The Rhapsody Semantics of Statecharts
(or, On the Executable Core of the UML) *

(Preliminary Version)

David Harel and Hillel Kugler

Department of Computer Science and Applied Mathematics
The Weizmann Institute of Science, Rehovot, Israel
{dharel,kugler}@wisdom.weizmann.ac.il

Abstract. We describe the semantics of statecharts as implemented in the current version of the Rhapsody tool. In its original 1996 version this was among the first executable semantics for object-oriented statecharts, and many of its fundamentals have been adopted in the Unified Modeling Language (UML). Due to the special challenges of object-oriented behavior, the semantics of statecharts in Rhapsody differs from the original semantics of statecharts in Statemate. Two of the main differences are: (i) in Rhapsody, changes made in a given step are to take effect in the current step and not in the next step; (ii) in Rhapsody, a step can take more than zero time. This paper constitutes the first description of the executable semantics of Rhapsody, highlighting the differences from the Statemate semantics and making an effort to explain the issues clearly but rigorously, including the motivation for some of the design decisions taken.

1 Introduction

In this paper we describe the semantics of statecharts as implemented in the Rhapsody tool. Some early work on incorporating statecharts into an object-oriented framework appears in [11, 1, 12]. However, the detailed basis for a semantically solid OO version of the language of statecharts first appeared in [5]. Two consequences of [5] were (i) the development of the Rhapsody tool to support object-oriented statecharts, and (ii) the essential adoption by the UML developers of its underlying semantics. As a result, Rhapsody [9] can be viewed as the tool that captures the executable kernel of the UML [13].

This having been said, and despite extensive UML documentation, it is also commonly known that there has never been a responsibly detailed description of the executable semantics of the OO statecharts language of [5], as captured by

* This research was supported in part by the John von Neumann Minerva Center for the Verification of Reactive Systems and by the European Commission project OMEGA (IST-2001-33522).

Rhapsody. This we take upon ourselves here, making an effort to explain the issues clearly, including the motivation for some of the design decisions taken. We focus on the differences between the object-oriented nature of Rhapsody compared to the original non-OO statecharts in Statemate. The general spirit and structure of this paper are similar to the paper that described the Statemate semantics [6], and occasionally we even borrow some of the phrases from there. This is done not out of laziness, but to allow readers familiar with statecharts to easily focus on the novel aspects in the new approach. Still, the paper is self-contained, and so is accessible to readers who are not familiar with Statemate semantics or with [6].

The current version of the semantics is a result of much experience gained by users of the Rhapsody tool over the years, which led to modifications and adjustments. Rhapsody executable models can be run in different modes of operation: regular mode, trace mode or animation mode. In the trace and animation modes the user can test the model's behavior by simulating the environment. After each step, the user can generate events and invoke triggered operations that will influence the run of the system. In trace mode, textual information about the system behavior is displayed, while in animation mode a visual graphical representation is displayed showing how the active configuration of each object's statechart changes, by highlighting states entered and transitions taken. The animator can also show inter-object behavior, by creating animated message sequence charts that show graphically how messages are sent between objects during runtime. These can then be compared with previously prepared sequence charts that capture requirements on behavior. There are many interesting issues related to the animation of executable models, but they are beyond the scope of this paper. An important fact that should be stressed is that in contrast to Statemate, the trace and animation modes in Rhapsody use code generated by Rhapsody with additional instrumentation, so that the behavior of the system in these modes is the same as that of the actual production code. This is one of the basic principles that gives added power to executable object modeling.

2 The Basics

An object-oriented system is composed of **classes**. A statechart describes the modal behavior of the class, that is how it reacts to messages it receives by defining the actions taken and the new mode entered. A class can have an associated statechart describing its behavior. These classes are called **reactive classes**. Simple classes that are data driven do not necessarily have statecharts. During runtime there can exist many **objects** of the same class, called **instances**, and each can be in a different **active configuration** — a set of states in which the system resides. Thus, a new statechart is "born" for each new instance of the class, and it runs independently of the others.

The statechart itself is similar to the original description in [4], and to that of Statemate [6, 7], in that there are three types of states, or-**states**, and-**states** and **basic states**. The or-states have substates related to each other

by "exclusive or", AND-states have orthogonal components that are related by "and", while basic states have no substates, and are the lowest in the state hierarchy. Fig. 1 shows the hierarchy and the three types of states that can be used in a statechart. States S, B, C, D are OR-states, state A is an AND-state and states $B1$, $B2$, $C1$, $C2$, $D1$, $D2$, E are basic states. When building a statechart in RHAPSODY an additional state is created implicitly, the **root** state, which is the highest in the hierarchy, in this case the root state has state S as a substate.

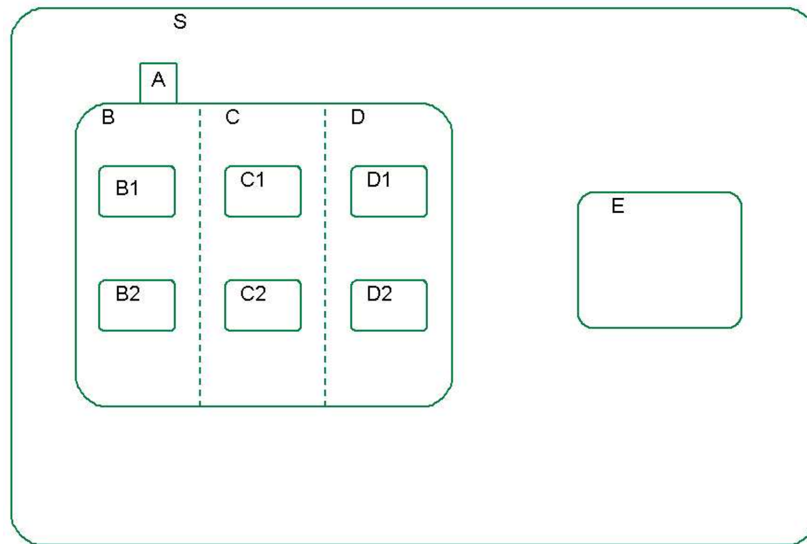

Fig. 1. A small hierarchy of states

The active configuration is a maximal set of states that the system can be in simultaneously, including the root state, exactly one substate for each OR-state contained, all substates for each AND-state contained and no additional states. An example of an active configuration of the statechart in Fig. 1 is :
$\{B1, B, C1, C, D2, D, A, S, root\}$.

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

In RHAPSODY, there is a single trigger, which can be an **event** or a **triggered operation**. Events mean asynchronous communication and triggered operations mean synchronous communication. This issue is discussed in greater length in a separate section. It is also possible to have a transition without a trigger, called a **null transition**. Another kind of message that is used in RHAPSODY is a

primitive operation, which corresponds to an invocation of a method call in the underlying programming language. A primitive operation cannot be used as the trigger of a transition in a statechart, but it can be used in the action part. A trigger can also be a special event timeout, abbreviated $tm(t)$, where t is the time in milliseconds until the event occurs (measured from the time the relevant source state was entered). In RHAPSODY, the guard and action are written in the implementation language[1] and in contrast to STATEMATE there is no special action language. This is a practical design decision, but it should be emphasized that in principle it would be no problem to incorporate such a language. In fact, once the community agrees upon an abstract action language, this could be integrated into the RHAPSODY tool semantics in a natural way.

Besides actions that appear along transitions, they can also appear associated with the entrance to (**Entry action**) or exit from (**Exit action**) a state (any state, on any level). Like actions on transitions, these too are written in the implementation language. 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.

A state can have **static reactions** (SRs), which have the same format as transition labels, i.e., "$m[c]/a$", and again the guard and action are written in the implementation language. Consider the statechart appearing in Fig. 2 (a). State W is associated with a static reaction, as noted by the $>$ symbol attached to its name in the statechart. The actual static reaction $f/act()$ is shown in the state menu at the bottom of the figure. The object is now in state W and in its substate U, and if method f occurs this causes the static reaction to be taken, which involves performing action $act()$. The active configuration of the object does not change, and it remains in U. Semantically, each static reaction in a state can be regarded as a transition in a virtual substate that is orthogonal to its ordinary substates and to the other SRs of the state. Thus, the statechart of Fig. 2 (b) describes the same behavior of that of Fig. 2 (a).

STATEMATE is based on the structured analysis paradigm, where the functional capabilities of the system are captured by activities that are dynamically linked to states in the statechart. Linking states to activities is not relevant to RHAPSODY. As mentioned before, in RHAPSODY the mode of an object of a reactive class is the active configuration of the object's statechart.

The behavior of a system described in RHAPSODY is a set of possible **runs**. 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 Fig. 3). The heart of the semantics, and the main goal of this paper, is to define the effect of a step.

[1] The original version of RHAPSODY used C++ as the implementation language, but current versions support also C, Java and Ada.

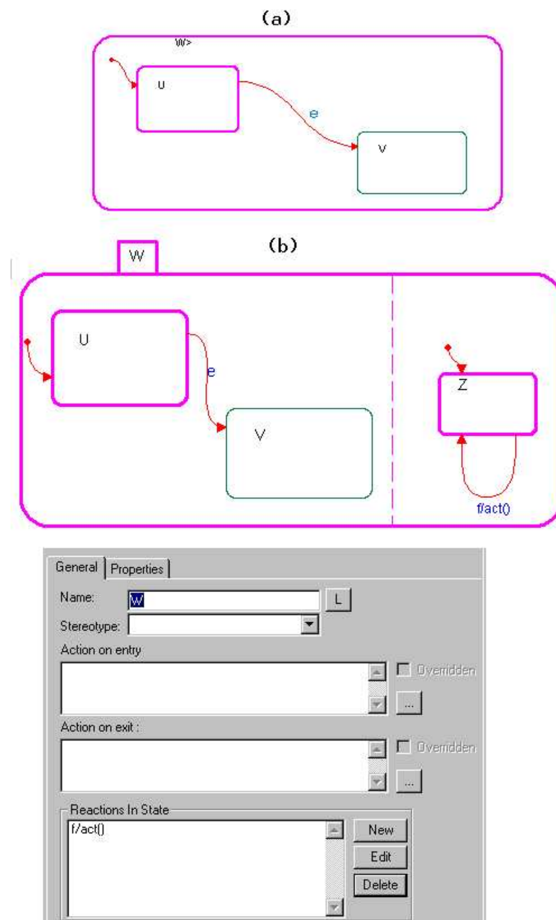

Fig. 2. Static reaction

Each step is composed of **microsteps**, as is shown by "zooming-in" on one of the steps in Fig. 3. The system, being in a certain status and as a response to an occurrence, undergoes a series of microsteps as part of the run-to-completion principle, until it reaches a final status, and at which point it is ready for the next occurrence. Thus the run-to completion principle applies to a step, and it means that as a response to some external occurrence a sequence of microsteps is performed leading to a final status for this step, at which point a new occurrence is considered, initiating a new step. A special case is that of null transitions, that is, transitions without a trigger, and these can be taken spontaneously. A loop of null transitions could in principle cause an infinite number of microsteps to be taken in a single step. However, in Rhapsody this is avoided by the system

setting a maximum value for the number of null transitions that can be taken as part of a step, and informing the user if this bound is violated.

Certain invariants regarding the system's behavior (e.g., being in an OR-state requires being in exactly one of its substates) hold at the beginning and end of a step, but not necessarily in each of the microsteps. Also, a microstep can correspond to performing an action in the implementation language, which can take time. As a consequence, in RHAPSODY a statechart may reside in an OR-state for some non-zero time prior to entering one of its substates.

RHAPSODY supports the development of reactive multi-threaded applications. In such applications each **thread** can perform steps in parallel to the other threads, which makes the definition and behavior more complicated. This topic will be discussed in Section 10, by explaining how threads are introduced into statechart-based systems and how the semantics are defined for them.

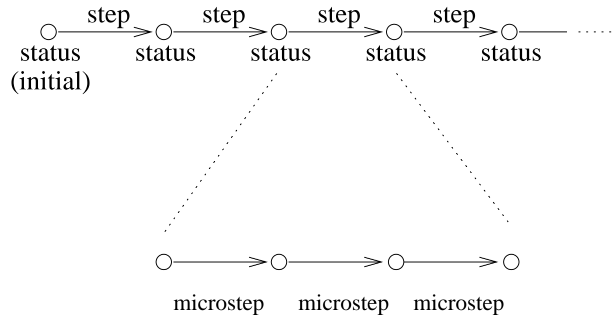

Fig. 3. The step model

A status contains information about all the objects in the system — the states in which the object currently resides, history information for states, values of data members, connections of relations and aggregations and event queues.

Here are some general principles adopted in defining the RHAPSODY semantics:

1. Changes that occur in a step may be sensed in the same step. There is no double buffering to prevent effects from being sensed immediately. This approach is the one more suited to the RHAPSODY context, since a system consists of classes, not all of which have statecharts, and the guards and actions are written in the implementation language. Double buffering would have entailed a high overhead.

2. In RHAPSODY, unlike the situation in STATEMATE, it is possible that many steps will be executed between the time an event is generated and put in the proper event queue and the time it is dispatched to the statechart. Once an event is dispatched to the statechart it will "live" for the duration of one step only, and will not be remembered in subsequent steps.

3. Calculations in one step are based on the current values of data members and the state configuration. When performing a microstep, first the set of relevant transitions is computed and only then are these transitions actually taken. Since there is no buffering as in Statemate, the calculation itself can effect the data members; an evaluation of a guard that has side effects can affect the system. It is not considered good practice to use guards with side effects.

4. A maximal subset of nonconflicting transitions and static reactions is always executed. We refer to this as the "greediness property" of the semantics.

5. The execution of a step does not necessarily take zero time. The time a step will take depends on the actions that are performed while taking the step, mainly those actions corresponding to method calls in the implementation language, and thus are not zero time. Rhapsody supports two models of time, real and simulated. More on these in section 9.

3 Basic System Reaction

A statechart describes the behavior of all instances of a class, but each instance (i.e., each instance's statechart) can be in a different active configuration. After the instance is created a special method, *startBehavior*, is invoked, initializing the behavior of the reactive object and causing its statechart to enter an active configuration according to the default transitions taken from the root. The active configuration can change according to the messages received by the object and the transitions that are performed. The object terminates its life-cycle if it is explicitly deleted or its statechart enters a termination connector.

Statecharts can react to messages by performing a transition from an active configuration to a new active configuration and possibly performing an action.

We now define the reaction of the system during a simple step: how the status of the system changes when performing a single transition between two or-states with the same parent state. Assume that the object in question is in state A in the statechart of Fig. 4(a), and message m (event or triggered operation) is dispatched to the statechart of the object.

The response of the system will be as follows: (*i*) The exit action of state A is performed. (*ii*) The action *act* specified by the transition is performed. (*iii*) The entry action of state B is performed. (*iv*) The active configuration is updated, and the object is 'placed' in state B. The new active configuration of the example is shown in Fig. 4(b).

The action *act* may be of the form $act1$; $act2$; ... act_i. In Rhapsody, actions are guaranteed to be performed in sequential order, each action being executed after the previous has terminated. This in itself does not cause a racing condition. The motivation for this semantics is that actions are written as code in an object-oriented programming language and thus sequential ordering without any double buffering is a natural choice.

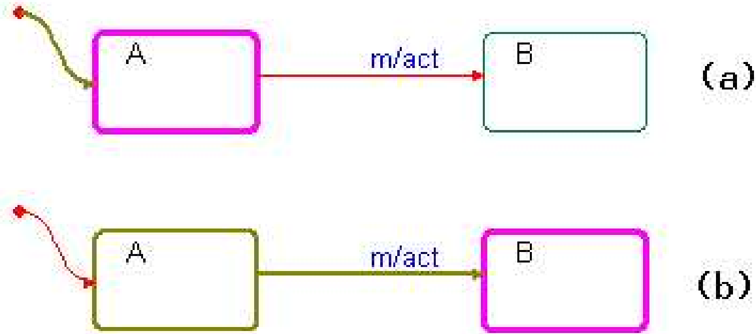

Fig. 4. A simple transition

The behavior described in Fig. 4 could actually be part of a larger step, during which in some microstep the triggered operation m occurs and activates the response described above.

Statecharts can communicate via an asynchronous communication mechanism that uses events and a synchronous communication mechanism that uses triggered operations. In Fig. 4, for example, the message m can be either an event or a triggered operation. We now discuss the two cases and the differences between them.

3.1 Events

Events are used to describe asynchronous communication. They are entities of the model and are defined as part of a *package*. Each class defines the set of events it can receive. The main motivation for using events is that the sender object can continue its work without waiting for the receiver to consume the event. Events can also be used early in the system development process, and later, when a better understanding of the system is gained and decisions regarding synchronization are made, some of these events can be converted to triggered operations.

Events are sent by applying the GEN method to the destination object: $O \rightarrow GEN(event(p_1, p_2, \cdots p_N))$ The sending object should be able to refer to the destination object O (possibly using a navigation expression based on relations in the model). Here $p_1, p_2, \cdots p_N$ are event parameters that match the event's formal arguments (data members). The GEN method creates the event instance and queues it in the event queue of O's thread. In this section we assume a single system thread, and thus all events are handled by the same event queue.

In a multi-threaded application (see Section 10) there is an event queue for each thread.

Events are managed by an event dispatcher in a queue. Once an event gets to the top of the queue, the dispatcher delivers the event to the proper object. When an object receives an event, it will process it according to the run-to-completion semantics. After processing, the event no longer exists and is deleted by the computational framework. Between the time an event is generated and put it in the queue and the time it is dispatched to the destination object, the destination object could be destroyed, in which case all events that were sent to it will be deleted and will have no effect.

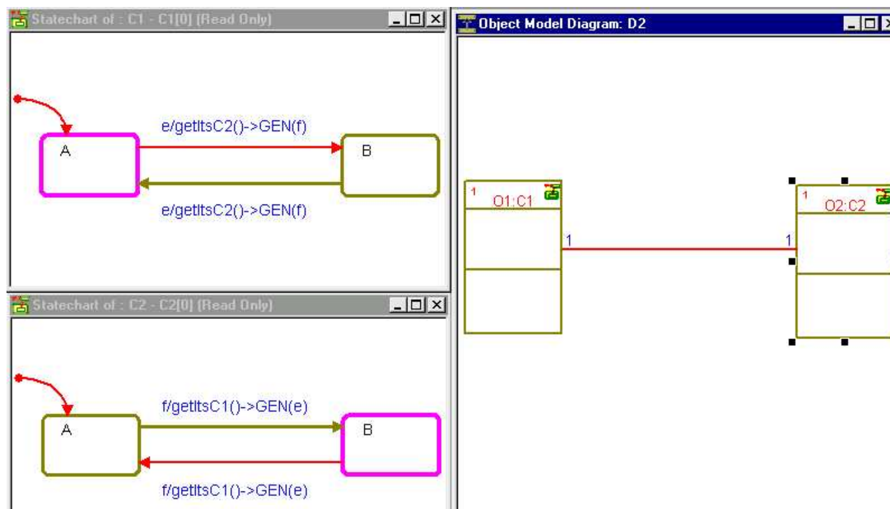

Fig. 5. Communication using events

Consider the system in Fig. 5, with objects O_1 and O_2 of classes C_1 and C_2, respectively, and with a one-to-one relationship between the objects. If object O_1 receives event e (say, from the user), the transition from state A to state B is taken, involving sending event f to object O_2 (by placing a new event f in the event queue), as specified by the action $getItsC2() \rightarrow GEN(f)$, since O_2 is the object that it recognizes from class C_2. Once the transition to state B of O_1 is completed, event f is removed from the event queue, and is dispatched to object O_2, causing it to take the transition from its state A to state B. In a similar way, object O_2 now sends event e to O_1, and the process repeats itself. In this way, a feedback loop is created, with objects O_1 and O_2 repeatedly moving between states A and B, and sending events e and f to each other, ad infinitum.

Unlike STATEMATE, there is no special treatment of internal events in RHAPSODY. Sending events internally is done simply by omitting the destination object from the send operation, as follows:

$GEN(event(p_1, p_2, \cdots p_N))$

Consider Fig. 6. On creation of an object of this class the statechart is initiated, and the default transition to state A is taken, performing the action $GEN(e(1))$, which causes the internal event e with parameter value 1 to be sent. Only after the object has completed the transition and is in state A, is the event e dispatched to the statechart. Processing this event causes a transition to state C, since of the two outgoing transitions from the condition connector, the guard of the transition to state C is satisfied (the parameter of the event in the transition is referenced using the $params \rightarrow$ command, and here the value was 1).

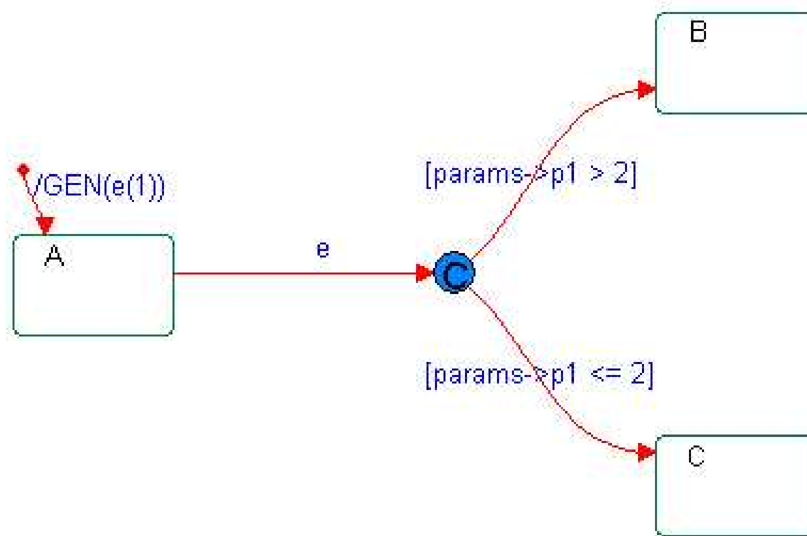

Fig. 6. More communication using events

Events are independent entities of the model and can be sub-classed like inherited objects, a mechanism that can be used in order to add attributes — event parameters. In particular, if event $e2$ is derived from event e in this way, $e2$ will trigger any transition that has e as a trigger. For example, in Fig. 7, after taking the default transition into state A and sending event e_2 to itself, the transition to state B is taken, since e_2 inherits from event e.

3.2 Triggered operations

Triggered operations are services provided by a class, and are defined as part of the serving class. They are a synchronous communication means between a

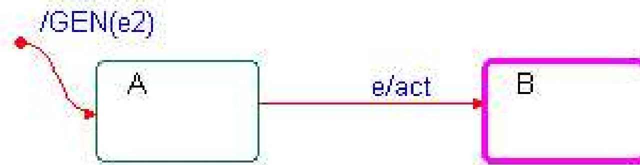

Fig. 7. Event $e2$ inherits from e

client and the server object. A triggered operation may return a value to the client object, since its activation is synchronous.

Unlike events, triggered operations are not independent entities; rather, they are part of the class definition, and are not organized in hierarchies. The use of a triggered operation corresponds to the invocation of a class member function. The main reason that triggered operations were integrated into the RHAPSODY framework was to allow the usage of statecharts in architectures that are not event-driven, and thus to specify the behavior of objects in the programming sense of operations and object state. Triggered operations also provide means for late design decisions to optimize execution time and sequencing, by converting event communication into direct triggered operation invocation.

A triggered operation is invoked like a primitive operation in the underlying implementation language:

$result = O \rightarrow t(p_1, p_2, \cdots p_N)$

A triggered operation may return a value whose type is the one defined in the object model, where the operation interface is defined. The return value for a triggered operation must be set within the transition. Replying to a triggered operation is done by calling the reply method defined for the class. The following transition label specifies a reply to the operation t:

$t/reply(17)$

Consider the two statecharts of classes X and Y, described in Fig. 8 (a). If an object of class X receives the event *go*, a transition from state A to state B is taken, which invokes the triggered operation t in the relevant object of class Y, as specified by the action $getItsY() \rightarrow t()$. The transition to state B of X is not completed before t is processed, causing the Y object to move from state A to state B, with 10 being the returned value of t. The figure shows the active configuration of the statecharts in animation mode, at a point when the Y object has completed its transition to state B, and the X object is in the midst of the transition. The X object's transition is completed after *setResult* is called with

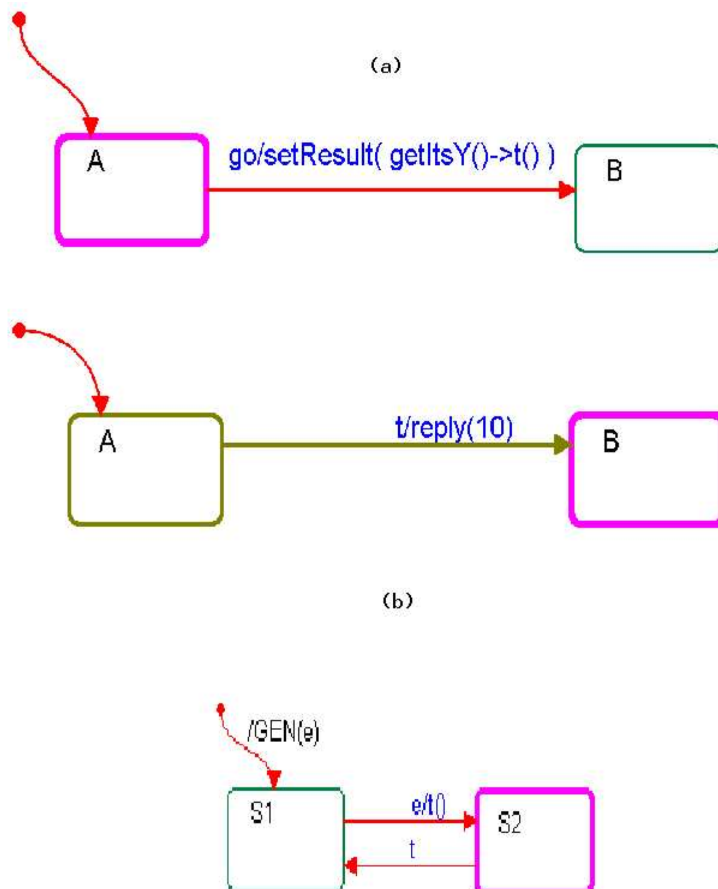

Fig. 8. Using triggered operations

the value 10 that was returned by the triggered operation, and the value of the data member *result* of X is updated. Only then is state B entered.

One thing that has to be resolved here is the reaction of an object to an invocation of a triggered operation when it is not in a **stable** state, i.e., when it is in the midst of performing a transition. This is especially relevant in RHAPSODY, since transitions do not take zero time. This would not be a problem if we considered only events, since events represent asynchronous communication and are queued; the next event is taken from the queue only after the step completes, so that the run to completion semantics assures the object is in a stable state. For triggered operations there is no such assurance.

This situation is demonstrated in Fig. 8 (b). While taking the default transition to state $S1$, event e is generated, causing the transition from state $S1$ to

state $S2$. However, during the process of carrying this out, the action $t()$ is performed which invokes the triggered operation t on this statechart. There are a number of alternatives for dealing with this kind of situation: One is to treat this as a deadlock, and a problem in the design. Another is to allow the transition to be completed and state $S2$ to be entered, and only then to process t, causing a transition to be taken back to state $S1$.

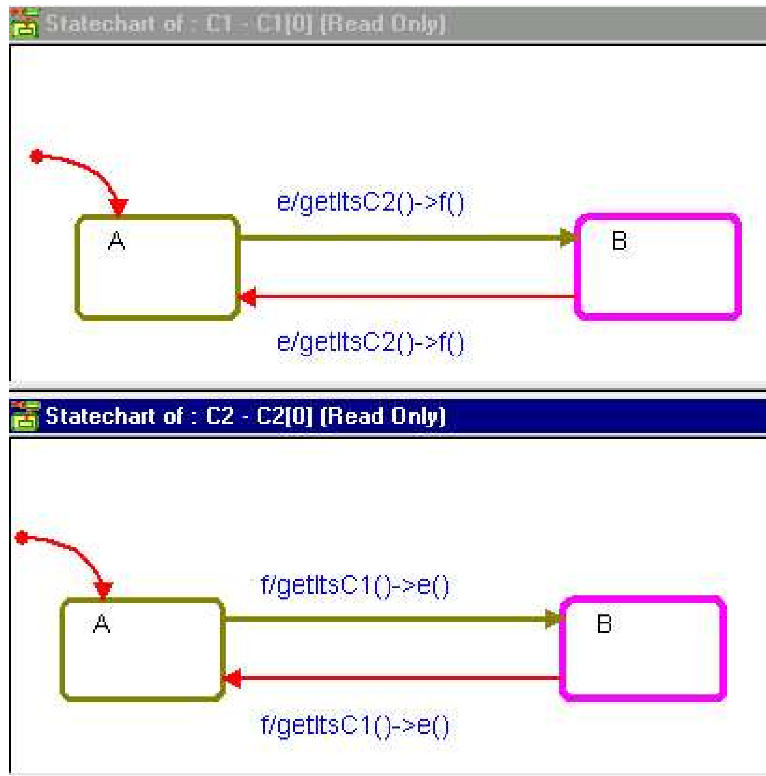

Fig. 9. Coordinated transitions

In RHAPSODY, a different choice was made: the invocation of a triggered operation t in the midst of a transition causes no effect, and the return value from such a call is undefined. In the above example, the object completes its transition to state $S2$ and remains there. The semantics is implemented by a locking mechanism that causes an object to ignore the invocation of triggered operations while in the middle of a step. A self call such as that in the example is a special case, but in general this can also occur as the result of a chain of calls between different objects, ending in a triggered operation invocation to one of the objects that is still in the process of performing a transition.

Earlier, in Fig. 5, we showed an example of a feedback loop between two statecharts. If we modify this example so that e and f are triggered operations instead of events, as shown in Fig. 9, then the result of invoking e on O_1 is that both objects enter state B. In this case, the feedback loop does not close, since when O_2 takes the transition to state B and invokes e on O_1, O_1 is still in the middle of the transition between A and B; hence e is ignored. Once both objects are in state B, if e is externally invoked on O_1 (or, alternatively, f is invoked on O_2), both objects take transitions to state A and remain there. Notice that had we changed only one of the two events to a triggered operation and left the other as an event, the feedback loop would have remained.

4 Compound Transitions

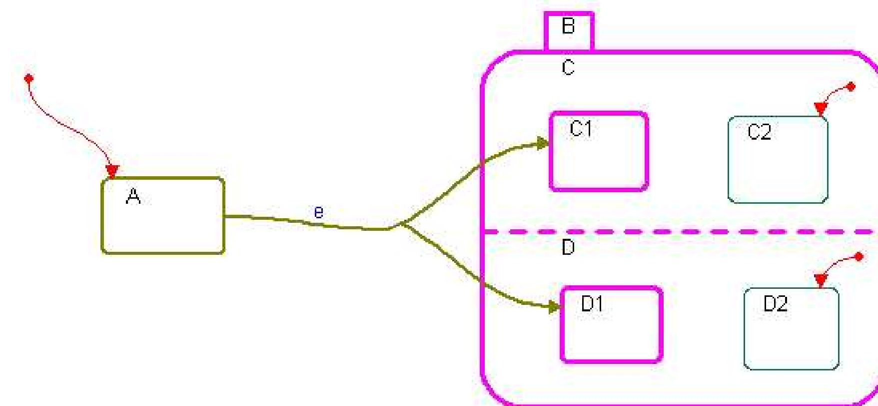

Fig. 10. A fork connector

Statecharts allow defining transitions in a richer way then just by the simple directed arrow that connects two states, of the kind shown in Fig. 4. This general construct is called a **compound transition** (CT) and may consist of a number of separate transitions appearing in different orthogonal state components. Each of these, in turn, may consist of a number of linked **transition segments**, which are the labeled arrows that connect states and connectors of various kinds. This section explains how transition segments are combined to form a compound transition (CT). We explain the semantics of the different types of connectors and restrictions on how they are used.

The connectors come in two different types: AND and OR. The **fork** and **join** are AND connectors. The transition segments connected to an AND connector will all participate in the same CT. Consider the statechart appearing in Fig. 10. It the object is in state A and the trigger e occurs, the CT transition is taken, causing entrance to state $C1$ in orthogonal component C and entrance to $D1$ in orthogonal component D. The fact that the fork is an AND connector implies that both the transition segment leading to state $C1$ and the one leading to $D1$ must be taken as part of the CT. The destination of a fork segment must be a state or a history connector and the segment cannot have a label.

An example of a join connector is shown in Fig. 11. If the object is in states $B2$ and $C2$ and in either $D1$ or $D2$ and the trigger e occurs, the CT is taken, which causes a transition to state E. The fact that the join is an AND connector implies that both the transition segment leading from state $B2$ and the one leading from state $C2$ must be taken as part of the CT. The transition segments entering the join connector cannot have labels.

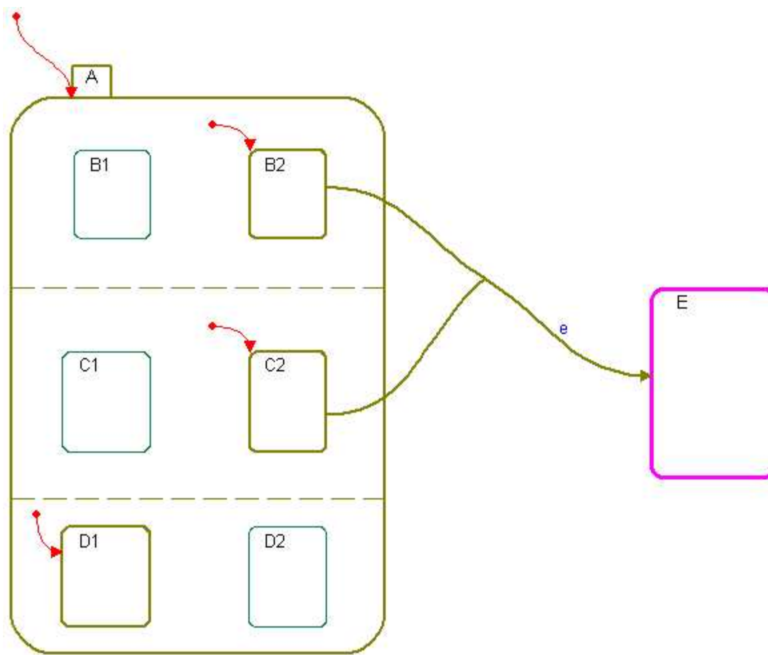

Fig. 11. A join connector

The **junction** and **condition** are OR connectors. Of the transition segments connected to an OR connector exactly one incoming transition segment and exactly one outgoing transition segment must participate in the CT.

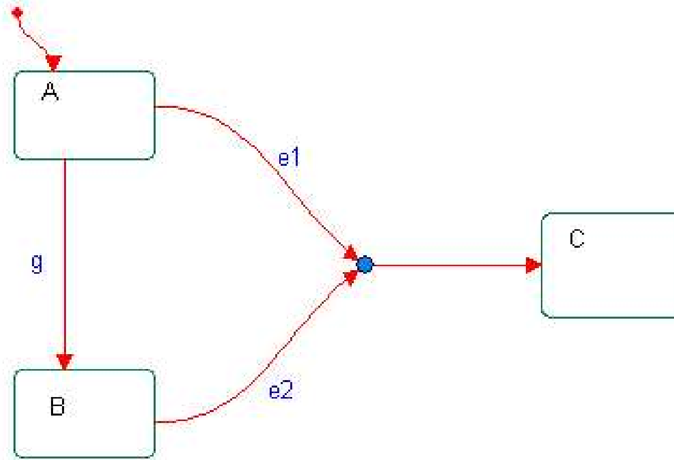

Fig. 12. A junction connector

An example of a junction connector is shown in Fig. 12. If the object is in state A and the trigger $e1$ occurs, or it is in state B and the trigger $e2$ occurs, a transition to state C is taken. In terms of the active configuration of the statechart, an equivalent statechart has two separate transitions, one from state A and one from state B, as shown in Fig. 13. The label is either written on each of the transition segments entering the junction connector, as in Fig. 12, or on the common transition segment exiting the junction connector, as shown in Fig. 14.

A **condition** connector has one incoming transition and can have several outgoing transition segments called **branches**. Branches are labeled with guards that determine which one is to be actually taken. Since the condition connector is an OR connector, only one of the branches can be taken. If the guard of more than one of the branches holds then one is chosen arbitrarily. Each condition connector can have one special branch with a guard labeled $else$, which is taken if all the guards on the other branches are false. Branches cannot contain triggers, but in addition to a guard they may contain actions. A branch can enter another condition connector, thus providing for the nesting of branches. An example is shown in Fig. 15.

When taking a transition, first the guards are all evaluated, and only then are the actions performed. In the statechart described in Fig. 16, for example, the state that is reached is B. The reason is that first the transition to be taken is selected by evaluating the guard, and in this stage $x = 1$; only when

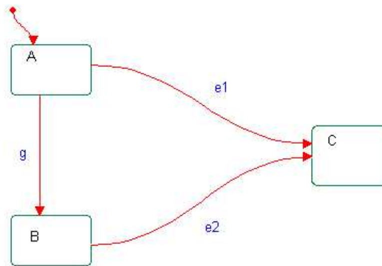

Fig. 13. A construct equivalent to a junction connector

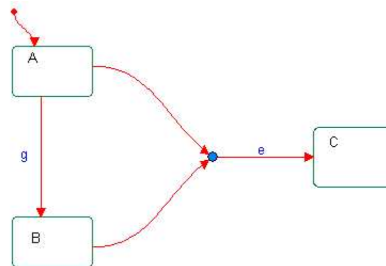

Fig. 14. A junction connector with a common label

performing the transition is the action $x = 2$ performed, but it cannot influence the transition taken.

A step always leads from one legal state configuration to another. A statechart can not remain "stuck" at a connector (with the exception of a termination connector). Similarly, a statechart cannot be in a non-basic state without the ability to enter appropriate substates. For this reason, every OR state with more than one substate must have a *default connector* with a transition to one of the OR state's substates. If a destination state of a CT causes a statechart to enter a non-basic state, the default transition associated with this state will be taken. For example, if the object in Fig. 17 is in state A and e occurs, the transition to state B is taken, followed by the default transition to state C.

Taking a default transition is considered to be a microstep. Attributes get their values just prior to the microstep and not the values present at the beginning of the entire step. Thus, in Fig. 18, if the object is in state A and e occurs, the transition to state B is taken, and this is followed by the default transition that leads to state C, since the action $x = 1$ is performed before the default transition's microstep is taken.

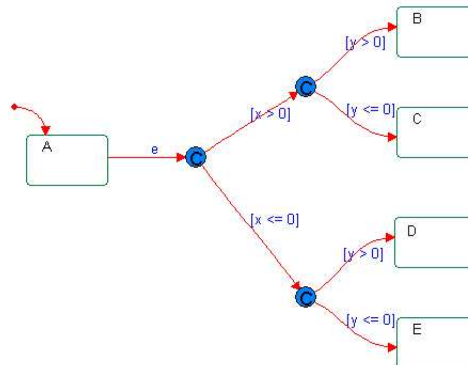

Fig. 15. Nested condition connectors with a common label

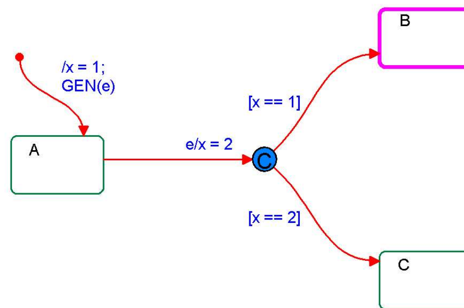

Fig. 16. A condition connector

As part of a CT, it is possible that several default transitions are taken, each one leading to a deeper state in the hierarchy until finally a basic state is reached. Each such default transition is a microstep and actions performed in the previous microsteps are taken into account.

5 Dealing with History

A **history** connector is used to store the most recent active configuration of a state. Each state can have at most one history connector. The semantics of the history connector is that when the connector is the source of a CT, the statechart transitively enters the most recently visited active states.

An example of a history connector is shown in Fig. 19. If the object is in state A, but has never yet entered state B, and the trigger e occurs, a transition to the history connector is taken, followed by the outgoing transition from the history

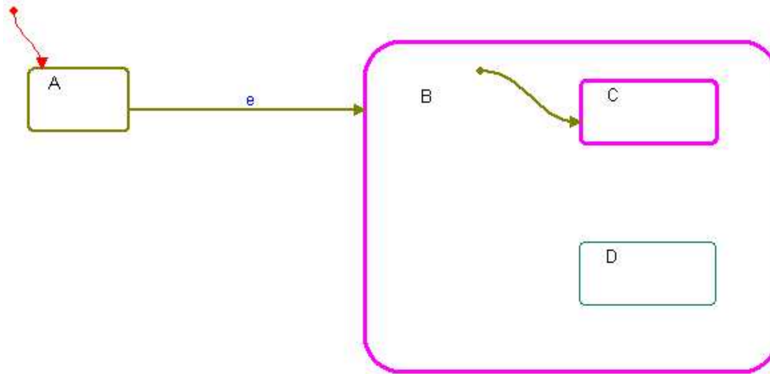

Fig. 17. A default connector

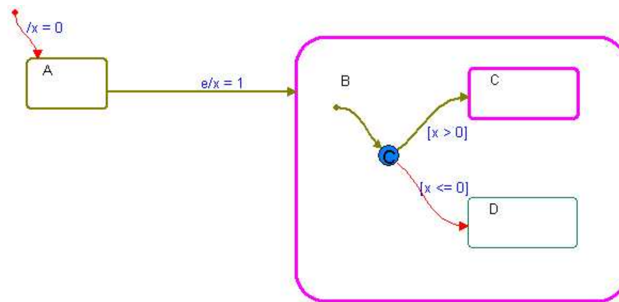

Fig. 18. A default transition as a microstep

connector to state D. Next, if the trigger f occurs the transition to state F is taken. Later, if the trigger f occurs again the active configuration is stored by the history connector and the transition to state A is taken. Finally, if the trigger e occurs, state D and then its substate F are entered, since they constituted the last active configuration prior to state B being exited. States D and F are entered without taking the outgoing transition from the history connector and without performing default transitions or any actions associated with them.

Unlike STATEMATE, the semantics of the history connector in RHAPSODY is the "deep history" semantics (in [4] this is associated with the special notation H*), which entails entering the substates of the most recent active configuration recursively, until basic states are entered. The shallow semantics of STATEMATE is not supported in RHAPSODY .

Also unlike STATEMATE, currently RHAPSODY does not support the $history-clear(S)$ operation, which erases the history of state S, thus causing the next transition to the history connector in S to proceed via the default transition as if it were the first time S is entered.

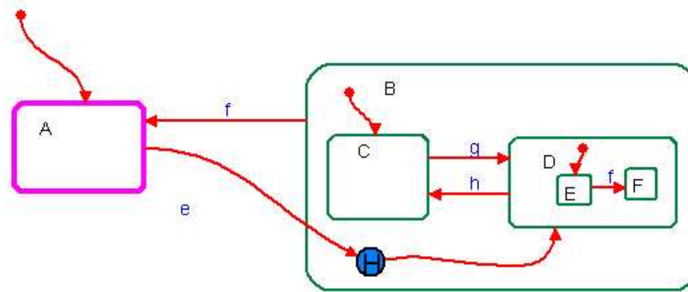

Fig. 19. A history connector

6 The Scope of a Transition

In taking a transition from a source to a target, a CT will often pass through different levels of the statechart hierarchy. As part of performing the CT this causes exiting some of the states and entering others, and performing the appropriate exit and entry actions.

The goal of this subsection is to define the **scope** of a transition, thus determining which states should be exited and which entered while taking a CT. The definition of the scope is the same as in STATEMATE, and we repeat it here for self containment. There are some differences in the usage of the definition, which are discussed later.

Before presenting the definition, consider the simple case of the statechart in Fig. 20. Taking the transition with trigger e causes exiting state B and entering state C. Any relevant entry and exit actions are performed. The scope of this transition is state A.

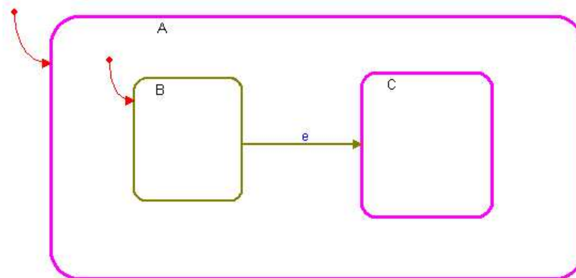

Fig. 20. The scope of a transition

The scope of a CT is the lowest OR state in the hierarchy of states that is a proper common ancestor of all the source and target states. Taking the CT will result in a change of the active configuration involving only substates in the scope. When the CT is taken, all the proper descendants of its scope in which the system resided at the beginning of the step are exited, 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.

We now illustrate the notion of scope by some examples. Consider the statechart of Fig. 21 (a). If the associated object is in state W and message e occurs, the transition with scope U is to be taken, since according to the previous definition U is the lowest OR state in the hierarchy that is a proper common ancestor of V. Thus, taking the transition implies exiting states W and V and entering states V and W. We defined U to be the scope of the transition since we consider state V to be both the source and the target of the transition. Notice that although the (implicit) default transition to state W is taken, we still consider V to be the transition's target since the default transition is taken as part of a new microstep. This is important, since if the statechart was modified so that the source of the transition becomes W, as shown in Fig. 21 (b), considering W to be also the target of the transition would have implied that V is the scope of the transition, while in fact according to our definitions U is the scope.

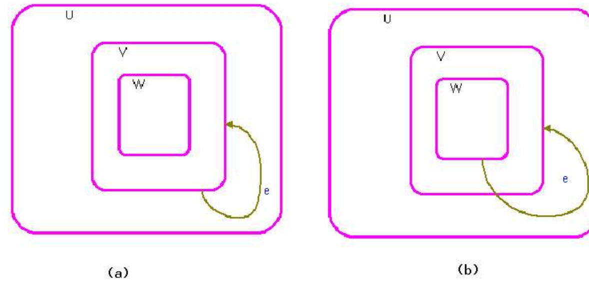

Fig. 21. The scope of a transition

Consider the statechart of Fig. 22. If the associated object is in states $B2$ and $C1$ and receives message f, it takes a compound transition, causing it to enter states $C2$ and $B1$ (the latter by the default transition). According to the previous definition, S is the scope of the transition, being the lowest OR state in the hierarchy of states that is a proper common ancestor of states $B1, B2, C1$ and $C2$. The states exited are $B2, B, C1, C$ and A, and those entered are $A, B, B1, C$ and $C2$. Notice that the notion of scope does not depend on the way the transition itself is drawn, but on its sources and targets only: the transition in Fig. 22 is drawn inside state A but this does not cause A to be the scope of the transition

rather than S. Even if the transition would have be drawn as exiting the contour of S, the scope would still be S.

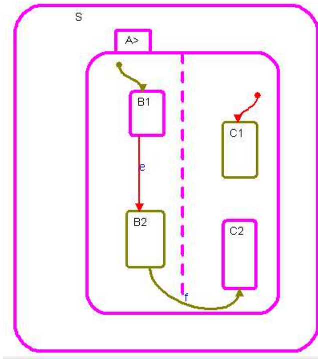

Fig. 22. More on the scope of a transition

7 Conflicting Transitions (Nondeterminism)

We say that two transitions are in **conflict** if there is some common state that would be exited if either of them were to be taken. Consider the statechart in Fig. 23 (a), the two outgoing transitions from state A labeled e are in conflict because they would each imply exiting state A. The transition from state U to state D is in conflict with the two outgoing transitions from state A and also with the transition from state B to state C, since if the transition from U to D is taken it implies also exiting whatever substate of U the object was in.

Two conflicting transitions cannot be taken in the same step. If they are both enabled only one will be taken. We now explain how this choice is made.

The two types of conflicts in Fig. 23 (a) are treated differently. If the object was in state A and message e occurred the system is faced with nondeterminism, since there is no reason to prefer a transition to one of the states B and C over the other. RHAPSODY detects such cases of nondeterminism during code generation and does not allow them. The motivation for this is that the generated code is intended to serve as a final implementation and for most embedded software systems such nondeterminism is not acceptable.[2]

[2] We suggest that an option be provided to the user to allow such nondeterminism, which can be useful in certain development stages where the model is not yet complete. In any case, the current implementation cannot block all nondeterminism when performing code generation, since we may have conflicting transitions with the same trigger but with different guards, and in general it is impossible to detect at compile time whether both guards will evaluate to true.

The second case of conflict in Fig. 23 (a) is that between the transition from A to B (assume that the transition between A and C has been removed) and the transition from U to D. In RHAPSODY, when a message can trigger several conflicting transitions priority is given to lower level source states. Hence, here the transition from A to B takes priority over the one from U to D, and there is no nondeterminism. At the end of the step the object will be in state B. Join transitions get priority according to their lower source state. If there is no hierarchal relation between the source states no priority is defined between the transitions. This priority strategy is different than that of STATEMATE, which determines priorities outside-in; in our case according to STATEMATE the object will end up in state D. The strategy in RHAPSODY is more object-oriented, since it enables substates to override transitions in higher states in a way similar to that in which operations in subclasses can override those of the superclass.

Another technical difference between STATEMATE and RHAPSODY is that in STATEMATE we determine priorities outside-in according to the scope of the transition, while in RHAPSODY we determine priorities inside-out according to the source state. Consider the statechart in Fig. 23 (b). If the object is in state E and message e occurs, then in RHAPSODY we take the transition to state F, since the source of this transition E is lower than the source of the transition to state C which is B. In STATEMATE the scope of both transitions is A, resulting in nondeterminism.

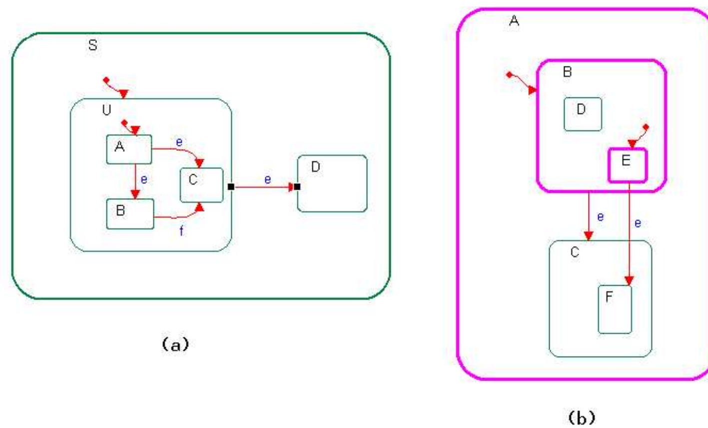

Fig. 23. Conflicting transitions

The priority of a static reaction is determined according to the state in which it is defined, giving high priority to lower-level states. If a CT and a SR are in conflict, the one with lower source state will be taken and the other will not. If the CT and SR have the same source state, as in Fig. 24, the CT has higher priority, thus the transition to state B will be taken and the static reaction will

not be carried out. This is different from the STATEMATE approach, were 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.

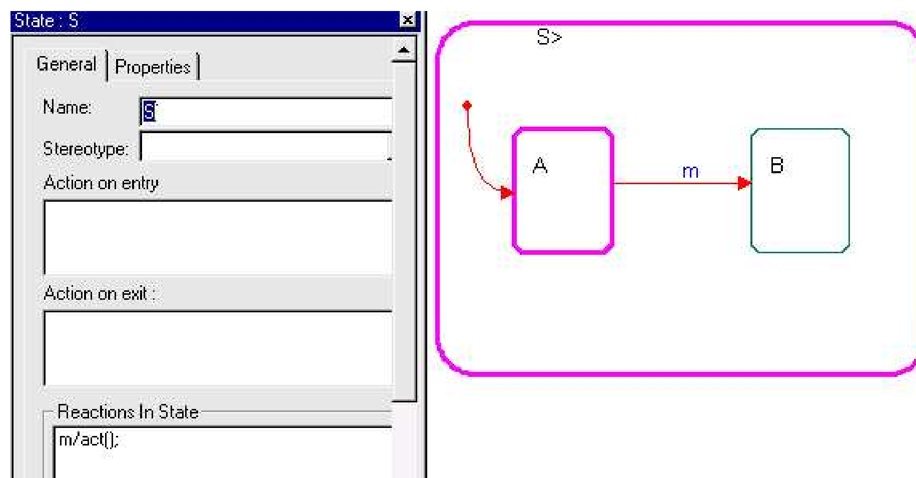

Fig. 24. Conflict between transition and static reaction

8 The Basic Step Algorithm

In this section we present a schematic description of the algorithm that executes a step. For a single threaded application, we do the following repeatedly:

If the event queue is not empty, get the next event and its destination from the queue. If the destination object still exists dispatch the event to that object's statechart. The event invocation may cause taking SRs or CTs and all the relevant default transitions, as explained in earlier sections. At the end of the run-to-completion the statechart of the object is in a (possibly new) active configuration. If the statechart does not specify a transition in response to the event, the active configuration remains unchanged. The loop can now be continued, processing the next event.

A pseudocode description of the procedure is:

```
procedure StepCycle ()
begin
  loop forever
    while Event-Queue ≠ empty do
      ev ← Get-Event-From-Queue
      dest ← Get-Destination-Of-Event
      if dest still exists then
        dest → takeEvent(ev)
      else
        Ignore ev
      end if
    end while
  end loop
end
```

Here now are the details of the main part of this procedure (*takeEvent*), in which an event is processed by the statechart.

- *Determine the CTs/SRs that will fire in response to the message:* Traverse the states in the active configuration from lowest states in the hierarchy upwards. A CT/SR is enabled if its trigger is the dispatched event *ev* or a super-event of *ev*, and the guard evaluates to true. Since for a given state CTs have priority over SRs, they are considered first. Once an enabled transition is found with a given source state stop traversing the states that are higher than this state in the hierarchy. States in orthogonal components are still considered since they may be taken without necessarily causing a conflict.
- *Perform the CTs/SRs that we found should fire:*
  For each transition do:
  - Update histories of exited states.
  - Perform the exit actions of the exited states according to the order states are exited, from low state to high state.
  - Perform the actions on the CT/SR sequentially according to the order in which they are written on the transition, from the action closest to source state to the action closest to target state.
  - Perform the entry actions of the entered states according to the order states are entered, from high state to low state.
  - For lowest level states that were entered, which are not basic states, perform default transitions (recursively) until the statechart reaches basic states.
  - Update the active configuration.

  The order of firing transitions of orthogonal components is not defined, and depends on an arbitrary traversal in the implementation. Also, the actions on the transitions of the orthogonal components are interleaved in an arbitrary way.
- *Deal with null transitions:* After reacting to a message, the statechart may reach a state configuration where some of the states have outgoing enabled null transitions — transient configurations. In such a case further steps need

to be taken until the statechart reaches a stable state configuration where no null transitions are enabled. Null transitions are triggered by null events that are dispatched to the statechart whenever a transient configuration is encountered. Null events are dispatched in a series until a stable configuration is reached. It is possible that the statechart will never reach a stable configuration; for example when there is a loop of null transitions. In Rhapsody the infinite loop is detected during runtime and execution is halted. It is possible using the execution framework to set a maximum value for null transitions. When performing the null transitions, each one is taken separately and the values used in the computation are the values after the previous null transition and not the values before the entire step.
– *Wrap up:* Once a stable configuration is reached, the reaction to the message is completed, control returns to the dispatcher and new messages can be dispatched.

9 The Time Model

The Rhapsody time model is more complex than that of Statemate, since Rhapsody allows describing both synchronous and asynchronous behavior in the same model. Moreover, a step does not necessarily take zero time. Due to these facts, the synchronous time model of Statemate is not relevant here. Rhapsody supports two different modes of handling the progress of time: real time and simulated time. In real time mode time advances according to the actual underlying operating system clock. In simulated time the user of Rhapsody can control the progress of time in an interactive way, thus enabling effective debugging and testing of the model. A detailed description of the Rhapsody time model will appear in the full version of the paper.

Recall that all aspects of the execution of a Rhapsody model, and this includes timing aspects too, are carried out via the generated code. This is important in Rhapsody, since one of the main goals is to develop production-code. However, there are many interesting opportunities for further research on the timing aspects of modeling object-oriented systems, especially regarding the simulated time mode. In fact, we predict that analytic techniques could be modified to apply to timed behavior, in ways that do not depend directly on the generated code and are thus more robust.

10 Multi-Threaded Systems

Rhapsody supports the development of reactive multi-threaded applications. In such applications each **thread** can perform steps in parallel to the other threads. Obviously, this makes the definition and behavior more complicated. We now discuss this topic in some detail, by explaining how threads are introduced into statechart-based systems and how their semantics is defined. A detailed description of this topic will appear in the full version of the paper.

An object-oriented system consists of objects exchanging messages. The ideal analysis view of such a world is that each object is an autonomous entity executing concurrently with all other objects. In order to have a more realistic and concrete model, this general abstraction can be given various interpretations, regarding the synchronization between objects and the semantics of messages.

In the *synchronous model* objects execute on a clock edge, and the period between two clock edges is called a step. This model is similar to digital hardware systems, where all components are synchronized by a clock. It is also the model implemented in STATEMATE, and although STATEMATE is executed on a sequential machine and concurrency is achieved by simulation, messages sent at a certain step being processed in the next step. The major advantage of this model is that it is deterministic and simple. However, it does not fit software systems for the following reasons: Software systems have a very limited form of concurrency, since in general they run sequentially on the same CPU. Also, in the case of concurrent software, tight synchronization is an undesired overhead, so that concurrent software components are by default asynchronous unless they are explicitly synchronized.

Since real concurrency does not exist in most software applications, the CPU is shared by all software objects. The sequence in which software functions execute is known as the thread of control, which can be thought of as a token (representing the CPU) passed between objects in the system, enabling them to execute. Initially, the token is given to the main program, and it is typically passed along by method activation. A client object sending a message to a server object actually gives up its control of the CPU in favor of the server object. This passing along can be nested, with $o1$ calling $o2$, who calls $o3$, and so forth.

In the general case, a system will have more than one thread, which means that conceptually it has multiple tokens, and this is a far more complicated setup than a single-threaded one. We now discuss the way RHAPSODY deals with some of the major issues in multi-threaded systems: thread creation and destruction, associating objects with threads, and communication and synchronization between threads.

Object/thread relationship: An important issue in a multi-threaded system has to do with which objects belong to which thread. In RHAPSODY, a class can be defined as an **active class** and then each of the instances of this class will have its own thread of control. Another way of defining the object/thread relationship is through composition. Instances that are components of a composite class run on the thread of the composite class, unless they are instances of an active class, in which case they have their own thread.

Instances of classes that are not designated as active classes run on the unique system thread, which is the default thread used by the main program.

It is also possible to set the thread of an object explicitly, by calling the $setThread()$ command. This gives developers more control over threading policies. However, it also introduces many delicate issues, such as thread destruction policy, and how to transfer events to the new event queue after an object changes

its thread. Some of these issues of dynamic object/thread relationship require further research to enable automatic support for more complicated groupings.

Creation and destruction of threads: The special system thread is created when the main program for the executable model is started. This thread will be destroyed only when the application terminates. Creating an object that is an instance of an active class causes the creation of a thread on which this object runs. This thread is destroyed when the object is destroyed, which can happen explicitly from the outside, or by the object's statechart entering a termination connector. In the case where components of a composite class run on the thread of that composite class, destroying a component does not cause the destruction of the thread; the thread will be destroyed when the composite class object is destroyed.

Automatic support for thread destruction in the case of explicit setting of an object's thread is not currently supported by RHAPSODY. A possible solution is to destroy the thread only when the last object running on the thread is deleted.

Communication and synchronization between threads: As discussed previously, the statechart of an object can deal with asynchronous communication using events and synchronous communication using triggered operations. In the multi-threaded case, an object can receive messages from different objects, each having its own thread of control and therefore running concurrently with other objects.

The case of asynchronous communication using events is simpler: The generated events are put in the event queue of the receiving object and are later dispatched to the statechart. In the case of synchronous communication using triggered operations, the sending object is blocked until the receiving statechart completes its response to the triggered operation. Hence, if different threaded objects invoke a triggered operation on the same statechart they will be posted to the statechart one at a time and each sending object is blocked until its invocation is completed. Situations of deadlocks and starvation are possible, and must be avoided as part of the model design.

Classes can also communicate by calling member functions; i.e., primitive operations. Since synchronization in multi-threaded applications is important, RHAPSODY allows the definition of *guarded* primitive operations. All the guarded primitive operations of a class are mutually exclusive, in that only a single operation can run at any given time and the other invocations are blocked. Operations that are not defined as guarded can run in parallel. Triggered operations can also be defined as guarded, thus causing all guarded operations (primitive or triggered) of the class to be mutual exclusive.

The step algorithm for a multi-threaded system consists of performing the step cycle described in the basic step algorithm for each thread. There are several complications in the semantics relative to the single-threaded case. For example, when one thread is in the middle of performing a step (and as explained earlier, this might take more than zero time), a second thread can interact with it by invoking primitive or triggered operations or sending events. For events,

the RHAPSODY execution framework guarantees that the event queue is not corrupted by different threads interacting with it simultaneously, and that events are not lost. This is achieved in RHAPSODY by locking mechanisms. Before accessing the event queue a *lock*() command is invoked, which prevents other threads from interfering with the queue. Only after the interaction with the event queue is over does the *unlock*() command allow other threads to lock the queue and use it. The code generation framework in RHAPSODY implements the *lock*() and *unlock*() commands using a mutual exclusion mechanism in the underlying operating system. In the multi-threaded case, these locking mechanisms can prevent an object attempting to send an event from proceeding until it manages to perform the lock. In contrast, the single-threaded case allows the sending object to generate an event and continue progress immediately.

11 Racing Conditions

A **racing condition** occurs when the execution of transitions in two different legal orders would cause the system to end up in two different configurations. In STATEMATE, the semantics and execution model were simpler, and this allowed the tool to detect and report such conditions. The fact that RHAPSODY deals with synchronous and asynchronous communication and well as with multi-threaded applications, and the fact that a step does not take zero time, make automatic detection and reporting of racing conditions a much harder task, and RHAPSODY does not attempt to undertake it. Developing tool support to handle these issues requires further research. Until this situation changes, users of tools that deal with such advanced features are advised to make efforts to avoid racing conditions by improving and tightening their models.

12 Comparison with other work

Readers interested in comparing the RHAPSODY semantics of statecharts with non-OO approaches to statechart semantics are referred to the discussion in Appendix A of [6]. We now briefly discuss appropriate object-oriented approaches.

ROOM: The ROOM method of [12], and its supporting tool OBJECTTIME (which later evolved into Rose-RT) were the first to introduce extended state-machines into an object-oriented paradigm in a way that allows development of fully executable models. The main formalism for describing behavior in ROOM is called ROOMcharts, which was inspired by the original statechart formalism [4]. ROOMcharts allow hierarchal nesting of OR-states but not orthogonality (AND-states), which thus renders the language much simpler. The semantics of the language is based on the run-to-completion principle, and an assumption is made that the time taken to process any single event should not exceed the maximum latency requirements of the object. The communication between objects implemented as ROOMcharts can be carried out using asynchronous events and

triggered operations of Rhapsody, and there is a mechanism for defining event priorities.

UML 2.0: In the very recent UML 2.0 there is a distinction between two kinds of state machines (both are variants of statecharts): **behavioral state machines** and **protocol state machines**. Behavioral state machines are really the original OO statecharts of [5] and the present paper, and they are used to describe the behavior of an object. In contrast, protocol state machines describe usage protocols, and are thus geared to specifying requirements of classes, interfaces and ports, rather then defining the entire behavior of an object. In the behavioral statecharts of UML 2.0 shallow history is allowed too, in addition to deep history. Triggers can be signals (corresponding to events in our paper) or operations (corresponding to our triggered operations). The semantics is that of run-to-completion [5], and the way conflicting transitions are handled is by a selection algorithm similar to the one introduced in Rhapsody and reported upon here. UML allows deferred events, which are not lost if dispatched to the object and the event is not enabled. This extension is currently not part of Rhapsody statecharts. The UML allows the definition of submachines, which is a syntactic way to break up a statechart and describe some of the more complex hierarchal states in different diagrams. This is also supported by Rhapsody, but is not essential to this paper because it is essentially a syntactic extension with virtually no impact on the semantics.

It should be noted that the UML standard leaves certain semantical options open, thus allowing "semantic variation points", that can be implemented differently by the tool vendors or according to the application domain.

Semantics for formal verification: Following the publication of [5] and the release of Rhapsody, and aided by the growing popularity of the UML [13], its application to safety-critical systems, and advances in the field of formal verification, extensive research efforts have been invested in formalizing the UML. The main goal is to develop formal semantics for the UML, which will make it possible to apply formal verification methods and tools.

In Damm et al. [3] a kernel of the UML is defined and formalized, by associating a model with a symbolic transition system. Semantics of a richer UML subset is then defined by compiling it into that kernel. The rich subset covers such features as active objects, dynamic object creation and destruction, dynamically changing communication topologies in inter-object communication, asynchronous signal based communication, synchronous communication using operation calls, and shared memory communication through global attributes. While the semantic model of [3] is quite general, the paper suggests certain restrictions on the communication scheme between objects, in order to optimize the verification process.

In contrast, Rhapsody takes a more general approach: rather than imposing restrictions, it allows users to make their own design decisions and supports powerful execution semantics through code generation capabilities. As the impact of formal verification methods increases and verification engines scale up to handle larger systems, we believe that tools like Rhapsody will be modified to

support and take advantage of certain restrictions and semantic idiosyncracies, of the kinds adopted in [3].

A more abstract version of the semantics of [3] appears in [8] by formalization in the langauge of the PVS theorem prover. For more details on other UML verification-driven semantics, e.g., [10, 2] see [3].

Acknowledgements:

We would like to express our deepest gratitude to Eran Gery and Yachin Pnueli for many helpful discussions on Rhapsody and its semantics. Thanks also to the entire Rhapsody development team at I-Logix Israel, Ltd. Finally, we thank one of the referees for his/her helpful comments.

References

1. G. Booch. *Object Oriented Analysis and Design with Applications.* Benjamin/Cummings, California, 1994.
2. E. Borger, A. Cavarra, and E. Riccobene. Modeling the Dynamics of UML State Machines. In *Int. Workshop on Abstract State Machines (ASM'00)*, volume 1912 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2000.
3. W. Damm, B. Josko, A. Pnueli, and A. Votintseva. Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML. In *Formal Methods for Components and Objects (FMCO'02)*, volume 2852 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2003.
4. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987. (Preliminary version: Technical Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, February 1984.).
5. D. Harel and E. Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 30(7):31–42, July 1997. (Also in *Proc. 18th Int. Conf. Soft. Eng.*, Berlin, IEEE Press, March, 1996, pp. 246–257.).
6. D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM TRANS. Software Engineering and Methodology*, 5(4):293–333, October 1996.
7. D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach.* McGraw-Hill, 1998.
8. J. Hooman and M. Van Der Zwaag. A Semantics of Communicating Active Objects with Timing. In *Specification and Validation of UML Models for Real-Time and Embedded Systems (SVERTS'03)*, 2003. Available from the European Project OMEGA homepage http://www-omega.imag.fr.
9. I-logix,inc., products web page. http://www.ilogix.com/fs_prod.htm.
10. G. Reggio, E. Astesiano, C. Choppy, and H. Husmann. Analysing UML active classes and associated statecharts - a lightweight formal approach. In *Proceedings Fundamental Approaches to Software Engineering (FASE'00)*, volume 1783 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2000.
11. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design.* Prentice - Hall, New York, 1991.
12. B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling.* John Wiley & Sons, New York, 1994.
13. UML. Documentation of the unified modeling language (UML). Available from the Object Management Group (OMG), http://www.omg.org.