important to note that since HOL only deals with total functions, every statechart has an interpretation.

The difficulty in giving these semantics is that the components of statecharts are not completely compositional. The meaning of expressions and actions are expressed simply by examining their parts. Timeout events use counters so they can be evaluated relative the current configuration only. But the overall conditions on transitions and values of variables in the next configuration have to

consider all parts of the statechart. Our definition of a *step* is simpler than that used by other versions of the semantics but is easier and clearer to express. It implements non-determinism among transitions on the same level and priority for transitions from states related in the hierarchy. Race conditions and multiple conflicting actions on a transition are resolved by considering all actions together when assigning values to the variables in the next configuration. Interpretations for enabled transitions with conflicting destination states and those which go between orthogonal components have not yet been included.

These semantics form the basis of a model checker for statecharts, but they could also be used to examine their properties. Since they are presented within the framework of HOL, it is open for others to use this theorem-prover to examine these semantics.

9 Acknowledgements

This work was completed while the first author was funded by a Canadian Natural Science and Engineering Research Council Post Graduate Scholarship. We are indebted to the Integrated Systems Design Laboratory for interesting discussions where many of these ideas originated and the graduate students at UBC for comments on drafts of this paper.

References

1. Nancy Day. A model checker for statecharts. Master's thesis, University of British Columbia, 1993. In preparation.
2. D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pages 54–64, Ithaca, New York, June 1987.
3. David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.
4. David Harel, H. Lachover, et al. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.
5. i-Logix Inc., Burlington, MA. *Statemate 4.0 Analyzer User and Reference Manual*, April 1991.
6. Jeffrey J. Joyce. Totally verified systems: Linking verified software to verified hardware. Technical Report No. 178, University of Cambridge Computer Laboratory, September 1989.
7. Nancy G. Leveson, Mats P.E. Heimdahl, Holly Hildreth, and Jon D. Reese. Requirements specification for process-control systems. Technical Report 92-106, University of California, Irvine, Information and Computer Science, 1992.
8. A. Pnueli and M. Shalev. What's in a step: On the semantics of statecharts.
9. Carl-Johan H. Seger and Jeffrey J. Joyce. A mathematically precise two-level formal hardware verification methodology. Technical Report 92-34, University of British Columbia, Department of Computer Science, December 1992.