DYNAMIC PROPERTIES OF STATECHARTS: A REACHABILITY TREE AND ANALYSIS OF SOME PROPERTIES

P. C. Masiero
Ines G. Boaventura
J.C. Maldonado
Department of Computer Science and Statistics - ICMSC
University of São Paulo, Brazil - P. O. Box 668, 13560 - São Carlos, SP

Abstract

In this paper we present an algorithm to create a reachability tree for statecharts and show how to use this tree to analyze dynamical properties of statecharts. The properties analyzed are: reachability from any state configuration, usage of transitions, deadlocks, and valid sequence of events. Statecharts are a recent extension to finite state machines with capability for expressing hierarchical decomposition and parallelism. They have also a mechanism called history, to remember the last visit to a superstate. These features make it more difficult building a reachability tree for statecharts and we show how these problems were solved in the tree we propose.

Keywords: Statecharts, reactive systems, reachability tree, exhaustive simulation, dynamic properties of statecharts.

Acknowledgement: The first author has financial support from CNPq.

1 Introduction

Statecharts are a visual formalism for specification of reactive systems, proposed by Harel, in [4]. They are an extension to conventional state-transition diagrams based on finite state machines. Its outstanding features are hierarchy of states, parallelism and a communication mechanism via broadcasting. It also features a rich set of special notations for augmenting the notational power thus allowing the specification of complex problems in very concise diagrams. This is also the main reason why it is difficult to validate a statechart against its intended behaviour.

A possible solution to this problem is the exhaustive simulation of the statechart to produce all possible scenarios. However, the number of different possibilities may be very large, rendering impractical any exhaustive search. This problem does not have a solution but can be circumvented if we simulate selected parts of large statecharts. This is greatly facilitated by the diagram's hierarchical nature. The same problem can be attributed to Petri Nets, for which the methods proposed so far for verification of complex systems are usually based on choosing partial specifications to validate [7]. The solution we propose here is based to some extent on the reachability tree already developed to simulate the different configurations of a Petri Net [9].

The tool called STATEMATE has already implemented exhaustive search algorithms to many of the properties we will show in this paper [6], [3]. We, nevertheless, believe the solution presented here is relevant because it is based on a concept of reachability tree which is very well established. Also, after a literature review, we have not

found any other solution to this problem and the method used by STATEMATE is not public domain.

The structure of this paper is as follows. After a brief overview of statecharts, a reachability tree for statecharts is proposed in Section 2. Also in this section, one example of statechart and its corresponding tree is presented. Next, in Section 3, based on the reachability tree proposed, several dynamic properties of statecharts are defined. It is also briefly shown how they can be verified using the reachability tree.

2 Defining a Reachability Tree for Statecharts

The basic components of the statecharts are the *states*, as in the conventional state-transition diagrams. However, states may be embedded into *superstates* creating hierarchies of states. The superstates may be of two types: AND-States or XOR-States. The components of an AND-State are called *orthogonal* components and have the distinctive feature that when a system is in an AND-State it is also in all its orthogonal components. Conversely, when the system is in an XOR-State, it must be in only one of its substates.

States interact through *transitions* which have three basic parts: an event name, a condition, and action statements. These three components appear as a label attached to the transition and have the syntax: $event - expression[condition]/action$. The event expression can be null if there is a condition. The action part is also optional. A transition fires when both the event-expression and the condition evaluate to true. In this case, the actions are executed. Execution of an action may generate other events which are broadcast to the orthogonal components possibly firing new transitions. Assignment of arithmetic and logical expressions to variables is also allowed as legal action statements.

Execution of a statechart is based on a sequence of time steps where, in each step, events generated by the environment, added to the ones generated internally as a consequence of action execution, cause a set of transitions to be fired. The intended semantics is that firing a transition takes no time while being in a state takes a certain time [5]. The example in Figure 1 shows that state A is an XOR-superstate and has five substates: B, C, G, H and D. Only one of them can be active when A is active. State F has two orthogonal components: F1 and F2. When the transition labeled by event g fires, F is entered and the basic states F11 and F21 are activated explicitly; F1 and F2, as well as F, are also activated implicitly.

The history symbol attached to the transition labeled by event f means that the default state E will be activated the first time that transition fires. This will never in fact happen in this diagram because E is the default state and will be activated together with A. Afterwards E or F will be chosen, whichever was last visited. If it is the case that F is chosen, then its default states will be activated. When we have recursive history (H*) the actual basic states last visited will be activated, as in the case of the transition labeled d, leaving from H.

The reachability tree proposed is based on statechart configurations. The following notation for representing a configuration is used: parentheses to represent XOR-superstates; brackets to represent AND-superstates; and, the values 1 and 0 to represent basic states, showing whether they are active or not active, respectively. Hence,

be noticed however, that we will consider each single transition and not more than one ocurring at the same time step as is the semantics defined in [5] and [8].

For each element in this sequence we take the set $SR = \{(\Upsilon_j, \Pi_j^*)\}$ of all possible system reactions. In the tree, each node will represent a statechart configuration. They are classified into four different types: new configuration, old configuration, terminal configuration and history configuration. After being computed, a configuration will be classified as new if it was not already in the tree; otherwise, it will be classified as an old configuration. A configuration will be terminal when it does not have any enabled transition, i.e., SR=ϕ. A history configuration represents all configurations reachable from transitions which have a history symbol attached. We will use the symbols "H" and "*" to represent this case. When eventually we search the tree, the history symbols will be replaced by the last configuration of those states in the scope of the history symbol. This will be clarified later with an example.

The initial (or default) configuration is the reachability tree's root node. Considering it as a new configuration to be processed, we take the set of all transitions which will cause this configuration to be left explicitly or implicitly. For each transition taken, a new step is computed generating the next configuration which is classified and appropriatelly inserted in the tree. This procedure is then repeated for each new configuration until there is none of them. The algorithm is listed in Figure 3.

Statechart semantics also permits many events to happen in a time instant thus firing different transitions in orthogonal components. We decided not to combine the different transitions enabled in orthogonal components. This would make the algorithm much more complex because in the case of more than two transitions enabled in orthogonal components we would have to consider all possible combinations.

The reachability tree shown in Figure 2 corresponds to the statechart of Figure 1. In it we can see that when we activate a node by history we simply place "H" or "*" in the coordinates corresponding to states within the history scope. They will be replaced by the correct sub-configuration by the search algorithms as will be seen in the next section. A problem caused by history occurs when it is possible to reach a node which was never activated before. This cannot happen with the statechart of Figure 1 as all its states with history symbols associated are activated the first time the external state is activated. If instead we had another state as the default state of A, it would not be possible to have as the only arc leaving the initial configuration one leading to a history node. We create then two nodes, one for the default configuration of the states affected by history and another for the conventional history node. The former has its arcs labeled "<event expression>/def" and the latter has the usual "<event expression>" label. When we search such a reachability tree, the "/def" arc will be visited only once and discharged.

3 Dynamic Properties Verified through the Reachability Tree

Using the reachability tree proposed in Section 3 we can analyze several properties of a given statechart. See [1] for a thorough treatment.

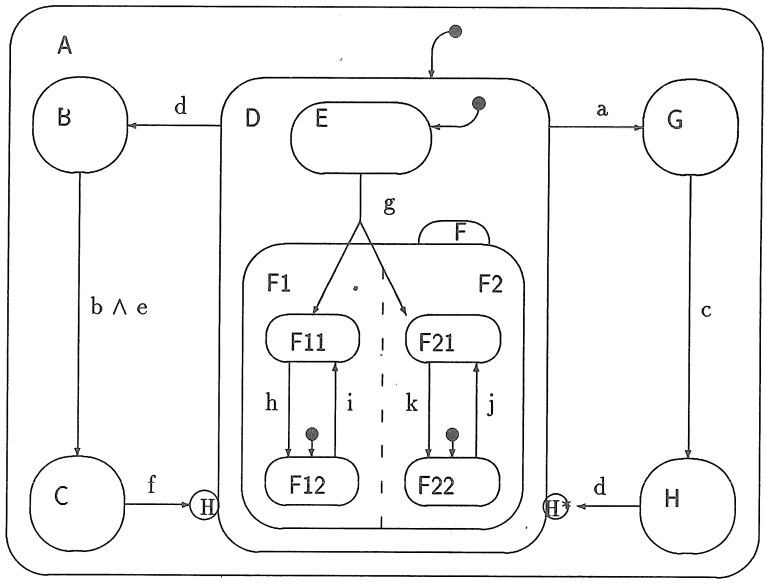


Figure 1: A Statechart with many features

the notation preserves hierarchy and decomposition type but only the basic states are explicitly represented. The superstates are inferred by the hierarchy. As an example, considering the statechart of Figure 1, any configuration would have the general form $(H, G, C, B, (E, [(F12, F11), (F22, F21)]))$. A configuration with the states F11 and F21 active is noted by $(0, 0, 0, 0, (0, [(0, 1), (0, 1)]))$ or by $(0, 0, 0, 0, (0, [(0, F11), (0, F21)]))$.

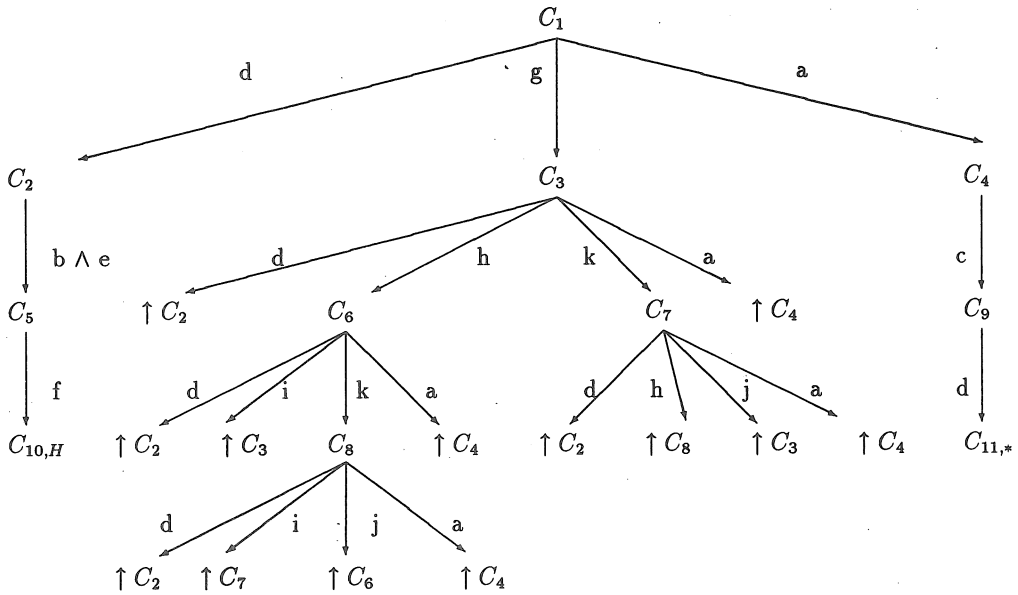
The reachability tree will be constructed considering every possible computation sequence. In each step we suppose that the event expression evaluates to true leading thus to a new configuration. Conditions are included in the event expressions and are also considered when firing a transition. The semantic rules for computing each step are the ones defined in [5]. The computation evolves in a sequence of time steps, which can be represented by the set $\{(SC_i, \Upsilon_i, \Pi_i^*), i \geq 0\}$, where

1. $SC_0 = (X_0, \Pi_0, \Theta_0, \xi_0)$, where X_0 is the initial configuration and the other symbols represent the external estimuli occurring in the first time interval I_0. Π_0 is the external set of events generated by the environment, Θ_0 is the set of primitive conditions whose value is *true*, and ξ_0 is a function that for every variable x in the environment computes the value of x.

2. Υ_i is a step taken in SC_i, at the time step σ_{i+1}, $i \geq 0$.

3. Π_i^* is the set of events generated by Υ_i, $i \geq 0$.

We have then a sequence of state configurations, where SC_{i+1} is the system



Where the configurations are:

C =$(H, G, C, B, (E, [(F12, F11), (F22, F21)]))$

C1 =$(0,0,0,0,(1,[(0,0),(0,0)]))$
C2 =$(0,0,0,1,(0,[(0,0),(0,0)]))$
C3 =$(0,0,0,0,(0,[(0,1),(0,1)]))$
C4 =$(0,1,0,0,(0,[(0,0),(0,0)]))$
C5 =$(0,0,1,0,(0,[(0,0),(0,0)]))$
C6 =$(0,0,0,0,(0,[(1,0),(0,1)]))$
C7 =$(0,0,0,0,(0,[(0,1),(1,0)]))$
C8 =$(0,0,0,0,(0,[(1,0),(1,0)]))$
C9 =$(1,0,0,0,(0,[(0,0),(0,0)]))$
C10=$(0,0,0,0,(H,[(H,H),(H,H)]))$
C11=$(0,0,0,0,(*,[(*,*),(*,*)]))$

Figure 2: Reachability Tree for the Statechart of Figure 1

Valid Sequence of Events

A sequence of event expression is valid if each event may cause a transition to fire yielding a configuration change. Let $s_1 s_2 ... s_{n-1}$ be a sequence of event expression. The sequence will be valid if we can find a sequence of state configurations $C_0 \xrightarrow{s_1} C_1 \xrightarrow{s_2} C_2 \ ... \ \xrightarrow{s_{n-1}} C_n$ where s_1 can fire from C_1, s_2 can fire from C_2, and so forth. Usually, C_0 is the initial configuration. The algorithm will stop if for any $C_i, 0 \leq i < n$, there is not an arc labeled s_i with source in C_i. The algorithm will succeed in validating the sequence of events when the whole sequence $C_0 C_1 C_2 ... C_n$ is found. Let s_2= g, h, d, b ∧ e, f, i. It is a valid sequence of events and the corresponding state configurations are: $C_1 \xrightarrow{g} C_3 \xrightarrow{h} C_6 \xrightarrow{d} \uparrow C_2 \xrightarrow{b \wedge e} C_5 \xrightarrow{f} C_{10,H} \rightsquigarrow C_6 \xrightarrow{i} \uparrow C_3$.

The example above illustrates two points. First, it is shown that when we reach node ↑ C_2 we know it is not a terminal node and that it is linked to the first occurrence of the same configuration in the tree. We then proceed from it. For example, all configurations ↑ C_2 are linked to the configuration C_2 which is the leftmost son of C_1.

C_6 as indicated by the history configuration.

Reachability

We want to know if a certain configuration can be reached from another one. A reachable initial state configuration must be a reachability tree's node. Therefore, we search the configuration in the reachability tree and, if it is found, the sequences starting from the initial configuration are shown – they are the arc's labels in the path from the initial configuration to the target configuration. There may be many paths leading to the input configuration. If we use a breadth-first search strategy the shortest path will be found first. Others will be found, if there are any, continuing the search after the first occurrence is found. Let us consider the configuration with active state B. The configuration labeled (0,0,0,1,(0,[(0,0),(0,0)]) will be found in node C2. The path from the initial configuration to it is given by the event sequence <d>. Other paths can be found for this particular configuration, e.g., <g,d> and <g,k,d>.

Reachability from any configuration C_1 to another configuration C_2 is easily obtained if we: a) make sure both C_1 and C_2 are valid configurations; and b) build the reachability tree considering C_1 as the initial configuration. Then we have to find C_2 in the tree, as before. When there is history involved we will find the path leading from C_1 to C_2 always considering the first (default) visit to C_2.

Deadlock

A system modelled by a Statechart may reach a deadlock situation if its corresponding reachability tree has a terminal node. The converse is not necessarily true because a system may reach intentionally an end state as part of its specification. The algorithm looking for deadlocks has then to search the entire reachability tree looking for terminal nodes. They are output as possible deadlock configurations as well as the path from the initial configuration to the terminal node. History nodes are also leaves but as we always know how to proceed from them they are not considered terminal nodes. The example shown in this paper does not reach a deadlock configuration; examples can be found in [1].

Usage of Transitions

This property is verifyed looking for transitions which will never fire or maybe transitions left out of the specification (completeness). If this is the case, an arc labeled with the transition's label we are looking for will not appear in the reachability tree. Hence, the algorithm accepts as input an event expression supposed to be the label of a transition and searches for an arc with that label. If found, its source node (configuration) is output; otherwise, the transition is not used. As an example let us consider the transition labeled c. It is used and may fire from node C_4, whose configuration is (0,1,0,0,(0,[(0,0),(0,0)]).

4 Concluding Remarks

The algorithm for building the reachability tree as well as the ones for verifying properties were implemented and some testing conducted within an environment where there was already available some tools to support statechart specification and simulation: a graphical statechart editor, an analyzer for textual input, and a simulator [2]. The existence

```
01 BEGIN BuildSTR
02   Create an empty queue CQ
03   Make SC0 the root of the reachability tree STR
04   Insert SC0 in CQ
05   WHILE CQ is not empty
06     Remove SC from CQ
07     Let SC be the statechart configuration
08     Obtain SR, the set of all possible transitions for SC
09     IF SR is not empty THEN
10       FOREACH transition in SR
11         Let T be a transition in SR
12         Compute the step for T obtaining SC'
13         IF any destination state DS^k has a history symbol
                or its default state has a history symbol THEN
14           IF DS^k has not been active in this path THEN
15             Get DS_0^k, the default configurations for DS^k
16             Create a node with SC' replacing the subconfiguration of DS^k by DS_0^k
17             IF SC' exists in SRT THEN
18               Make node SC' an old configuration
19             ELSE
20               Make node SC' a new configuration
21               Insert SC' in CQ
22             ENDIF
23             Insert node SC' in SRT as a son of SC
24             Label the arc from SC to SC' as "<event-exp> /def"
25           ENDIF
26           Create a node SC'' from SC' replacing the DS^k subconfiguration by (H/*)
27           Make node SC'' a history configuration
28           Insert node SC'' in SRT as a son of SC
29           Label the arc from SC to SC'' as "<event-exp>"
30         ELSE
31           Create a node with SC'
32           IF SC' exists in SRT THEN
33             Make node SC' an old configuration
34           ELSE
35             Make node SC' a new configuration
36             Insert SC' in CQ
37           ENDIF
38           Insert node SC' in SRT as a son of SC
39           Label the arc from SC to SC' as "<event-exp>"
40         ENDIF
41         Reinstate SC as the statechart configuration
42       ENDFOREACH
43     ELSE
44       Make node SC a terminal configuration
45     ENDIF
46   ENDWHILE
47 END BuildSRT
```

Figure 3: Algorithm for Building a Reachability Tree

of these tools eased enormously the implementation effort and preliminary validation of these algorithms.

The many existing features in the statecharts forced us to define extensions to the conventional reachability trees, the major one being the history mechanism. Also, the mechanism for restoring a history configuration from the list of states in the search path is essential to representing history in the reachability tree. A trade-off decision had to be made as to represent explicitly or not the transitions leaving a superstate. We made the decision of representing it in each subconfiguration, trading the tree's size by having simpler algoritms for searching the tree. The algorithm for building the reachability tree would be much simpler if we had dropped history, as was done, for example, in [8], yet we believe the solution presented in this paper is simple and elegant. Without history we would not have available a powerful mechanism to represent concisely many recurrent system requirements.

References

[1] I. G. Boaventura. *Dynamical Properties of Statecharts* (in Portuguese). M.Sc. Dissertation, ICMSC-USP, São Carlos, 1992.

[2] R. P. de M. Fortes. *A Tool for Statechart Simulation* (in Portuguese). M.Sc. Dissertation, ICMSC-USP, São Carlos, 1991.

[3] i-LOGIX. The STATEMATE Approach to Complex Systems. 1989.

[4] D. Harel. STATEMATE: a Visual Formalism for Complex Systems. *Science of Comp. Programming*, 8 (3), 231-274, 1987.

[5] D. Harel. STATECHARTS: On the Formal Semantics of Statecharts.*Proceedings of the 2nd IEEE Symposium on Logic in Computer Science.* Ithaca, New York, 1987.

[6] D. Harel et alli. STATECHARTS: A Working Environment for the Development od Complex Reactive Systems. *Proceedings of the Tenth International Conference on Software Engineering.* (Singapure, April), Washington D.C.,IEEE, 1988.

[7] N. G. Leveson; J. L. Stolzy. Safety Analysis Using Petri Nets. *IEEE Trans. on Software Engineering*, SE-13 (3), 386-397, 1987.

[8] J. J. M. Hooman; S. Ramesh; W. P. de Roever. A compositional Axiomatization of Statecharts. To appear in *Theoretical Computer Science.*

[9] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Inc., Englewood Cliffs, N.J.,1981.