The STATEMATE Semantics of Statecharts

DAVID HAREL
The Weizmann Institute of Science
and
AMNON NAAMAD
i-Logix, Inc.

We describe the semantics of statecharts as implemented in the STATEMATE system. This was the first executable semantics defined for the language and has been in use for almost a decade. In terms of the controversy around whether changes made in a given step should take effect in the current step or in the next one, this semantics adopts the latter approach.

Categories and Subject Descriptors: D.2 [**Software**]: Software Engineering; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms: Languages

Additional Key Words and Phrases: Behavioral modeling, reactive system, semantics, statechart, STATEMATE

1. INTRODUCTION

The article that introduced the language of statecharts [Harel 1987] presented only a brief discussion of how its semantics could be defined. A rigorous semantics was first defined for the language in Harel et al. [1987]. Since then, many variants of statecharts have been proposed in the literature, and several papers include definitions of semantics too. Some examples are Huizing and de Roever [1991], Huizing et al. [1988], Kesten and Pnueli [1992], Leveson et al. [1995], Maraninchi [1992], and Pnueli and Shalev [1991]. A recent survey [von der Beek 1994] discusses nearly 20 variants.

This work is a revised version of "The Semantics of Statecharts," Technical Report, i-Logix, Inc., 1989 and 1991. D. Harel's work was supported by a grant from the Israel Academy of Sciences.
Authors' addresses: D. Harel, The Weizmann Institute of Science, Rehovot, Israel; email: harel@wisdom.weizmann.ac.il; A. Naamad, i-Logix, Inc., Andover, MA 01810; email: amnon@ilogix.com.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 1996 ACM 1049-331X/96/1000–0293 $03.50

Subtle issues arise when one tries to define a semantics for the language, and there is no consensus on the "right" way to go about the task. Since statecharts were intended from the start to be used by real engineers in specifying real systems, one of the central considerations in deciding upon a useful semantics is clarity and simplicity. A user need not see the details of the mathematical definition, but he or she must be able to understand how it works in a relatively intuitive way.

This article describes the semantics of the language of statecharts as implemented in the STATEMATE[1] system [Harel et al. 1990; Harel and Politi 1996]. The initial version of this semantics was developed by a team about 10 years ago. With the added experience of the users of the system it has since been extended and modified. This executable semantics has been in operation in driving the simulation, dynamic tests, and code generation tools of STATEMATE since 1987, and a technical report describing it has been available from i-Logix, Inc. since 1989. We have now decided to revise and publish the report so as to make it more widely accessible, to alleviate some of the confusion about the "official" semantics of the language, and to counter a number of incorrect comments made in the literature about the way statecharts have been implemented. For example, the survey [von der Beek 1994] does not mention the STATEMATE implementation of statecharts or the semantics adopted for it at all, although this semantics is different from the ones surveyed therein (and was developed earlier than all of them except for Harel et al. [1987]). As another example, Leveson et al. [1995] describe a case that exhibits an unacceptable kind of behavior in a statechart, which they say is what the "semantics of statecharts" leads to (pp. 695–697). Unfortunately, they base their discussion of statechart semantics on one of the many semantics proposed by various authors (that of Pnueli and Shalev [1991]) and give the reader the impression that this is the official semantics of the language.

Being an unofficial language, statecharts clearly have no official semantics, and researchers are free to propose semantics as they see fit. However, the only implemented and working semantics for statecharts has for many years been the one described here, and it does not in any way exhibit the kind of behavior described in the example and surrounding text in Leveson et al. [1995].[2] In fact, our semantics is quite similar to the approach suggested (seven years later, we might add) in Leveson et al. [1995] itself, although almost a dozen years ago we contemplated building STATEMATE with a semantics close to that of Harel et al. [1987] and Pnueli and Shalev [1991]. The main difference, which served as the central topic of our often heated deliberations, was whether changes that occur in a given step (such as generated events or updates to the values of data items) should take effect in the current step or in the next one. The semantics we finally

[1]The current version of STATEMATE, available from i-Logix, Inc., has been termed STATEMATE MAGNUM.
[2]These parts of Leveson et al. [1995] have managed to alarm some users of STATEMATE, who feared that their models were dangerously erroneous.

adopted, in contrast to those of Harel et al. [1987] and Pnueli and Shalev [1991], takes the latter approach. In Appendix A we address the comparison of our semantics with others, and especially relate to the issues spelled out in von der Beek [1994].

The main consideration behind the definition and selection of the semantics described here is that STATEMATE is a commercial tool, designed for the specification and design of real-life complex systems, coming from a variety of disciplines. As such, the semantics has to be rich enough to support different styles of modeling, yet it should be simple and intuitive. It must also be technically straightforward enough to enable fast simulation of models and to generate useful hardware and software code out of these models. It is noteworthy that VHDL, the IEEE standard language for the description of hardware, which evolved around the time the semantics of statecharts as described here was defined, adopted the same approach, i.e., that changes carried out in a given step (or "delta" in the VHDL nomenclature) can be sensed only in the following one.

Despite all of this, we make no qualitative claims about this semantics (and, as any semanticist knows, anomalous examples can be constructed for almost any semantics proposed for a concurrent language), except that it represents the way the language has been implemented in STATEMATE.

Due to the somewhat different goals and usage of STATEMATE's analysis tools (simulation and dynamic tests) on the one hand and its generated code on the other, in some extreme cases there may be slight differences between the behaviors entailed by these tools. These include the treatment of nondeterminism, racing, and states without enabled default transitions. In all such cases, the analysis tools issue a warning message. More details are provided later.

The article concentrates mainly on notions that are special to statecharts, such as the hierarchy of states, orthogonality, and history connectors. There are a number of issues that are not unique to statecharts yet are not widely available in other formalisms either, and thus they may be new to STATEMATE users. Some of these, such as generic charts, multivalue logic (MVL), and queues, are treated in detail in i-Logix documents, and we do not describe them here. We do describe one of these in Appendix B, namely, combinational assignments. Also, Appendix C discusses the priority of transitions, which is in the process of undergoing a change in coming versions of STATEMATE.

2. THE BASICS

The STATEMATE set of languages is used for modeling reactive systems [Harel and Pnueli 1985] based on the structured analysis paradigm. It is described in more detail in Harel et al. [1990] and Harel and Politi [1996]. The backbone of the system model is an **activity-chart**, which is a hierarchical data-flow diagram, and in which the functional capabilities of the system are captured by **activities** and the data elements and signals that can flow between them. The semantics of this functional description is

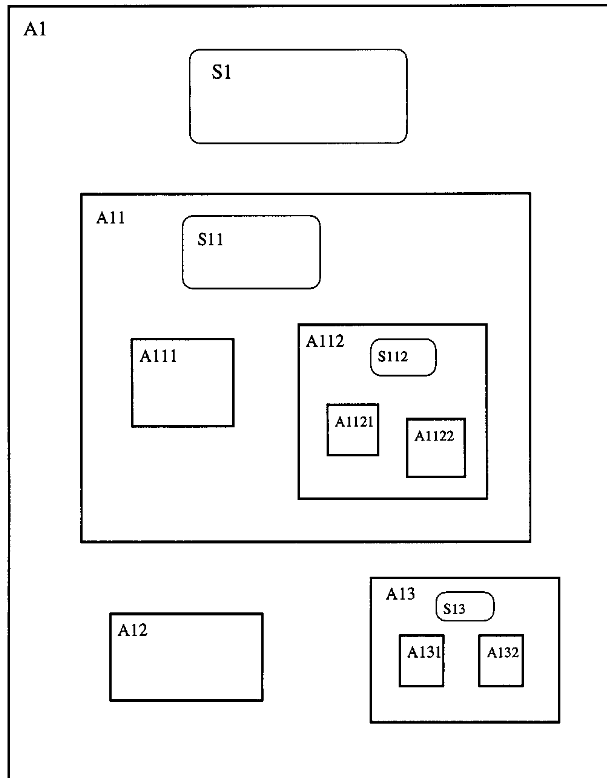

Figure 1.

dynamically noncommitting, in that it asserts that activities *can* be active, that information *can* flow, and so on, but it does not specify what *will* happen, or when or why. These behavioral aspects are specified in statecharts—sometimes called **control activities**—potentially one for each activity in the activity-chart. An activity's statechart controls the dynamics of subactivities and their data-flow, including the ability to activate and deactivate activities, to cause data to be written, modified, and read, to send signals, and to sense when such things have happened, thus affecting subsequent behavior. Figure 1 shows the structure of a simple hierarchy of activities in a STATEMATE model. The A-boxes are activities, and the rounded S-boxes are their control activities, which, as mentioned, are specified using statecharts.

The precise way in which statecharts describe behavior and thus control the behavior of the entire setup of activities and data over time is at the heart of the system model, which is why defining the semantics of statecharts is so crucial. In operational terms, STATEMATE's simulation, code generation, and dynamic tests tools all execute the model, either explicitly or implicitly, based on that very semantics.

A full definition of the allowed syntax of statecharts in STATEMATE is beyond the scope of this article. We refer the reader to a short description

in Harel et al. [1990] and a full one in Harel and Politi [1996]. Here are some brief reminders.

There are three types of states in a statechart: OR-**states**, AND-**states**, and **basic states**. OR-states have substates that are related to each other by "exclusive-or," and AND-states have orthogonal components that are related by "and." Basic states are those at the bottom of the state hierarchy, i.e., those that have no substates. The state at the highest level, i.e., the one with no parent state, is called the **root**.

The general syntax of an expression labeling a transition in a statechart is "$e[c]/d$," where e is the **event** that triggers the transition; c is a **condition** that guards the transition from being taken unless it is true when e occurs; and a is an **action** that is carried out if and when the transition is taken. All of these are optional.

There are several special events, conditions, and actions that relate to a STATEMATE model's other entities, such as activities, data items, or other states. For example, a can be the special action $start(P)$ (abbreviated $st!(P)$) that causes the activity P to start. Similarly, rather than being simply an external, primitive event, e might be the special event $entered(S)$ (abbreviated $en(S)$) that occurs (and hence causes the transition to take place) when state S is entered. Many of these are not mentioned explicitly in this article, as we concentrate on the statecharts themselves.

Events are closed under the Boolean operations *or*, *and*, and *not*, and so are conditions. The expression $e[c]$ above is interpreted as "e *and* c."

Besides allowing actions to appear along transitions, they can also appear associated with the entrance to or exit from a state (any state, on any level). Actions associated with the entrance to a state S are executed in the step in which S is entered, as if they appear on the transition leading into S. Similarly, actions associated with the exit from S are executed in the step in which S is exited, as if they appear on the transition exiting from S. (On the other end, the events $en(S)$ and $ex(S)$ are sensed one step after S was entered or exited, respectively.)

In addition, each state can be associated with **static reactions** (SRs), which are of the same format as a transition label, namely, $e[c]/a$, and are to be carried out (whenever enabled) as long as the system is in (and is not exiting) the state in question. Semantically, each SR in state S can be regarded as a transition in a virtual substate of S that is orthogonal to its ordinary substates and to the other SRs of S. For example, Figures 2(a) and 2(b) depict the same behavior.

An activity can be linked directly to a state S, by specifying it to take place *throughout* S or *within* S. The semantics of "activity A is active throughout state S" is that A starts being active upon entering S and stops upon leaving it. The semantics of "activity A is active within state S" is that A may be active when the system is in S; it entails stopping A when S is exited.

An action a can be scheduled for d time units later on by carrying out a new action of the form $schedule(a, d)$ (abbreviated $sc!(a, d)$). Similarly, the special event $timeout(e, d)$ (abbreviated $tm(e, d)$) occurs d time units

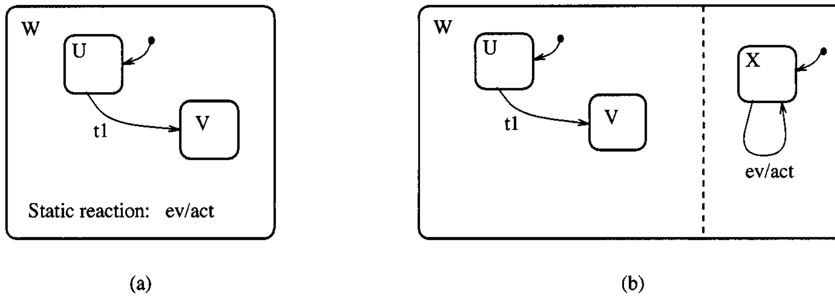

Figure 2.

after the most recent occurrence of the event e. The way these are calculated is discussed in Sections 8 and 9.

The behavior of a system described in STATEMATE is a set of possible **runs**, each representing the responses of the system to a sequence of external stimuli generated by its environment. A run consists of a series of detailed snapshots of the system's situation; such a snapshot is called a **status**. The first in the sequence is the initial status, and each subsequent one is obtained from its predecessor by executing a **step** (see Figure 3). As we shall see, defining a step precisely, with all its ramifications and side-effects, is what the semantics is all about.

A status contains information about active states and activities, values of data-items and conditions, generated events and scheduled actions, and some information regarding the system's history (its past behavior). At the beginning of each step, the environment supplies the system under description with external stimuli. These, together with changes that occurred in the system during and since the previous step, trigger transitions between states and static reactions within states. As a result, the system moves into a new status. Some states are exited, and some are entered; values of conditions and data-items are modified; new events are generated; activities are started and stopped, and so on.

Some of the general principles we have adopted in defining the semantics are the following:

(1) Reactions to external and internal events, and changes that occur in a step, can be sensed only after completion of the step.
(2) Events "live" for the duration of one step only, the one following that in which they occur, and are not "remembered" in subsequent steps.
(3) Calculations in one step are based on the situation at the beginning of the step (e.g., which states the system was in, which activities were active, and the values of conditions and data-items at that time.)
(4) A maximal subset of nonconflicting transitions and SRs is always executed. We refer to this as the "greediness property" of the semantics.

The execution of a step takes zero time. The time interval between the executions of two consecutive steps is not part of the step semantics.

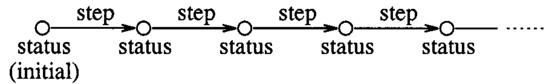

Figure 3.

Rather, it depends on the execution environment and the time model, over which users of the tool have a significant degree of control. This is discussed in Section 9. It is important to note, however, that STATEMATE supports time models in which several steps may execute at the same point in time. An event generated in step n at time t may be sensed in step $n + 1$ only. Moreover, this occurrence of the event is not sensed in steps that follow step $n + 1$, even if those steps also execute at time t.

In general, it is quite straightforward to define the effect of a step involving a single statechart transition, or several nonconflicting transitions in separate orthogonal components. However, there are less trivial situations. For example, when several transitions are enabled, some of them may be conflicting and thus unable to participate together in the same step. Also, when a transition arrow crosses the boundaries of nested states, one has to accurately define the states that have been exited by taking the transition and those that are entered. The main technical goal of this article is to describe the way the general case is treated. We have organized the definition incrementally, starting with the simplest and most obvious cases and complicating things as we go along.

3. BASIC SYSTEM REACTION

A **configuration** is a maximal set of states that the system can be in simultaneously. More precisely, given a root state R, a configuration (relative to R) is a set of states C obeying the following rules:

—C contains R.
—If C contains a state A of type OR, it must also contain exactly one of A's substates.
—If C contains a state A of type AND, it must also contain all of A's substates.
—The only states in C are those that are required by the above rules.

It follows that configurations are "closed upwards"; that is, when the system is in any state A, it must also be in A's parent state (unless, of course, A is the root, in which case it has no parent). In fact, to uniquely determine a configuration it is sufficient to know its basic states. Consequently, we use the term **basic configuration** to refer to a maximal set of basic states that the system can be in simultaneously, or in other words, the set of basic states in a legal configuration. To illustrate, consider Figure 4. In it, $\{B1, C1, D1\}$ is a basic configuration, and its full configuration contains also B, C, D, A, and S (the root). $\{B1, C1\}$ is not maximal and hence is not a basic configuration. $\{B1, B2, C1, D1\}$ is not a legal configuration, because the system cannot be simultaneously in the two

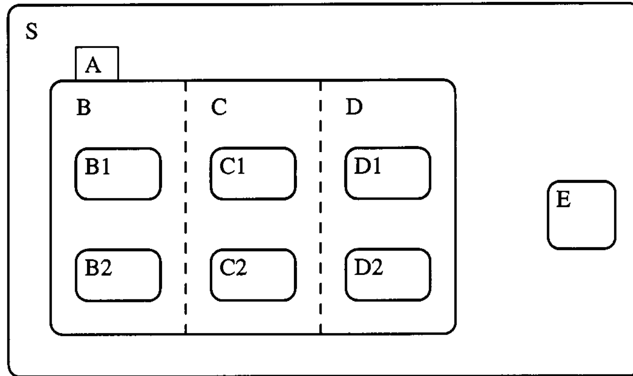

Figure 4.

offspring of the OR-state B. $\{E\}$ is a basic configuration, and its full configuration is the set $\{E, S\}$.

In a step, the system will typically carry out operations of four types: transitions, static reactions, actions performed when entering a state, and actions performed when exiting a state. In this section we discuss the simplest kind of step, involving a single transition in an ordinary, unadorned statechart. Consider Figure 5, and assume that the system is in state A and that event *ev* has just occurred.[3] The response of the system will be as follows:

—The transition $t1$ becomes enabled because the system is in $t1$'s source state and because its trigger (the event *ev*) is generated. Hence, $t1$ is taken, meaning that the system will exit state A, enter state B, and execute action *act*.

—The special events *exited*(A) and *entered*(B) are generated (and will be sensed in the following step).

—The special condition *in*(A) becomes false, and the condition *in*(B) becomes true.

—The actions specified to take place upon exiting state A are executed.

—The actions specified to take place upon entering state B are executed.

—All the SRs of state S that are enabled, that is, whose trigger is true, are executed. (This is because the system was in state S before the step and did not exit S during the step.)

—All activities that were specified as being active *within* or *throughout* state A are deactivated, while those defined as being active *throughout* state B (but not necessarily those defined as being active *within* B) are activated.

[3]In the figures we often write names for transitions, such as $t1$ in Figure 5. These are not parts of the STATEMATE syntax and are added just to help in the exposition. Also, shaded states in the figures serve to represent the basic configuration the system is in at the moment.

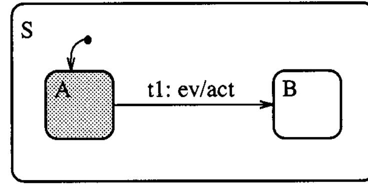

Figure 5.

As a consequence of the direct changes listed above, other events might very well be generated, and other conditions might change their value. For example, if the special event $fs(in(A))$ (or $tr(in(B))$) appears somewhere in the model, it will be generated as a result of executing $t1$, since it occurs whenever the condition in (A) becomes false. Also, if an activity P is activated, then the event *started*(P) occurs, and the condition *active*(P) becomes true.

As mentioned in the introduction, all these changes are sensed only in the following steps. For example, the fact that state A is exited indeed generates the event *exited*(A), but any static reaction in S that is to be triggered by that event will be executed only in the next step. The same applies to all generated events and changes in conditions and data-items. Among other things, this means that the system cannot enter and then exit a given state in the same step. (Interestingly, it is possible, by a self-looping transition, to *exit* and then *reenter* a state in one step.)

As an example, suppose that in Figure 5 the action *act* is defined as

$$X := X + 1; \quad Y := X * 5; \quad \textbf{if } X = 5 \textbf{ then } act1 \textbf{ else } act2 \textbf{ end if}$$

The value of X used in evaluating the arithmetic expression $X * 5$ and the Boolean expression $X = 5$ is the value X had at the beginning of the step (before it was incremented). If the value of X at the beginning of the step was 4, then $act2$ (and not $act1$) is executed. Thus, the semicolon in an action (whether appearing along a transition or within a static reaction) signifies more of "do this too" rather than "and then do," meaning that actions are to be executed as if in parallel.

When two or more actions executing in the same step call for changing the value of a common data-item, we cannot predict the outcome. Thus, two correct implementations of the same semantics may yield different results. In such cases, we say that we have a **racing condition**. The example above illustrates a less acute type of racing, which according to STATEMATE's semantics cannot affect the behavior: one action changes, and others use the value of X in the same step. Since the change occurs only at the end of the step there is no issue of unpredictability or differences in behavior, but this situation should be detected too, since the modeler's intention might have been different from the outcome. Both types of racing are detected and reported by STATEMATE's simulation and dynamic tests tools. Racing is discussed in more detail later.

4. COMPOUND TRANSITIONS

Execution of a step must always lead the system to a legal configuration. In particular, a statechart cannot be "stuck" during execution at a connector (with the exception of a termination connector). Similarly, a statechart cannot be in a nonbasic state without the ability to enter the appropriate substates. With this in mind, the actual transition between configurations that is taken in a step is often more complicated than that of Figure 5. It may consist of a number of separate transitions appearing in different orthogonal state components, and each of these may consist of a number of linked **transition segments**, which are the labeled arrows that connect states and connectors of various kinds.

This section illustrates the way transition segments are combined to form full transitions. It uses a number of different kinds of compound transitions. The first of these is a **basic compound transition** (CT), which is a maximal chain of transition segments, linked by connectors, that are executable simultaneously as a single transition. The trigger of a CT is taken to be the conjunction of the triggers of its constituent segments, and its action is the concatenation of the actions thereof.

The connectors that enable transition segments to be combined to form a CT come in two forms: AND and OR.

The **joint** and **fork** are AND-connectors (see Figures 9 and 10). The transition segments connected to an AND-connector will all participate in the same CT. In other words, if T is the set of transition segments leading to or emanating from an AND-connector C, then any CT that contains a segment from T must contain all the segments of T.

The **condition**, **selection**, and **junction** are OR-connectors (e.g., see Figures 7 and 31). Given an OR-connector C, let $T1$ and $T2$ be the sets of transition segments leading to and emanating from C, respectively. Any CT that contains a segment from $T1 \cup T2$ must contain exactly one segment from $T1$ and one from $T2$.[4]

There are two types of basic CTs: an **initial CT** is a CT whose source is a state, and a **continuation CT** is a CT whose source is a default or history connector. The targets of both types of basic CTs are states or history or termination connectors. A **full CT** is a combination of one initial CT and possibly several continuation CTs, which, when executed, lead the system to a full basic configuration. In many cases, like the one depicted in Figure 6(b), a full CT consists of only one basic transition. As the following examples show, the execution of a basic CT does not guarantee that the system will end up in a legal configuration. Therefore, a full CT is the central concept we have to work with.

[4]Another type of OR-connector is the **diagram** connector. Diagram connectors help reduce the clutter of arrows in the chart. Semantically, the best way to view diagram connectors is as if all the ones with the same label are actually a single OR-connector that happens to appear in several locations.

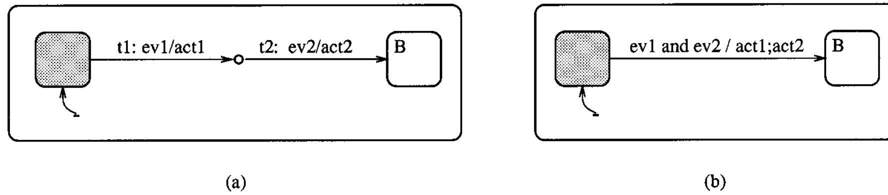

Figure 6.

—The transition $t1$ in Figure 6(a) cannot be executed without also executing $t2$. Thus the set $\{t1,\ t2\}$ is a CT (actually, it is a full CT) whose source is A and whose target is B. Its trigger is $ev1$ & $ev2$, and its action is $act1$; $act2$. Thus Figure 6(a) is equivalent to Figure 6(b).

—In Figure 7 there are two CTs: $\{t1,\ t2\}$ and $\{t1,\ t3\}$, and they are also full CTs. (We have left out the triggers and conditions here.)

—In Figure 8, $t1$ is an initial CT that must be accompanied by $t2$ to form a full CT.

—Since the transitions $t1$, $t2$, and $t3$ in Figure 9 must be executed together, $\{t1,\ t2,\ t3\}$ (but none of its subsets) is a CT. Note that this CT has two targets, whereas that of Figure 10 has two sources. A CT can thus have several sources and several targets, and we may refer to its source set and target set.

—Figure 11(a) depicts a more complex situation. In it, $t1$ and $t2$ must be executed together, and to lead to a full configuration they must be accompanied by $t5$ and by either $t3$ or $t4$. What we have are one initial CT, $\{t1,\ t2\}$, and two continuation CTs, $\{t5,\ t3\}$ and $\{t5,\ t4\}$, which form two full CTs, $\{t1,\ t2,\ t5,\ t3\}$ and $\{t1,\ t2,\ t5,\ t4\}$.

Figure 11 exhibits one of the very few situations in which STATEMATE's simulation tool behaves differently from the generated code. Consider a case in which the system is in state S, and $ev1$ and $ev2$ are generated, but both $C3$ and $C4$ are false. According to the semantics, Figures 11(a) and 11(b) are identical, and the simulation tool will therefore stay in S. It will, however, notify the user that it could not reach basic states after an initial CT was enabled. This message serves to warn the user that the generated code may behave differently from the simulation and that a potentially undesired (probably unintentional) behavior was discovered. As to the generated code, it will execute the initial CT, and only after reaching W will it discover that it cannot enter basic states; and at this point it will not "roll back" to S. It is theoretically feasible to generate code that would roll back under these circumstances, but we could not find an efficient way to do so. This issue seems to require more research.

—In Figure 12, the initial CT $\{t1,\ t2,\ t3\}$ must by accompanied by the two continuation CTs $t4$ and $t5$ in order to become a full CT.

The rest of the article concentrates on full CTs.

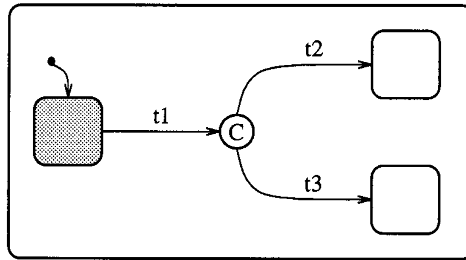

Figure 7.

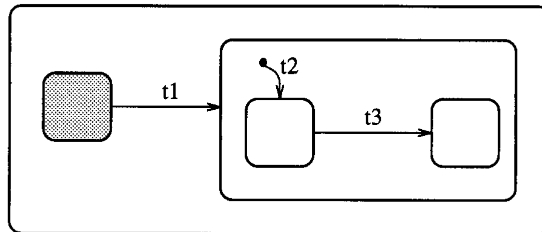

Figure 8.

5. DEALING WITH HISTORY

Statecharts feature two kinds of history connectors: H and H*. Here is the basic algorithm for interpreting history. Suppose we are executing a CT $t1$ whose target is a history connector h of state S. (Note that the graphical syntax of statecharts requires the connector to reside in some state's area, so that S cannot be an AND-state, which has no area of its own.)

if S has history
  then if h is an H connector
    then let S' be the substate of S which the system was in when most recently in S; $t1$ is treated as if its target is S'.
    else (h is an H* connector)
      let S' be the basic configuration relative to S which the system was in when it was most recently in S. $t1$ is treated as if its targets are all the states in S'.
  else (the system was never in S, or S's history was erased by a clear-history action since it was last in S)
      $t1$ is treated as if its target is S; however, if there are transitions emanating from h, then these have priority higher then those emanating from the default connector of S.

As an example, consider Figure 13, in which $t1$ is to be taken. If the system was in $B1$ when it was most recently in B, then we have the following: $t1$ is treated as if its target state was $B1$, and the full transition taken is $\{t1, t2\}$; whereas if the system was most recently in $B2$ the target state of $t1$ is taken to be $B2$, and the full transition taken is $\{t1, t3\}$. If the system was never previously in B, or if B's history was cleared since the

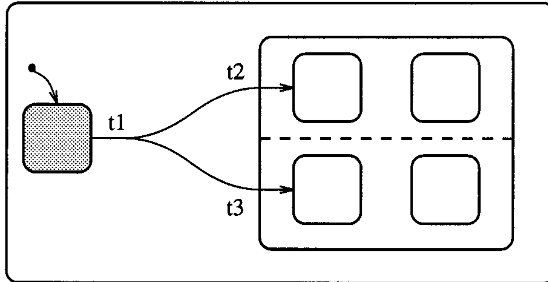

Figure 9.

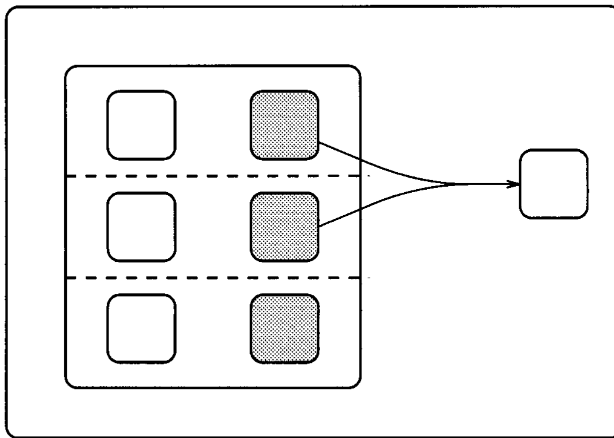

Figure 10.

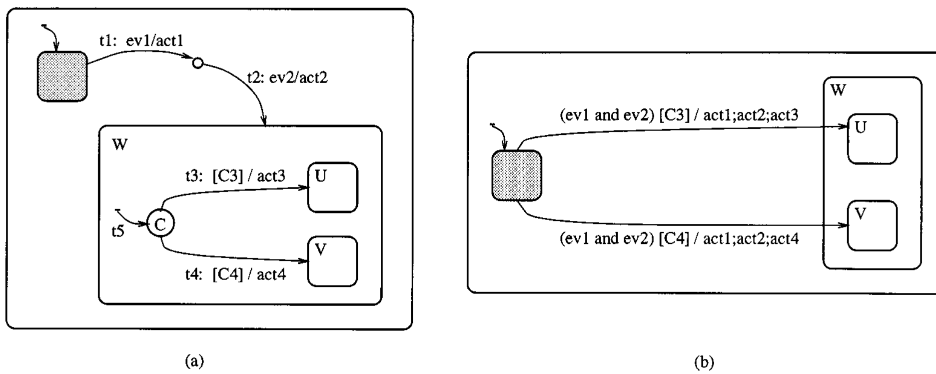

Figure 11.

system was last in B, then $t1$ is treated as if its target is B, and the full transition taken is $\{t1, t4, t2\}$. In this last case $t4$ is taken, in contrast to the case where $B1$ is entered by virtue of B's history, and hence any actions associated with $t4$ are executed.

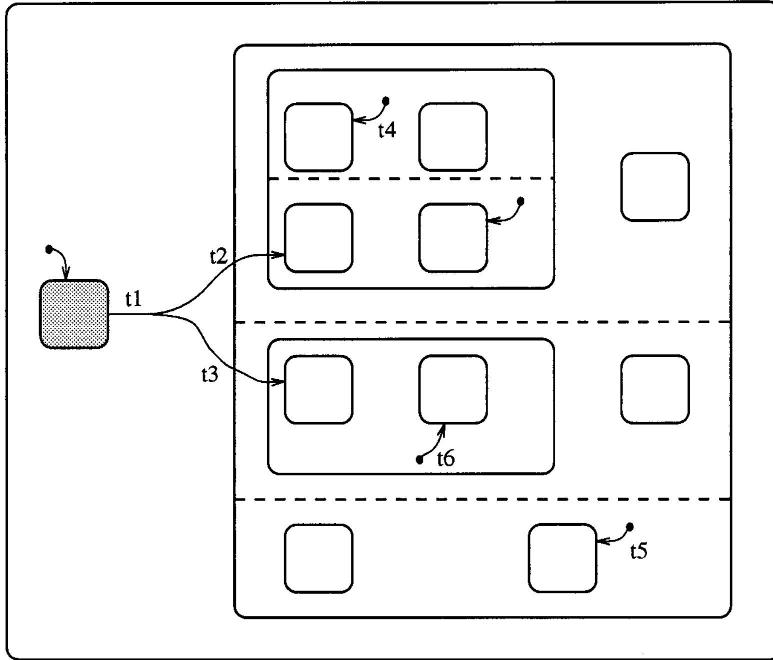

Figure 12.

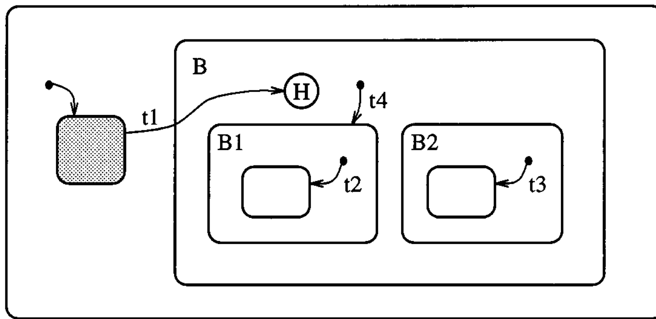

Figure 13.

The actions that erase the history of a state are *history-clear*(S), which applies to the history of S itself, and *deep-clear*(S), which has the same effect but applies to S and all of its descendant states too. A new entrance to S (and in the case of *deep-clear* the entrance to any descendent too) causes a new item of history information to be registered for subsequent entrances, capturing the substate of S presently entered.[5]

[5]The type of clear action that is taken has nothing to do with the type of history connector that is being cleared. When the history of state S is cleared by either action it applies to both the ordinary history connector H and the deep-history connector H*.

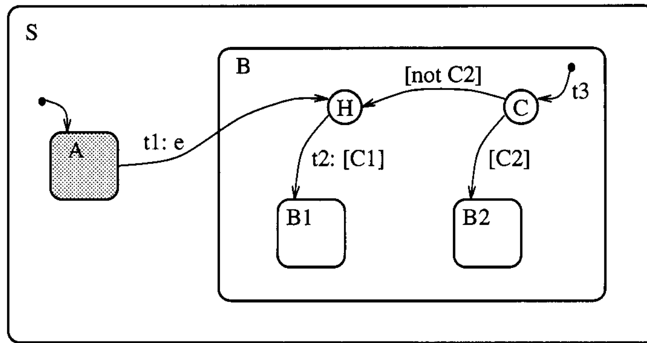

Figure 14.

Here is a more complicated situation, illustrating the priority given to transitions that emanate from history connectors. Consider Figure 14, and assume the system is in state A, that event e is generated, and that state B has no history. Upon the execution of $t1$, transition $t2$ gets priority over B's default entrance $t3$ and is executed. However, if $C1$ is false, the default $t3$ is attempted, which will succeed if $C2$ is true. However, if $C2$ is also false, it would appear that we are in a loop. STATEMATE will detect this, and both the simulation and generated code will issue an appropriate message.

A nice way of illustrating the fact that values of elements are changed only at the conclusion of a step is the following. Suppose that in the same step we enter a history connector drawn in state S and perform an action that clears the history of S, as in Figure 15 ($hc!$ abbreviates *history-clear*). Do we first clear the history of S (and then, lacking history, enter the default substate of S, which in Figure 15 is A), or do we first enter S via its history and only then clear the history? The answer is the latter. Recall, however, that once we have entered S we do not leave it again in the same step. This has the curious effect of making redundant the subsequent clearing of S's history, since this most recent entrance to S will register as its history even if the clearing is not performed.

6. THE SCOPE OF TRANSITIONS

Let us now be a little more precise about how full CTs are assembled. A full CT is really a collection of transitions, some of which are linked in a head-to-tail fashion by the rules governing the different connectors. If the result of these attachments contains an internal loop (i.e., one from connector to connector—not a loop from source state back to the same state, which is allowed, of course), the compound transition is illegal.

The maximality requirement on these attachments implies that the source of the full CT contains states only (no connectors), and its target contains basic states and termination connectors only. Otherwise the statechart is incomplete, since it presumably lacks one or more default arrows. For the CT to be legal, every two states in the source and every two states in the target must be mutually orthogonal (so that both the source

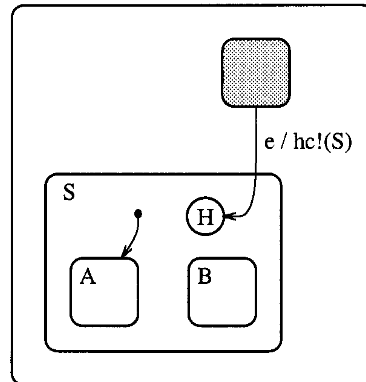

Figure 15.

and the target can be parts of legal basic configurations). In fact, the target set must be maximal: if it contains a descendant of a component of an AND-state, then it contains descendants of all of its other components too. A CT is said to be **enabled** in a step if at the beginning of the step the system is in all the states of its source set and if its trigger (i.e., the conjunction of all its components' triggers) is true. We should mention that empty triggers always evaluate to true.

As a simple example, consider Figure 16. If at the beginning of the step the system was in state A, and events e and f were generated during the previous step, $t1$ becomes enabled, but $t2$ does not. The reason is that although $t2$'s trigger f was true at the beginning of the step, and its source state B is entered during the step, the system was not in $t2$'s source state at the beginning of the step. As explained earlier, unless event f occurs again while the system is actually in B, the transition $t2$ will not be taken even in the next step, since the previous occurrence of f is lost.

In taking a transition from the source to the target, the CT will often pass through different levels of the statechart hierarchy. The question arises as to which nonbasic states are exited and entered in the process of taking a transition. This is important for several reasons, one of which is the set of actions that may be called for when exiting and entering states. For example, when executing $t1$ in Figure 17, do we exit and then reenter state A? When executing $t4$ in Figure 18, do we exit and then reenter state V? When executing $t6$ in Figure 19 do we exit and then reenter state U?

To answer these and similar questions we introduce the concept of the scope of a compound transition.[6] The **scope** of a CT tr is the lowest OR-state in the hierarchy of states that is a proper common ancestor of all the sources and targets of tr, including nonbasic states that are explicit sources or targets of transition arrows appearing in tr. History connectors that are targets of such arrows are represented by the states in which they

[6]In the previous reports related to this semantics (in particular, in the 1989 document "The Semantics of Statecharts"), we used the **lowest common ancestor** (LCA) instead of scope.

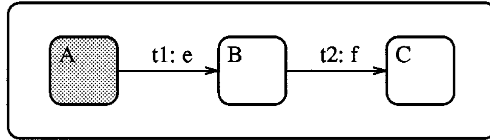

Figure 16.

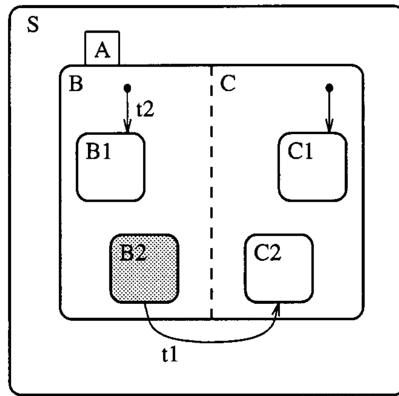

Figure 17.

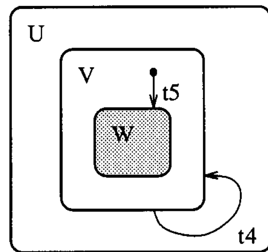

Figure 18.

are actually drawn. When the transition tr is taken, all proper descendants of its scope in which the system resided at the beginning of the step are exited,[7] and all proper descendants of that scope in which the system will reside as a result of executing tr are entered. Thus, the scope is the lowest state in which the system stays without exiting and reentering when taking the transition.

In Figure 17, for example, the scope of $t1$ is S (and not A, which is of type AND), and taking it implies exiting states $B2$, B, A, C, and either $C1$ or $C2$, depending on which one of them the system was in at the beginning of the step, and entering states A, B, $B1$, C, $C2$. State S is not exited. It is noteworthy that this would be the case even if $t1$'s arrow would have been

[7]Clearly, the system is in the transition's scope, and perhaps in other states too, before the step starts.

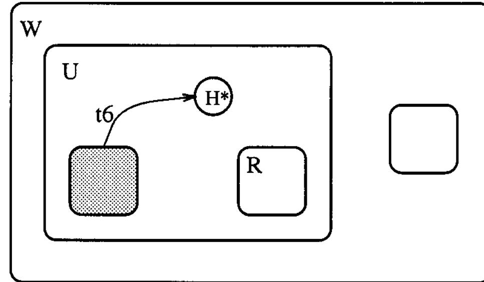

Figure 19.

drawn as exiting S's contour. S, and not S's parent, would have still been $t1$'s scope. The notion of scope does not depend on the way the arrow itself is drawn, but on its sources and targets only.

In Figure 18, the scope of $t4$ is U, and taking it implies exiting states W and V and entering states V and W, but U is not exited.

Finally, in Figure 19 the scope of $t6$ is W (since the history connector is represented by U), and taking it implies exiting and entering U. This might appear to be somewhat counterintuitive, since an arrow drawn entirely within a state can cause that state to be exited. However, again the way the arrow is drawn is unimportant, and the semantics in this case is a result of the special nature of history connectors, which must always be entered from outside the state.

A full CT with scope S always exits a legal configuration relative to one substate of S, and enters a legal configuration relative to potentially (but not necessarily) another substate of S.

7. CONFLICTING TRANSITIONS

We say that two CTs are in **conflict** if there is some common state that would be exited if any one of them were to be taken. In Figure 20, for example, $t1$ and $t2$ are in conflict because they would each imply exiting state A. Also, $t4$ is in conflict with all of $t1$, $t2$, and $t3$, since if and when $t4$ is taken, the system must have been in U and thus also in one of its substates. It follows that $t1$ and $t4$, for example, cannot both be taken in the same step.

The semantics treats these two kinds of conflicts differently. In the first case, if the triggers of both $t1$ and $t2$ are enabled at the beginning of a step, the system is faced with **nondeterminism**, since there is no reason to prefer one of them over the other. The situation is different when $t1$ and $t4$ are enabled in the same step. Here $t4$ has priority over $t1$ (and over $t2$ and $t3$), and there is no nondeterminism. Currently, priorities between transitions are determined outside-in, as we now describe, but this priority scheme is being revised for future versions of STATEMATE; see Appendix C.

Let tx and ty be two conflicting transitions, and let Sx and Sy be their scopes, respectively. Since the two transitions are in conflict, there must be

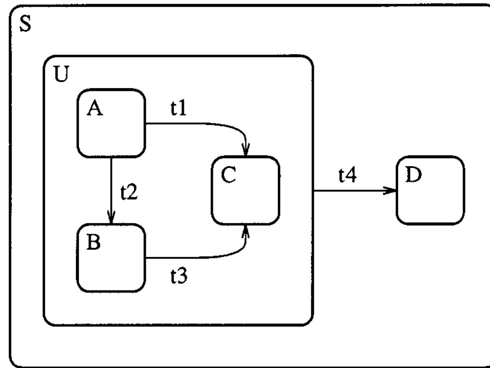

Figure 20.

a common state in their source sets, which implies that their scopes cannot be orthogonal or exclusive. Unless they are equal, one of the two scopes must be an ancestor of the other in the state hierarchy. Accordingly, priority is given to the transition whose scope is higher in the hierarchy. If $Sx = Sy$, neither one of tx and ty has priority over the other, and nondeterminism occurs. In Figure 21, for example, the scope of $t1$ and $t2$ is A; the scope of $t3$ is V; and the scope of $t4$ is U. Thus $t1$ and $t2$ have the same priority, and $t3$ has higher priority than $t1$ or $t2$. On the other hand, $t4$ is not in conflict with any of $t1$, $t2$, and $t3$.

In general, nondeterminism occurs when two or more conflicting CTs with the same priority are enabled in the same step, and no other conflicting CT with higher priority is enabled. In such a case, different steps are possible, which can lead to different statuses. Of course, at most one of these steps can be taken in any particular run.[8] Here is how the various STATEMATE tools deal with nondeterminism that arises during an execution. With the simulation tool STATEMATE waits for one of the possibilities to be selected. The selection can be carried out interactively by the user, or by specifying a selection criterion at the start. The dynamic tests tool will try out all the different possibilities in an exhaustive fashion. The code synthesized by the software code generator will select the first possibility it finds that is enabled and will proceed to execute it. The hardware code generated behaves similarly, but the user also has the option of asking that the code detect and report nondeterministic situations.

We should say a word about static reactions (SRs). As explained earlier, an enabled static reaction defined in state S is executed if the system was in S at the beginning of the step, but S was not exited by any CT during the step. In the present context, we might say that an SR defined in state S is in conflict with all the CTs that exit S or one of S's ancestors. Furthermore,

[8]Note that nondeterminism will occur even if both transitions lead to the same target state, since it is not only the resulting states that count; the two transitions might involve different actions.

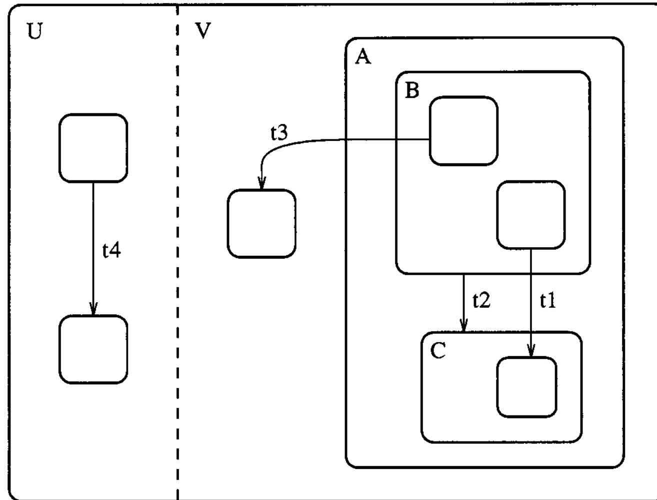

Figure 21.

CTs have higher priority than SRs, since if a state S is exited as a result of some CT, its SRs are not executed. Note that if S is exited and reentered in the same step, its SRs will not be executed in that step.

8. THE BASIC STEP ALGORITHM

In this section we present a schematic description of the algorithm that executes a single step. It lies at the heart of all the dynamic analysis tools of STATEMATE.

The Inputs

(1) The status of the system, which includes:
- —a list of states in which the system currently resides;
- —a list of activities that are currently active;
- —current values of conditions and data-items;
- —a list of events that were generated internally in the previous step;
- —a list of scheduled actions and their time for execution;
- —a list of timeout events and their time for occurrence;
- —relevant information on the history of states;

(2) The current time (see the discussion of time later).

(3) A list of external changes presented by the environment since the last step. These may include, for example, events that have occurred and changes in the values of conditions and data-items.

Comment. A scheduled action appears in the list as a pair $(a,\ next\text{-}a)$, where a appeared in an action expression of the form $sc(a,\ d)$ that has already been carried out, and *next-a* is the time at which the scheduled a is to be executed. Similarly, a timeout event E of the form $tm(e,\ d)$ appears

as a pair (E, *next-E*), where *next-E* is the time for the next generation of the timeout event E itself.

The Output

—A new system status.

The Algorithm

Stage 1: *Step Preparation*

(a) Add the external events to the list of internally generated events.

(b) Execute all the actions implied by the external changes. This includes changing the values of data-items, conditions, and activities, but not states.

(c) For each pair (a, *next-a*) in the list of scheduled actions, do the following:

if *next-a* $\leq$ *current-time*
then carry out a and remove (a, *next-a*) from the list.

(d) For each pair (E, *next-E*) in the timeout event list, with $E = tm(e, d)$, do the following:

if e is generated
then set *next-E* := *current-time* + d;
else if *next-E* $\leq$ *current-time* then
generate E and set *next-E* := ∞.

Comments

(i) The order of execution at this stage is important. Timeouts are evaluated last because we want to guarantee that $E = tm(e, d)$ cannot occur at the same step in which e occurs.

(ii) As a result of executing stage 1, the system status may change. This potentially includes data-items, conditions, activities, and new events. In the rest of the algorithm, the new status is used.

(iii) Logically we could use equality (i.e., =) for time comparison (and not $\leq$), but in practice, because of floating-point inaccuracies, and because in real life steps do take time, we use $\leq$.

(iv) It is important to note that if the delay d is an expression (i.e., not a constant) then it is evaluated with the values its operands have when the triggering event e is generated. In this sense, the interpretation of $tm(e, d)$ is not "e occurred last d time units ago," but rather it should be viewed as an operation, which, whenever e occurs, is recorded to be carried out in the future.

Stage 2: Compute the Contents of the Step. (At this stage, we mark the CTs and SRs to be executed.)

(a) Compute the set of enabled CTs.

(b) Remove from this set all the CTs that are in conflict with an enabled

CT of higher priority.

(c) Split the set of enabled CTs into maximal nonconflicting sets. (*Comment:* A set of CTs is nonconflicting if no two CTs in the set are in conflict. Being maximal means that each enabled CT not included in the set is in conflict with at least one CT that is included in the set. An example of nonconflicting sets is given below.)

(d) For each set of CTs, compute the set of enabled SRs defined in states that are currently active and are not being exited by any of the CTs in the set.

(e) If there are no enabled CTs or SRs
   then the step is empty
   else if stage (c) above resulted in a single set
      then this set constitutes the step
      else (i.e., stage (c) produced more than one set) we have nondeterminism, and any one of the sets can be chosen as the step.

(*Comment:* The simulation tool informs the user that nondeterminism was detected and will let him or her select the set to be executed. The generated code will select a set arbitrarily and upon specific request will also issue a message that nondeterminism was encountered.)

Stage 3: Execute the CTs and SRs. Let EN be the set of enabled CTs and SRs computed in stage 2.

(a) For each SR X in EN, execute the action associated with X.

(b) For each CT X in EN, let Sx and Sn be the sets of states exited and entered by X, respectively;

   update the history of all the parents of states in Sx;
   delete the states in Sx from the list of states in which the system resides;
   execute the actions associated with exiting the states in Sx;
   execute the actions of X;
   execute the actions associated with entering the states in Sn;
   add to the list of states in which the system resides all states in Sn.

Notes and Comments

(i) Figure 22 contains an example of nonconflicting sets (see item (c) in stage 2 of the algorithm). In it the various sr denote static reactions associated with the corresponding states. Assume that the system is in the configuration represented by the shaded basic states and that all the transitions and static reactions are enabled. Now, $t3$ has higher priority than $t1$ and $t2$; $t4$ and $t6$ have higher priority than $t5$; and the three transitions in the right-hand component have the same priority. Thus, in the left-hand component $t3$ will be taken, which implies that $sr1$ will be carried out in this step but not $sr4$. In the middle component $t4$ or $t6$ will be taken, and $sr2$ will be carried out; and in the right-hand component one of the three transitions will be taken, and $sr3$ will be carried out. More compactly, the maximal nonconflict-

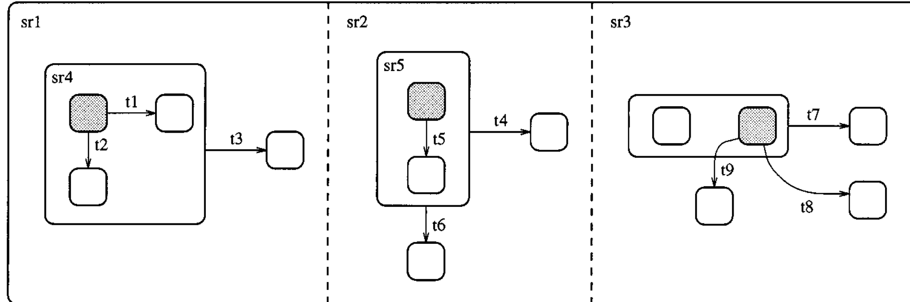

Figure 22.

ing sets are as follows, which means that there are six steps that can be taken:

$$\{t3, t4, t7, sr1, sr2, sr3\}$$
$$\{t3, t4, t8, sr1, sr2, sr3\}$$
$$\{t3, t4, t9, sr1, sr2, sr3\}$$
$$\{t3, t6, t7, sr1, sr2, sr3\}$$
$$\{t3, t6, t8, sr1, sr2, sr3\}$$
$$\{t3, t6, t9, sr1, sr2, sr3\}$$

(ii) In order to implement the semantics of a step, assignments are carried out in two stages. The first creates a list of pairs, each one of the form ⟨element, new-value⟩. Such a pair specifies that the element is to be assigned to the new-value at the end of the step. This guarantees that during the execution of the step the old values of elements are used, and thus the system is insensitive, as much as possible, to the order in which actions are executed. (For example, the semantics does not specify the order in which the CTs and SRs are executed in stage 7, nor does it specify the order in which the actions of a given CT or SR are executed.) Now, only after this first stage has been completed for all the actions executing in the current step, the second stage starts, in which the elements are assigned their new values.

When an element is assigned a new value more than once during a step, the last assignment is the one that counts. In such cases, which we refer to as **write-write racing**, the order of action execution can affect the results of the step. STATEMATE's simulation and dynamic tests tools are able to detect these cases. In Section 10 we discuss racing detection in more detail.

(iii) Actions performed on activities may cause chain reactions. When an activity is stopped or suspended, all its subactivities are stopped or suspended too. An activity that has not been endowed with a controlling statechart has default rules for its subactivities: when it is started, all its subactivities are started too, and when all its subac-

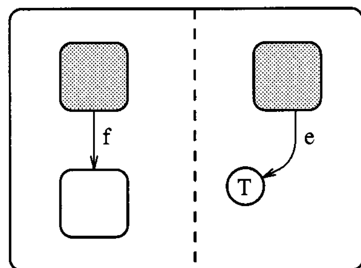

Figure 23.

tivities have stopped, the parent activity is stopped too. All these reactions are executed in the same step. An activity with control can activate its subactivities only following the step in which it was activated itself.

(iv) All the events generated in a step are collected in a new list of generated events, which is used as one of the inputs to the next step. The current list of events (containing those generated in the previous step) is used in step 7 to evaluate actions that have the form

when ⟨event⟩ **then** . . . **else** . . . **end when**

(v) An action a that is scheduled for later on in a scheduled action expression of the form $sc(a, d)$ is not executed in stage 7. Rather, it is added to the list of scheduled actions as the pair $(a, d + \mathit{current\text{-}time})$.

(vi) A termination connector is treated as a basic state, whose parent state is the one in which it is drawn. Upon entering a termination connector, however, no more steps are executed. For example, if event e occurs in Figure 23, the termination connector will be entered, and no more steps will be executed, even if f occurs later. Moreover, if the statechart in question serves as the control activity of activity P, then when execution enters a termination connector therein, activity P is stopped (at the end of the step), and the event sp (P) is generated.

9. TWO MODELS OF TIME

So far we have discussed the semantics of a single step and have skirted issues involving sequences of steps. One particularly important issue is that of time and its relationship to steps: when is the internal clock advanced relative to the execution of steps, and how long do steps take in terms of the clock?

STATEMATE supports two models of timing: synchronous and asynchronous. The **synchronous time model** assumes that the system executes a single step every time unit, reacting to all the external changes that occur in the one time unit that elapsed since the completion of the previous step. The **asynchronous time model** assumes that the system reacts whenever

an external change occurs, allowing for several external changes to occur simultaneously and, most importantly, allowing several steps to take place within a single point in time. Such a collection of steps is sometimes called a **superstep**. In both models, the very execution of a step may be viewed as taking zero time as far as the environment is concerned, since during the execution of the step itself no external changes have any effect, and it is as if time stops for the duration of the execution.

While the basic algorithm for a step given in the previous section is implemented in STATEMATE's simulation and dynamic tests tools, and in its various code generators, each of these executes a step under somewhat different circumstances, and the way the two models of time are reflected in the execution differs slightly among them. We shall concentrate on the way time is treated in the simulation tool and then briefly discuss its treatment in the other tools.

The synchronous time model fits systems that are highly synchronous. Assuming that the previous step was executed at time t, issuing a GO command during a STATEMATE simulation works as follows:

execute all external changes reported since completion of the previous step;
increment the clock by one time-unit;
execute all timeout events and scheduled actions whose due time falls inside $(t, t + 1]$ (that is, the interval that excludes t but includes $t + 1$);
execute one step.

The asynchronous time model (which we might call the greedy model) fits most kinds of asynchronous systems. In this time model, since the execution of steps does not advance the internal time of the simulation, the simulator's operator or environment must do so explicitly. There are several different GO commands that let the user control the advance of time during simulation.

The most important GO command is GO-REPEAT, which works as follows:

execute all external changes reported since completion of the previous step;
execute all timeout events and scheduled actions whose time is due (up to and including the current time);
repeatedly execute one step until the system is in a stable state (i.e., there are no generated events and no enabled CTs or SRs).

Note that GO-REPEAT does not increment the internal clock, so that many steps may be executed at the same point in time. The repeat loop in the above description is thus a superstep, namely, the series of steps that are executed in the course of a single GO-REPEAT command at the same point in time without external changes occurring between the constituent steps.

A GO-REPEAT command may result in an infinite loop. Suspected infinite loops are detected by the simulation tool. As an example, consider Figure 24, and assume that the conditions $C1$, $C2$, and $C3$ are false and that event e was generated by the environment. The GO-REPEAT command will cause the following four steps to be executed:

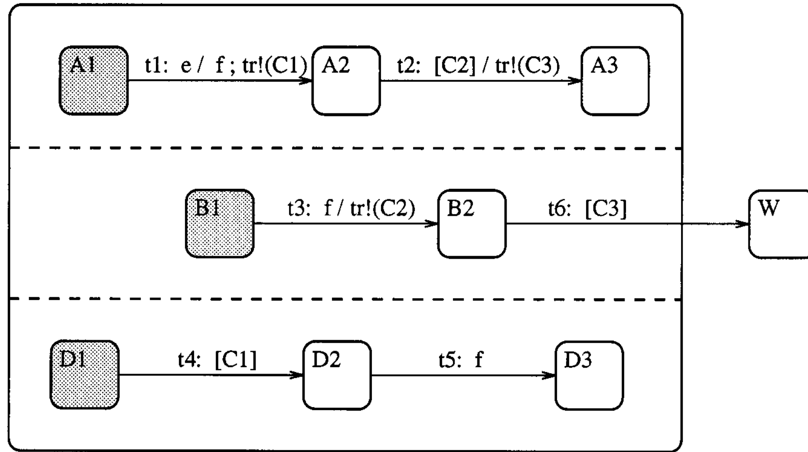

Figure 24.

(1) transition $t1$ is taken, leading to the basic configuration $\{A2, B1, D1\}$, making $C1$ true and generating f;

(2) transitions $t3$ and $t4$ are taken, leading to the basic configuration $\{A2, B2, D2\}$, and making $C2$ true;

(3) transition $t2$ is taken, leading to the basic configuration $\{A3, B2, D2\}$, and making $C3$ true. Note that $t5$ is not taken because event f is "alive" only during step (2); in step (3) it no longer exists.

(4) $t6$ is taken, leading to the basic configuration $\{W\}$.

The GO-ADVANCE command is designed to be used in conjunction with GO-REPEAT, to advance the clock. Before doing so, however, it must pay all "debts" that are due before the time planned for the advance. Suppose that the current time is t and that the command GO-ADVANCE n is used to advance the time to $t + n$. Here is what happens:

execute all external changes reported since the completion of the previous step;
set $t' := t + n$;
repeat the following until $t = t'$:
  execute all timeout events and scheduled actions whose time is due (up to and including the current time t);
  execute GO-REPEAT;
  set t'' to be the time of the closest scheduled action or timeout event;
  set t to the smaller of t' or t''.

The GO-REPEAT and GO-ADVANCE commands suffice to implement the greedy asynchronous time model. However, the following additional GO commands make life a little easier:

—GO-STEP: Executes one step without advancing the time. This command enables easier debugging of the model.

—GO-NEXT: Advances the clock to the time of the next timeout event or scheduled action without carrying out a step. Before the time is actually advanced, all the steps that can be executed are executed.

—GO-EXTENDED: A combination of GO-NEXT and GO-REPEAT, which is intended to execute the next superstep that really does something. It works as follows:

```
execute all external changes reported since the completion of the previous step;
if there are generated events or if there are enabled CTs or SRs
  then execute a superstep
  else advance the clock to the time of the next timeout event or scheduled
    action;
    execute the scheduled actions and timeout events whose time is due;
    execute a superstep.
```

This completes our discussion of the treatment of time in the simulation tool of STATEMATE. In the dynamic tests tool, external changes are sensed only once every fixed period of time in both time models, but the user has the option of injecting external changes between steps of a superstep too.

STATEMATE's hardware code generators let the user select between two code styles, each compatible with a different time model. The **RTL code style** means generation of VHDL or Verilog code in which the mechanism that determines when a step should execute is sensitive to one event only: the rising or falling edge of a clock. This is similar to STATEMATE's synchronous mode. On the other hand, HDL code generated with the **behavioral code style** reacts to any change in the inputs the moment they occur. This is identical to STATEMATE's asynchronous mode.

The software code generators generate one style of code; however, two different schedulers are provided that support different time models. One of them uses the CPU clock time. This has the effect that steps, and therefore supersteps, take more than zero time. As in the simulation, external changes are sensed only at the start of a step. Since steps take more than zero time, however, external changes, timeout events, and scheduled actions may occur before the system has stabilized. In other words, the synchrony hypothesis (see Berry and Gonthier [1992] and Pnueli and Shalev [1991]) is not preserved. This also means that with the CPU clock time model, the equivalent of STATEMATE's simulator's GO-REPEAT, is not supported.

Another scheduler uses a simulated clock. The simulated clock advances only after the system is in a stable status. External changes, timeout events, and scheduled actions occur only when the system is stable. The simulated clock time model guarantees a behavior which is identical to STATEMATE's asynchronous mode.

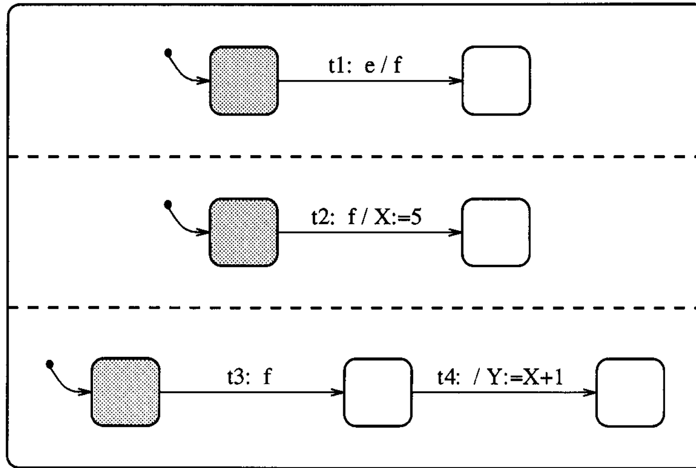

Figure 25.

10. RACING CONDITIONS

Racing conditions arise when the value of an element is modified more than once or is both modified and used at a single point in time. Now, since our approach in the greedy model is that multiple steps can be executed at (what appears to be) the same point in time, racing problems can arise not only within a step, but also between transitions or actions that are executed in different steps. Thus racing must be checked within supersteps, not only within single steps, and we must take into account causality dependencies between the transitions that are carried out in different steps within a single superstep.

For example, suppose a transition labeled "e / f; X := 5" is executed in some step, enabling another transition in the same superstep labeled "f / X := 6." Here, although X is modified more than once at the same point in time, there is no racing problem, since the second transition necessarily comes after the first. Thus, while the steps in a superstep are executed in zero time, they are considered to be executed in order.

Let us be a little more precise about how racing is defined. In each step and superstep several transitions may be enabled. The enabled transitions are viewed as executing in some implementation-sensitive order. However, this order must be consistent with the enabling order, in the sense that each transition is to be executed after the ones that enabled it. A race situation is one in which, had we executed the enabled transitions in a different order (yet a legal one according to the above criteria), we might have obtained different results in one or more of the data-items or conditions.

Consider Figure 25. When event e occurs, the semantics prescribes that transition $t1$ will be taken in the first step, $t2$ and $t3$ in the second step, and $t4$ in the third step. Consequently, X will attain the value 5 in the second step, and Y will attain 6 in the third step. However, notice that the

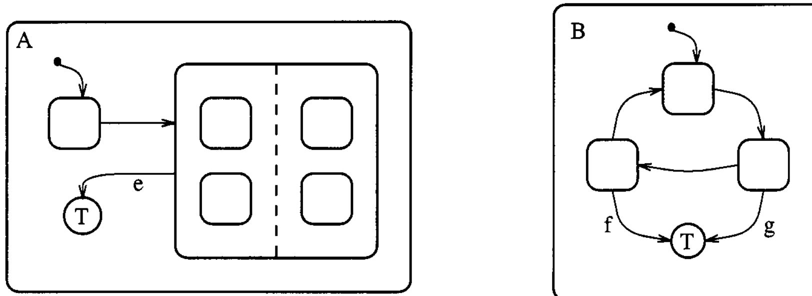

Figure 26.

only constraints on the ordering of these transitions are that $t1$ must come before $t2$ and $t3$, and that $t3$ must come before $t4$. It is only the greediness of our semantics that causes $t2$ and $t3$ to be taken in the same step. There is actually nothing wrong with a semantics that would postpone taking $t2$ until after both $t3$ and $t4$ are taken (in two consecutive steps), but this may result in a different value in Y. Cases like this will also cause STATEMATE to report a racing condition.

11. MULTIPLE STATECHARTS

The semantics as described applies in all its aspects to several statecharts running simultaneously, which in STATEMATE occurs when they represent activities that happen to be active concurrently.

Multiple active statecharts are treated basically as orthogonal components at the highest level of a single statechart, except that when one of the statecharts becomes nonactive (by entering a termination connector, or when the activity it controls is stopped) the other charts continue to be active. When an activity is restarted, the statechart controlling it is reactivated by reentering its default configuration.

Consider, for example, the two charts of Figure 26. They are treated essentially as the single chart of Figure 27. Notice that entering a termination connector in the original chart is just like entering the special idle state in the corresponding orthogonal component of the new chart. The special *start* events leading out of these idle states depict the events that cause the original statechart to start, such as *started*(*activity*) for the activity that is controlled by the statechart in question.

In the asynchronous time model, the steps in multiple statecharts are carried out in synchronization, just as they would have if they were represented as orthogonal components of one big statechart. Thus, what is really happening is that the greediness property applies at the system level too, meaning that not only are all enabled transitions in a single chart executed simultaneously, but so are those in all of the charts, and therefore all the transitions in the entire system are executed simultaneously. In the synchronous time model, on the other hand, each chart may have its own clock, telling it when to execute a step. And since each of these may be

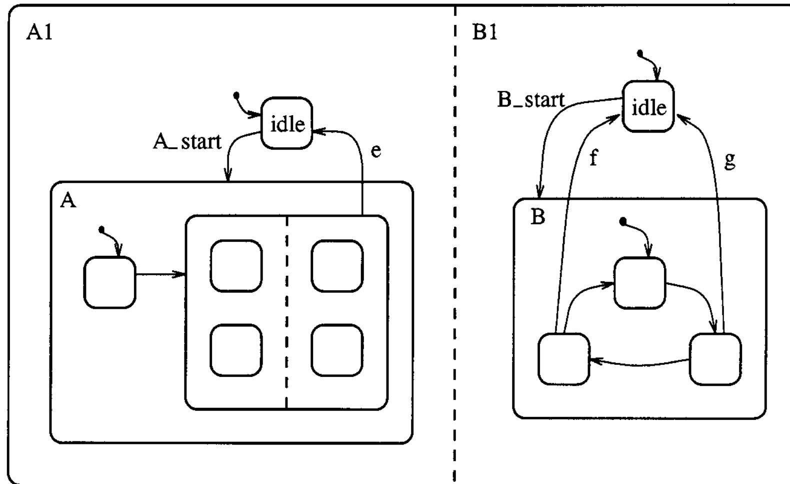

Figure 27.

specified to have a different time unit (i.e., a different clock rate), what is happening here is that within a chart all the orthogonal components remain synchronized, but each chart is synchronized with its own clock and not necessarily with the other charts. In both time models, the time units used for evaluating time expressions (in timeout events and scheduled actions) may be defined differently for each chart. Thus, for example, if two actions *schedule*(a, n) and *schedule*(b, n) are executed in the same step in the statecharts A and B of Figure 26, respectively, actions a and b might not be carried out at the same time, since the time units for evaluation in A and B could have been defined differently.

APPENDIX

A. COMPARISON WITH OTHER WORK

This appendix is devoted to issues regarding the comparison of our semantics with other proposed semantics for statecharts.

We have already mentioned the fundamental question as to whether changes that occur in a given step (e.g., generated events and value updates) take effect in the current step or in the next one. Our semantics adopts the latter, and this is in contrast to the semantics of Harel et al. [1987] and Pnueli and Shalev [1991]. We shall thus say no more about these two papers.

The most natural candidate for comparison is the RSML language of Leveson et al. [1995], for various reasons. First, it was some of the comments in Leveson et al. [1995] that prompted us to publish the present article; second, as in our case, the semantics of Leveson et al. [1995] is claimed to be supported by a tool; third, despite a difference of around seven years in the conception of the two semantics, most of the underlying

principles of the two are very similar, such as the fact that changes take place in the next step.

The main differences between the two approaches are syntactical. For example, RSML does not support the history connector; it features directed communication over channels instead of broadcast communication; it allows AND/OR tables for entering condition expressions; it does not support disjunctions of trigger events, and more.

Although there appear to be no significant differences in the dynamic semantics, we use different terms. Our steps are RSML's microsteps, and our supersteps are RSML's steps. For us, a step is the basic atomic operation, whereas a superstep is something that STATEMATE users can compose in commands to the simulator and as an option for the generated code. From the description in Leveson et al. [1995], it appears that their step (our superstep) is the basic atomic operation. One of the reasons that it is somewhat difficult to talk more precisely about the semantics of RSML is that many issues that we discuss at length are ignored in Leveson et al. [1995], such as how CTs are built, which are the legal configurations, etc.

We now turn to the survey in von der Beek [1994]. It lists 19 issues relevant to proposals for the semantics of statecharts.[9] Since the implemented semantics of statecharts as described in this article and its 1989 precursor are not mentioned in von der Beek [1994] at all, and since these 19 issues are used in von der Beek [1994] to catalog other published semantics, we feel that the best thing for us here is to comment on how our approach deals with each of these issues. In the following, we list each of the issues by number and title (similar to von der Beek [1994]), give a one-sentence explanation thereof (often adapted from the phrasing in von der Beek [1994]), followed by a short discussion of the way our language and its semantics relates to it. We leave to the reader the task of consulting von der Beek [1994] for a more detailed explanation of, and motivation for, the issues themselves. We should mention, however, that (i) some of them are questions about which features the language supports (note that the approaches surveyed in von der Beek [1994] have a varying syntax too) and (ii) the semantic aspects of most of them are relevant only to supersteps.

1. *Perfect-Synchrony Hypothesis.* The perfect-synchrony hypothesis asserts that external events are responsed to immediately.

The STATEMATE semantics supports two time models: the asynchronous and the synchronous. In the asynchronous model, the system reacts in zero time the instant it is presented with an external change. Moreover, when the system reaction is stretched over several steps, with one event triggering another, all these steps execute in zero time. Therefore, under this time model, the perfect-synchrony hypothesis holds.

In the synchronous time model, the system reacts whenever it detects an occurrence of one special event (usually the edge of a clock cycle). Hence, it

[9] They are referred to in von der Beek [1994] as "problems" with the way statecharts are described in Harel [1987].

is possible that nonzero time passes between the instant in which an external event occurs and the time the system reacts to it. In this model, if the reaction requires several steps, each step executes only when the special event is sensed. Hence, the time it takes to react depends on the number of steps needed to complete the system's reaction, and the hypothesis does not hold.

2. *Self-Triggering, Causality.* Self-triggering involves events that are executed based on their own actions and are not caused by an external event; causality means that each transition taken was caused (perhaps indirectly) by an external event.

This is not really an issue in our semantics, because internal events are sensed only in the step following the one in which they were generated. Thus, in a rather trivial sense one might say that STATEMATE respects causality.

3. *Negated Trigger Event.* A negated event is one that models the nonoccurrence of an ordinary event.

Our approach supports negated trigger events, which are allowed by the syntax and are covered by the semantics. STATEMATE's simulator supports them in full. As to the code generator, experience shows that virtually none of its users were employing negated events unless they were AND-ed with positive events. For this reason, the code generator was set up so that its default mode supports negated trigger events only when AND-ed with positive events. Thus, for example, *not E* is not supported; *F and not E* is supported; and *F or not E* is not supported. This expedites the execution of the generated code by about 20%. However, a nondefault scheduler is available to the user, which supports general negated trigger events.

4. *Effect of Transition Executing is Contradictory to Its Cause.* An example of this is an event triggering an action that is essentially its negation.

This is not an issue in our approach, since the effect of a trigger event is sensed only in the following step. In particular, STATEMATE has no problem with a transition labeled *not E/E*.

5. *Interlevel Transition.* A transition crossing the borderlines of states. STATEMATE supports these transitions in full.

6. *State Reference.* This is the ability to refer to the states of an orthogonal component in deciding whether a transition is to be taken.

STATEMATE supports this in full. In some cases, the main output of a statechart is the set of states it is in. The hardware language code generators also support adding states to the output ports.

7. *Compositional Semantics, Self-Termination.* Compositional semantics defines the meaning of a construct based on the meanings of its constituent constructs; by self-termination, von der Beek [1994] means that stratified transitions are used to replace interlevel ones, to aid in making semantics compositional.

Our semantics is not defined in a compositional manner. In fact, neither is the syntax, which allows interlevel transitions.

8. *Operational versus Denotational Semantics.* Our semantics is operational.

9. *Instantaneous State.* A state that can be entered and exited simultaneously (e.g., with an action generated upon entrance that also causes the state to be exited).

In our semantics instantaneous states are prohibited in a step. However, the assertion in von der Beek [1994], to the effect that this implies that "at one instant of time several transitions may only be executed if they reside in parallel components" is inaccurate, since in the asynchronous time model several steps may execute at the same instant of time.

10. *Durability of Events.* This issue concerns the question of whether an event is available for more than "an instant of time."

In our semantics, an event is sensed only in the step following the one in which it was generated. Even when multiple steps execute at the same instance of time, an event generated in step n is sensed only in step $n + 1$.

11. *Parallel Execution of Transitions.* This issue concerns the question of whether transitions allowed to execute simultaneously must reside in different orthogonal components.

In our approach, only transitions whose scopes are mutually orthogonal can execute in one step.

12. *Transition Refinement.* This issue concerns the difference between a single transition and its refinement into a sequence of transitions (during a top-down development process, for example), where the latter might take longer than the former.

In the asynchronous time model of our semantics this is not a problem, since several steps may execute at the same instance of time, and the refinement will be equivalent to the refined transition. In the synchronous time model, however, the difference does exist.

13. *Multiple Entering or Exiting of an Instantaneous State.* This concerns the possibility of an infinite loop caused by the repeated entering and exiting of an instantaneous state.

Within one step of our semantics this is not an issue, since instantaneous states are not supported (see issue number 9). However, this could happen within a superstep.

14. *Infinite Sequence of Transition Executions at an Instant of Time.* Similar to number 13.

15. *Determinism.* This is concerned with whether a statechart's behavior can exhibit nondeterministic choices.

One can model nondeterministic behavior in our statecharts. STATEMATE's code generators treat it as underspecification or "don't care," and the distinction is up to the user.

16. *Priorities for Transition Execution.* This feature provides a way for the model to determine which transition from among a set of conflicting transitions that exhibits nondeterminism is to be executed.

In the current versions of STATEMATE, priorities of transitions are determined by the scope of the transition. The priority scheme that will be implemented in future versions of STATEMATE is based on the transition's source and is described in Appendix C.

17. *Preemptive versus Nonpreemptive Interrupt.* A preemptive interrupt is a high-level transition that prevents taking transitions on lower, encapsulated levels.

STATEMATE supports preemptive interrupts with priorities given to higher-level transitions. (The priority concept relevant to this issue is insensitive to the difference between the new and old versions of priorities in STATEMATE that are discussed in Appendix C.)

18. *Distinguishing Internal from External Events.* This concerns the question of whether there is a difference in sensing internal versus external events in the microsteps that make up a macrostep, with relevance to the possibility of satisfying the synchrony hypothesis.

In our semantics, internal events are indeed sensed within a superstep, but externally generated ones are not.

19. *Time Specification, Timeout Event, Timed Transition.* These concern the issue of when time actually progresses and whether there is support for timeout events and timed transitions (the latter are transitions bounded in time from below and/or from above).

In our semantics, time can advance only when the system is in a stable status, meaning that no transitions can execute without an occurrence of an external event. Our statecharts support timeout events but not timed transitions.

If there were a column for our semantics in the table given in von der Beek [1994, p. 145], it would read as follows:

$$g, +, +, -, +, +, +, +, +, +, \circ, -, +, -, t, d, +, +, -, +, +, -, +, +, -, d$$

B. COMBINATIONAL ASSIGNMENTS

Combinational assignments (CAs) are supported by both the software and hardware versions of STATEMATE. They are not associated with states, transitions, or activities, but with an entire chart. They can be viewed as a continuous implementation of a function, i.e., a function that is computed automatically immediately after a change in any one of its parameters. The syntax of a CA is similar to that of a regular kind of assignment and may also contain **when-else** clauses.

This appendix concentrates on specifying when and how CAs are executed. A CA is executed immediately following a change in one of the operands of its assigned expression (i.e., the right-hand side of the assign-

ment). Put another way, during any step in which an operand is changed, either externally or internally, all the relevant CAs are executed. For models that contain CAs, the step algorithm of Section 8 is modified so that following stage 3 a new stage is carried out in which the CAs are executed. This is done in phases, so that in each phase, all the CAs whose operands have changed are executed. First, the right-hand-side expressions of all the participating CAs are evaluated, and then the assignments are carried out simultaneously, in such a way that if an element X is changed by some CA in phase n, but X is an operand of some other CA, then the new value of X is available only in phase $n + 1$.

Loops in the definition of CAs are legal, which implies the possibility of infinite loops. For example, consider the following two CAs:

$$A := 5 \textbf{ when } B = 3 \textbf{ else } 10$$

$$B := 3 \textbf{ when } A > 5 \textbf{ else } 4$$

Here A and B will never stabilize. This is an error in the specification, and STATEMATE's analysis tools will detect the infinite loop at run time.

If an element X changes its value several times within one step, but ends up with the original value after all the phases of executing the CAs are over, it is as though its value never changed, and in particular, the event *changed*(X) (abbreviated $ch(X)$) is not generated. For example, consider the following two CAs:

$$X := A + B$$

$$A := A1$$

Suppose that at the beginning of the step the values were $A1 = 5$, $A = 5$, $B = 5$, $X = 10$. Now suppose that during the execution of a transition, $A1$ changed to 3, and B changed to 7. The values of the elements will be modified as follows: in phase 1, X becomes 12, and A becomes 3; and in phase 2, X returns to its old value of 10. The event $ch(X)$ will not be generated.

This semantics is implemented in all STATEMATE tools, with one exception. When generating code for hardware (in VHDL or Verilog) we have decided to implement the STATEMATE CAs as CAs in the target hardware language. When generating hardware code, the user has a choice of generating behavioral-style code or RTL-style (register transfer level) code. For our purposes, behavioral style is asynchronous, and RTL style is synchronous. Here is the difference: each phase in the execution of CAs takes some "delta," which in hardware description languages is like a step that takes no time. In RTL style, several deltas can be added after the execution of the "real" step (before the next clock signal arrives) so that indeed all phases of the CAs are executed as part of the step. On the other hand, in the behavioral style, transitions can execute the moment they are

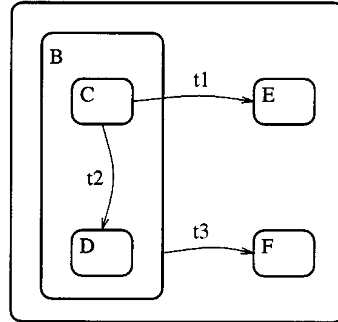

Figure 28.

enabled. Since we could not find an elegant way to postpone transition execution until all the CAs have completed, in the behavioral style CAs can be carried out in the same step as the transitions that follow the step that changed their operands. Thus, if we have a transition that changes the value of X, and a CA of the form $Y := X$, then if X changes value in step n, then Y will change value in step $n + 1$, when some other transition might be taken.

C. PRIORITY OF TRANSITIONS

Two aspects are relevant to the determination of the priority of transitions in STATEMATE. The **structural priority** (SP) of a compound transition and the **priority numbers** (PN) associated with transition segments.

Structural priority is the main criterion for determining the overall priority of a transition. If transition $t1$ has higher SP than $t2$ then $t1$ has an overall priority higher than $t2$. SP is defined for initial CTs only. It is determined by a particular state associated with the CT, which we call its **structural priority determinant**. The higher the determinant of a CT in the hierarchy of states, the higher its SP. The determinant of a CT is determined by its sources and targets. In the current versions of STATEMATE, the determinant of an initial CT tr is defined to be its scope. (Recall that the scope of tr is the lowest OR-state in the hierarchy of states that is a proper ancestor of all the sources and targets of tr.) In future versions of STATEMATE, the determinant will be based primarily on the explicit source(s) of tr. Let the source set of tr be denoted by R and its scope by S. The structural priority determinant of tr is defined as follows:

```
let st be a state in R (st must be a proper descendant of S);
  let L be the set of all states that are both proper ancestors of st and proper
    descendants of S (L may be empty);
  if there are no AND-states in L
    then (R must be a singleton) the determinant of tr is st
    else the determinant of tr is the highest-level AND-state in L
```

Consider, for example, Figure 28. The current versions of STATEMATE

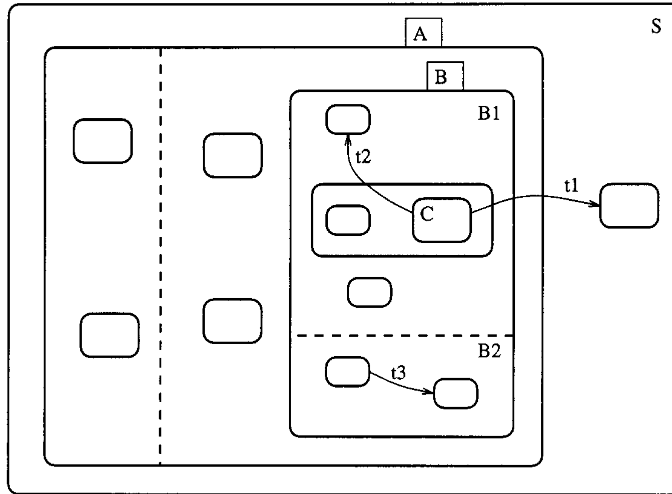

Figure 29.

give $t1$ and $t3$ the same priority, higher than that of $t2$, whereas the future versions will give $t1$ and $t2$ the same priority, lower than that of $t3$. In Figure 29, the determinant of $t1$ is A—the highest-level AND-state that is exited by $t1$—and the determinant of $t2$ is C. Thus, although $t1$ and $t2$ have the same source, $t1$ will have higher priority than $t2$ in the future versions too.

The reason the determinant of $t1$ is A is the following: $t1$ and $t3$ are in conflict and therefore cannot be executed in the same step. If we were to leave the determinant of $t1$ to be C, there would be no way to specify which of $t1$ or $t3$ has higher priority. In addition, the orthogonal components of an AND-state would no longer be truly orthogonal, since each one has to examine what is happening inside the other one. And most importantly, the generated code would become very inefficient.

Note that this definition suffices, because transitions whose determinants are neither identical nor mutually ancestoral are never in conflict, so that, indeed, the higher the determinant of a CT in the state hierarchy, the higher its SP.

The main reason for the change being made in the definition of priority is that the new one is more intuitive; it seems to be preferred among most STATEMATE users. Also, with the new definition one can endow a transition with a higher SP without changing its meaning. In Figure 30, for example, $t1$ and $t2$ are equivalent under the current semantics, but the new semantics gives $t1$ higher priority than $t2$, although their behavior is the same.

The **priority number** (PN) is an integer one can affix to transition segments. The priority of a transition segment is compared with the priorities of other segments with the same source (state or connector) or the same priority determinant. Let X be a compound transition composed of the transition segments $X1, \ldots, Xn$ in order. Thus, Xi's source is $X(i - 1)$'s

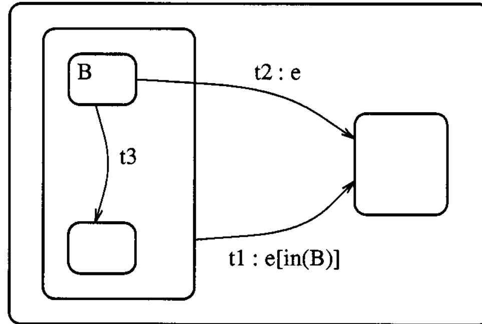

Figure 30.

target, for $i = 2, \ldots, n$. Similarly, let Y be a CT composed of $Y1, \ldots, Ym$ in order. Suppose now that the structural priority determinants of X and Y are the same, so that we have to compare their priorities according to their PNs. Here is the comparison procedure:

```
i := 1
while Xi = Yi do i = i + 1
  (i is now the index of the first different pair of segments)
if PN(Xi) > PN(Yi)
  then return "X has higher priority than Y"
  else if PN(Xi) < PN(Yi)
    then return "Y has higher priority than X"
    else return "X and Y have the same priority"
```

As an example, consider Figure 31, in which the priority numbers are parenthesized. The CT leading to state $T1$ has higher priority than the CTs leading to $T2$, $T3$, and $T4$ and has the same priority as the CTs leading to $T5$ and $T6$. The table below summarizes the priority relationships between the CTs (represented by their target states):

	$T2$	$T3$	$T4$	$T5$	$T6$
$T1$	$>$	$>$	$>$	$=$	$=$
$T2$		$>$	$>$	$=$	$=$
$T3$			$=$	$=$	$=$
$T4$				$=$	$=$
$T5$					$<$

Interestingly, the priority relationship, as we define it, is not transitive. For example, while the pairs $(T5, T1)$ and $(T1, T6)$ have equal priorities, that of $T6$ is greater than that of $T5$. This does not pose any problem, however: whenever several transitions with the same determinant are enabled, those whose priority is not maximal are not candidates for execution. In other words, if a CT X is enabled, and there is another enabled CT Y with greater priority than X's, then X will not be a candidate

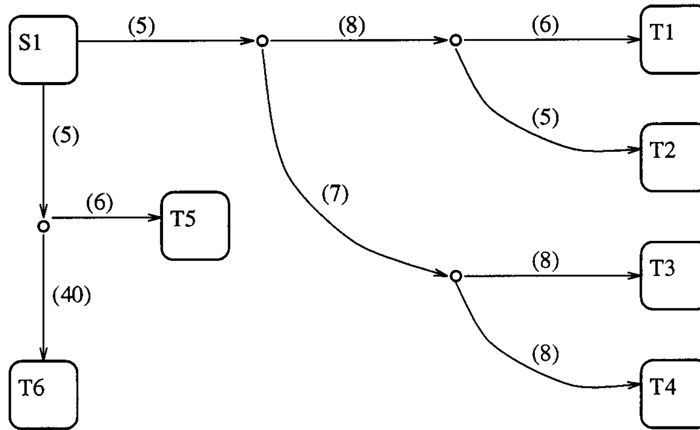

Figure 31.

for execution. If in Figure 31 all the transitions are enabled, there are actually two possible steps: those leading to *T1* and *T6*.

When a CT contains segments connected by joint or fork connectors, it is not always possible to arrange them in linear order, with one's target being the next one's source. In such cases, we view each combination of segments incident to the connector—some incoming and some outgoing—as a single virtual segment. The properties of the virtual segment are defined, using those of its constituent (actual or virtual) segments, as follows:

—its priority is the maximum of those of the constituent segments;
—its source set consists of the sources of those of the constituent segments, excluding the fork/joint connector;
—its target set consists of the targets of those of the constituent segments, excluding the fork/joint connector.

If necessary, this definition is carried out recursively, until all the fork and joint connectors have been removed. Only then can the actual and virtual segments be linearly ordered. The CT depicted in Figure 32, for example, will be regarded as having three segments: one consisting of $t1$, $t2$, $t3$, $t4$, and $t5$, one consisting of $t6$, and one consisting of $t7$, $t8$, and $t9$.

ACKNOWLEDGMENTS

The basic concepts of the semantics described here were developed between 1984 and 1986 during lengthy sessions and deliberations by a group of people that included, besides ourselves, Amir Pnueli, Michal Politi, Jeanette Schmidt, and Rivi Sherman. This article would not have been possible without them. We also wish to thank other members of the STATEMATE development team, who contributed to the updating and revision of the semantics over the years. These include Moshe Cohen, Eran Gery, Aharon Shtull-Trauring, and Mark Trakhtenbrot. Special thanks to

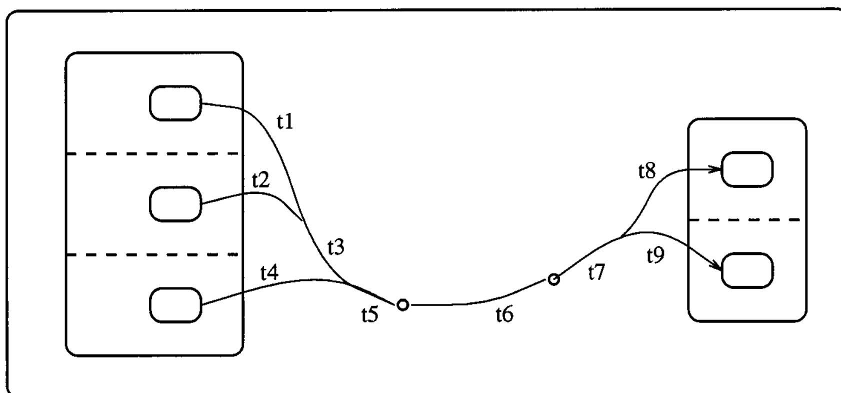

Figure 32.

Amir Pnueli, Jim Armstrong, Philip Brookes, Nick Cropper, Paul Urban, and the reviewers, for comments on a previous version.

REFERENCES

BERRY, G. AND GONTHIER, G. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program. 19,* 87–152.

HAREL, D. 1987. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program. 8,* 231–274. Preliminary version available as Tech. Rep. CS84-05, The Weizmann Inst. of Science, Rehovot, Israel. Feb. 1984.

HAREL, D. AND PNUELI, A. 1985. On the development of reactive systems. In *Logics and Models of Concurrent Systems,* K.R. Apt, Ed. NATO ASI Series, vol. F-13. Springer-Verlag, New York, 477–498.

HAREL, D. AND POLITI, M. 1996. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach.* To be published. Preliminary version available as Tech. Rep., i-Logix, Inc., Andover, Mass. Titled "The Languages of STATEMATE."

HAREL, D., LACHOVER, H., NAAMAD, A., PNUELI, A., POLITI, M., SHERMAN, R., SHTULL-TRAURING, A., AND TRAKHTENBROT, M. 1990. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Trans. Softw. Eng. 16,* 403–414. Preliminary version appeared in *Proceedings of the 10th International Conference on Software Engineering.* IEEE Press, New York, 1988, pp. 396–406.

HAREL, D., PNUELI, A., SCHMIDT, J. P., AND SHERMAN, R. 1987. On the formal semantics of statecharts. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science.* IEEE Press, New York, 54–64.

HUIZING, C. AND DE ROEVER, W. P. 1991. Introduction to design choices in the semantics of statecharts. *Inf. Process Lett. 37,* 205–313.

HUIZING, C., GERTH, R., AND DE ROEVER, W. P. 1988. Modeling statecharts behavior in a fully abstract way. In *Proceedings of the Colloquium on Trees in Algebra and Programming.* Lecture Notes in Computer Science, vol. 299. Springer-Verlag, New York, 271–294.

KESTEN, Y. AND PNUELI, A. 1992. Timed and hybrid statecharts and their textual representation. In *Formal Techniques in Real-Time and Fault-Tolerant Systems,* J. Vytopil, Ed. Lecture Notes in Computer Science, vol. 571. Springer-Verlag, Berlin, 591–619.

LEVESON, N. G., HEIMDAHL, M. P. E., HILDRETH, H., AND REESE, J. D. 1995. Requirements specification for process-control systems. *IEEE Trans. Softw. Eng. 20,* 684–707.

MARANINCHI, F. 1992. Operational and compositional semantics of asynchronous automaton compositions. In *Proceedings of CONCUR '92.* Lecture Notes in Computer Science, vol. 630. Springer-Verlag, Berlin, 550–564.

PNUELI, A. AND SHALEV, M. 1991. What is in a step: On the semantics of statecharts. In *Proceedings of the Symposium on Theoretical Aspects of Computer Software.* Lecture Notes in Computer Science, vol. 526. Springer-Verlag, Berlin, 244–264.

VON DER BEEK, M. 1994. A comparison of statechart variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems,* L. de Roever and J. Vytopil, Eds. Lecture Notes in Computer Science, vol. 863. Springer-Verlag, New York, 128–148.

Received November 1995; revised February 1996; accepted July 1996