# 1

# Specification and Analysis of Real-Time Systems: Modechart Language and Toolset

**ALOYSIUS K. MOK and DOUGLAS A. STUART** [1]
*The University of Texas at Austin*

**FARNAM JAHANIAN**
*The University of Michigan*

**ABSTRACT**

This chapter illustrates the use of the Modechart specification language and the MT toolset by using them in the specification and analysis of the control rod system of a nuclear reactor. The specification language and an associated logic, RTL, are introduced, as is a tool for building specifications. Two approaches are then used to analyze the specification. The Modechart simulator is used to exhibit individual behaviors of the system. The Modechart verifier is used to verify global properties of the system.

## 1.1 INTRODUCTION

Since real-time systems are often safety-critical systems, ensuring the correctness of a real-time system is very important. Since real-time systems are also typically complex systems, with

[1] Supported in part by a research grant from the Office of Naval Research under ONR contract number N0014-94-1-0582, the Texas Advanced Research Program and the Naval Research Laboratory

*Book Title*, Edited by TRENDS Editor

behaviors that depend on the inter-relationships among the timing of the events of the system, testing is inadequate as the sole means for ensuring the correctness of such systems. Formal methods offer the hope of guaranteeing the correct behavior of such systems. In particular, they offer the hope of mechanical techniques which can be used to demonstrate the correctness of a real-time system.

The ultimate goal of mechanical verification would be to have a system designer give a natural language description of a specification to a verification system, and have the verifier say, 'Yes, the system is correct.' This is not possible with the current state of the art, and is likely to be unattainable for some time. Accordingly, we will attempt to construct an interactive methodology where the system designer would create a formal specification, and then guide a mechanical verification system to the conclusion that the system is correct. The basis of the approach will be to begin with a core mechanical verification tool, and surround it with a (continually expanding) number of "helper" mechanisms, and provide the designer with an (ever more effective) interface for using the helper mechanisms to lead the core tool to the correct conclusion. This chapter will focus on Modechart, one formal specification language, and some of the techniques that make up the basis of the core mechanical verification process. Modechart ( [JM94]) is a graphical specification language, influenced by Statechart ( [Har87]). Modechart is intended for describing embedded real-time applications, whose control structure can naturally be described as timed hierarchical state machines. This chapter will also provide an introduction to the Modechart Toolset, which has been implemented jointly at the Naval Research Laboratory and the University of Texas at Austin.

One example will be used throughout this chapter to illustrate the techniques and concepts being introduced. This example first appeared in [JM86]. A nuclear reactor has two control rods, which control the pace of the reaction in the reactor core, and therefore the power output of the reactor. Each rod can be either inserted or withdrawn, and it takes between one and fifty time units for the rod to change positions. For safety reasons, at most one of the rods may be moving at any time. Throughout the remainder of this chapter, this example will be developed and extended to illustrate how Modechart may be used to specify such a system, and how the tools available may be used to discover properties about the system.

Modechart represents the control structure of the system being specified by modes and transitions between them. Figure 1.1 shows a modechart specification for part of the nuclear reactor system example (the figure was generated automatically by the Modechart Toolset from a textual description of the specification). This portion of the specification includes only one of the control rods, and its controller. The specification is a single mode, *nuke*, containing two modes. *Controller1* represents the controller of the rod, and cycles between the modes *idle1*, *req1*, *busy1*, and *done1*. *Req1* represents a request to move the rod being present, and *busy1* represents the controller viewing the rod as being in motion. *Rod1* represents the control rod, and is in *rest1*, *move1*, or *reset1*.

There are three types of analysis that can be performed on such a specification. Modechart specifications can be converted into a formal first order logic, RTL ( [JM86]), and natural deduction can be used to make inferences about the specification, either by hand, or using a mechanical theorem prover of some kind. Since RTL is a first order logic, the whole panoply of natural deduction and theorem provers may be used with minimal modification. This approach will not be further discussed in this chapter.

Another type of analysis useful to a system designer is to examine a typical computation of the specification. The Modechart Toolset provides a simulator for this purpose. Using the simulator allows the user to examine an individual computation of the specification. While

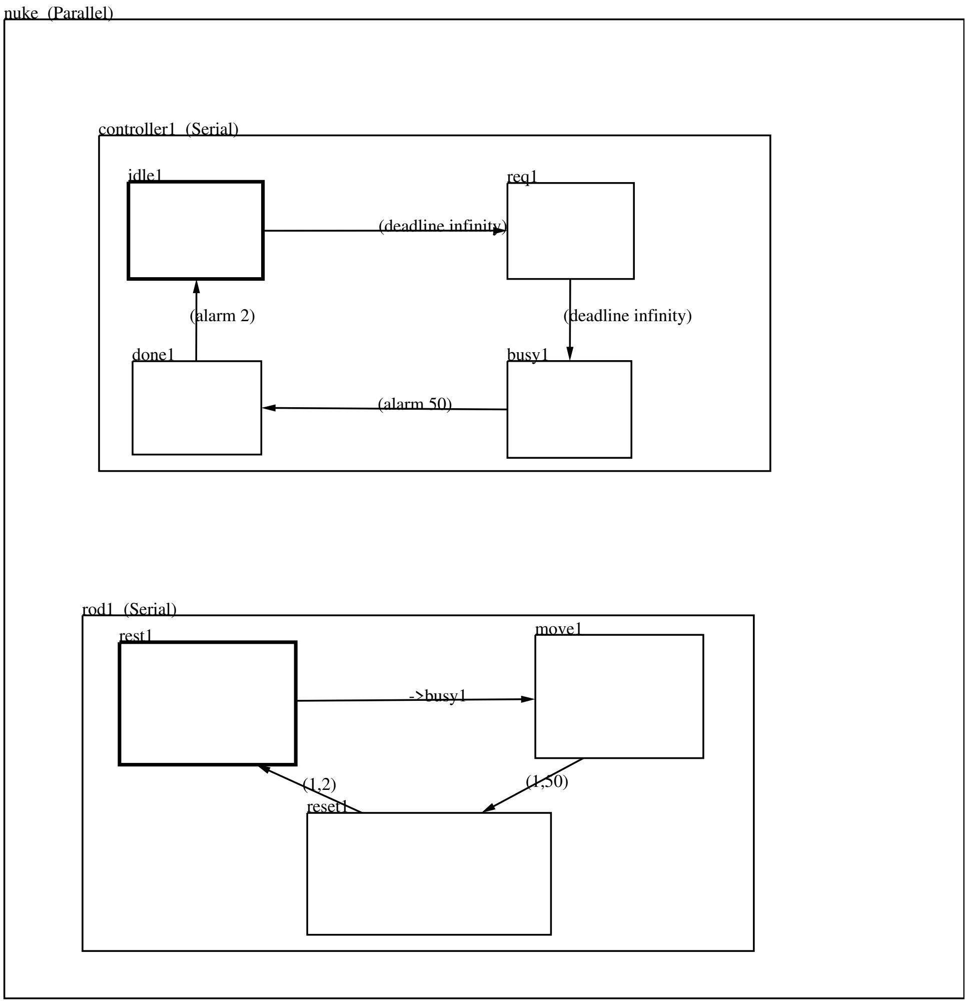


**Figure 1.1** Nuclear Reactor Controller and Rod (figure automatically generated)

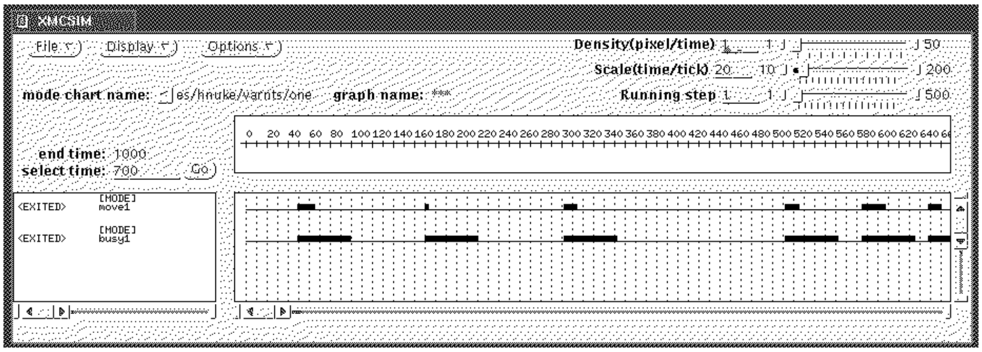


**Figure 1.2** Nuclear Reactor Controller and Rod Simulation

this type of analysis can not provide any guarantees about the correctness of the system, it is often helpful in the process of matching the formal specification with the natural specification. Figure 1.2 shows the the result of simulating the specification of Figure 1.1. The lower two windows indicate when given modes are active. In this case, the top line indicates the intervals when *move1* is active, and the bottom line when *busy1* is active. The active intervals are indicated by a thick bar in the time line to the right of the name of the mode. As an example, the first active interval for *move1* is approximately from time 48 to time 65, while the first active interval for *busy1* is from time 48 to time 98. This particular execution indicates that whenever the rod is moving, its controller is busy. For this computation, then, the modechart specification does appear to be functioning as intended.

The third type of analysis, also supported by the Modechart Toolset, is verification. Although the simulation of Figure 1.2 indicates that the specification does behave as desired in one computation, it can offer no guarantees that the specification is actually correct, since it does not rule out the specification behaving incorrectly in some other computation, or even at some later point in the computation being simulated. To actually prove the correctness of the specification it is necessary to examine all of the computations of the specification. The initial approach to verification provided by the Modechart Toolset is that of model checking. The model checking approach involves constructing a finite model of all of the behaviors of the specification, and then examining that model to ensure the correctness of all of the behaviors so represented. The particular finite model representation used by the Modechart Toolset verifier is the *computation graph*. Figure 1.3 is the computation graph of the specification of Figure 1.1. Each point in the computation graph is a configuration of the specification, and each edge in the graph represents a transition of the specification. Using this computation graph, the Modechart verifier can be used to prove that whenever mode *move1* is active, mode *busy1* is active as well. This ensures that the specification behaves correctly at all times in all computations, while the simulation just indicated the correctness of the specification in a segment of one computation.

This section has provided a brief introduction and overview of the specification language Modechart, and the analysis techniques and tools that can be used to analyze Modechart specifications. The remainder of this chapter will provide a more detailed tutorial introduction. The next section will briefly introduce the computation model used by Modechart. Section 1.3

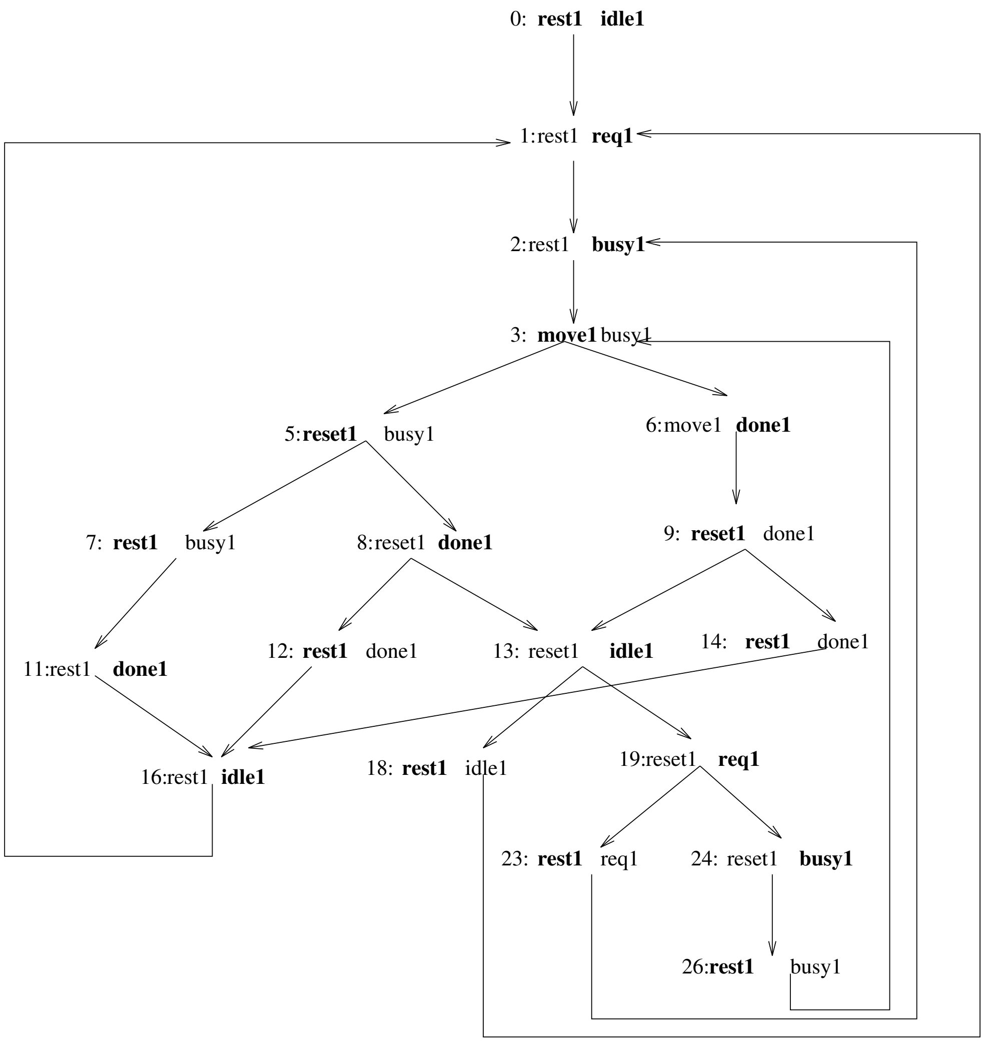


**Figure 1.3** Nuclear Reactor Controller and Rod Computation Graph

will introduce Modechart. Sections 1.4 and 1.5 will introduce the Modechart simulator and verifier respectively. Section 1.6 will explore the nuclear reactor example in more detail.

## 1.2 THE MODECHART COMPUTATION MODEL

A specification language such as Modechart provides a means of describing real-time systems in a precise way, and for formally showing that a specification in the language satisfies certain properties. Although some properties may be purely syntactic, such as the size of the specification, or the number of processes or agents, most properties of interest are semantic properties, that is, requirements on the behaviors of the specification. Before proceeding to a more detailed introduction to Modechart, the computation model that will be used to represent the computations of a Modechart specification must first be introduced.

Modechart regards a specification as a collection of events. The building block of a Modechart computation is the event occurrence. An event occurrence is a time-stamped event, representing the occurrence of the event at the indicated time. A computation of a Modechart specification is a set of time-stamped event occurrences. Since each specification is finite, there are only a finite number of events in each specification, so an infinite computation will contain an infinite number of occurrences of at least one event. The formal definition of what it means for a computation to satisfy a modechart can be found in [JM94].

Intuitively, a computation can be regarded as a sequence of sets of event occurrences, where all of the event occurrences in the same set have the same time-stamp. Since Modechart is primarily concerned with timing properties of specifications, in the sense of properties that can be expressed in terms of the times of various event occurrences, the only ordering on the events of a computation considered by Modechart is the ordering introduced by the time-stamps on the individual event occurrences. As a consequence, event occurrences with the same time-stamp are truly considered to be simultaneous, neither is considered to precede the other. This feature of Modechart computations must be borne in mind when writing Modechart specifications, and some of the consequences of this will be seen in subsequent sections.

As a final note on computations of Modechart specifications, Modechart uses a discrete time model. This means that the events in a computation occur at integer times, and more importantly, with a fixed minimum separation between non-simultaneous events. This is in contrast to continuous time, where event occurrences could have real time-stamps, and so non-simultaneous events could be arbitrarily close.

## 1.3 THE MODECHART SPECIFICATION LANGUAGE

As indicated in section 1.1, Modechart is a hierarchical graphical specification language for real-time systems. The two basic elements from which Modechart specifications are constructed are *modes* and *transitions*. Modes represent the control state of the system being specified. Transitions represent the flow of control of the system being specified.

Each mode in a specification, or modechart, must be one of three types. *Atomic* modes have no internal structure, and are the basic building blocks of a specification. Referring to Figure 1.1, the atomic modes are *rest1*, *reset1*, *move1*, *idle1*, *req1*, *done1*, and *busy1*. Each represents a primitive control state of the system.

*Serial* modes have internal structure. Each serial mode has at least one child mode, and has a designated child mode which serves as an *initial mode*. A serial mode represents the sequential composition of its children, and the children of a serial mode are considered to be in series. Accordingly, at any time a serial mode is active, exactly one of its child modes must also be active. The initial mode of the serial mode is the mode that is considered to be entered when the serial mode itself is entered, unless some other child of the serial mode is entered explicitly. Referring again to Figure 1.1, the serial modes in the specification are *rod1* with initial mode *rest1*, and *controller1*, with initial mode *idle1*. In figures, the initial mode of a serial mode will be designated with a **bold** outline. In any computation of the specification of Figure 1.1, whenever *rod1* is active, exactly one of *rest1*, *reset1* and *move1* will be active.

*Parallel* modes also have internal structure. Each parallel mode as a number (possibly zero) of children. Although a parallel mode with zero children is syntactically legal, it is in practice equivalent to an atomic mode. A parallel mode represents the parallel composition of its children, and the children of a parallel mode are considered to be in parallel. Accordingly, at any time a parallel mode is active, all of its child modes must also be active. Turning again to Figure 1.1, there is only a single parallel mode, *nuke*, which represents the entire specification. The children of *nuke* are *rod1* and *controller1*.

While modes represent the control state of a system, the control flow among control states is represented by *transitions*. Graphically, a transition is a directed edge between modes. Each transition therefore has a source mode and a destination mode. Taking a transition from one mode to another represents control leaving the state represented by the source mode and being transferred to the destination mode, and is considered to be instantaneous. Since all of the children of a parallel mode must be active when the parallel mode is itself active, transitions are only possible between children of a serial mode, that is, modes that are in series. However, since Modechart is a hierarchical language, a transition could have its source mode inside one child mode of a serial mode, and its destination inside a different child of the same serial mode. Accordingly, this requirement is more properly that the first common ancestor of the source and destination modes of a transition must be a serial mode, or the transition must be a self loop, and the parent of the source and destination mode must be a serial mode. Recalling the modechart specification in Figure 1.1, there are three transitions among the children of *rod1*, and four transitions among the children of *controller1*. There is a transition in *rod1* with source mode *rest1* and destination *move1*.

Before turning to the question of when transitions can be taken, which is a semantic rather than syntactic issue, it is necessary to provide the link between the syntax of Modechart being described here, and the computation model presented in the preceding section. Since the computation model views a system as a collection of events, it is necessary to identify the events associated with modes and transitions. Since an event is a point in time at which something occurs, a mode being active is not an event. The events associated with modes are the entry and exit of the mode. Syntactically, the entry event for mode *M* is denoted by $\rightarrow M$, and the exit event for mode *M* is denoted by $M \rightarrow$. The event associated with a transition is the event of the transition being taken, known as the transition event. The transition event for a transition from mode *M1* to mode *M2* is represented as $M1 \rightarrow M2$. Every computation of a modechart will consist solely of mode entry and exit events and transition events. Accordingly, every computation of the modechart of Figure 1.1 will consist of one or more occurrences of each of the 27 distinct events, the 20 entry and exit events of the 10 modes in the modechart, and the 7 transition events of the 7 transitions.

Having defined the events of the specification, in order to define the computations of the specification, it remains only to establish when transitions among modes are taken. Conditions are associated with transitions to govern when they may be taken. Each transition has exactly one condition associated with it, and that condition is in disjunctive normal form. Each disjunct is either a *timing condition* or a *triggering condition*.

A timing condition consists of a *delay* and a *deadline*. The delay and deadline are both non-negative integers, and the deadline must be greater than or equal to the delay. A timing condition with delay *r* and deadline *d* is denoted *(r,d)*. There are also a number of abbreviations for some special timing constraints. *(delay r)* is an abbreviation for $(r, \infty)$. *(alarm r)* is an abbreviation for $(r, r)$. *(deadline d)* is an abbreviation for $(0, d)$. Referring again to Figure 1.1, the condition on the transition $move1 \rightarrow reset1$ is the timing constraint *(1, 50)*, representing a delay of 1 and a deadline of 50, and the condition on the transition $done1 \rightarrow idle1$ is *(alarm 2)*, representing a delay and deadline of 2. A timing condition is satisfied if the source mode of the transition with the timing condition has been active at least as long as the delay of the timing condition.

A triggering condition is a conjunction of events and predicates, which is satisfied when all of the events in the triggering condition occur while all of the predicates are also true. A triggering condition is considered to have a deadline of zero with respect to the time at which the condition becomes true. This is in contrast with the deadline (and the delay) of a timing condition, which is with respect to the time at which the source mode is entered. Each conjunct of a triggering condition must be one of the following:

- The event $\rightarrow M$ is satisfied when mode $M$ is entered.
- The event $M \rightarrow$ is satisfied when mode $M$ is exited.
- The event $M1 \rightarrow M2$ is satisfied when the transition $M1 \rightarrow M2$ is taken.
- The predicate $M == true$ is satisfied if mode ***M*** is active.
- The predicate $M == false$ is satisfied if mode ***M*** is not active.
- The mode list predicate $\{$*(M1, . . . , MN)*$\}$ is satisfied if any of the modes in the list are active.
- The before list predicate $\{< M1, \ldots, MN)\}$ is satisfied if any of the modes in the list are active and have been active for at least one time unit.

Note that at the time instant when a mode is entered or exited, the mode is considered to be both active and inactive. It is important to consider this when constructing triggering conditions on transitions.

The only triggering condition in Figure 1.1 is on the transition $rest1 \rightarrow move1$, and consists of the single event $\rightarrow busy1$, which is the entry event for the mode *busy1*.

A transition may only be taken at a time at which one or more of its condition's disjuncts, of either type, are satisfied. A condition being satisfied is a necessary condition for the transition to be taken, but does not force the transition to be taken. The deadline on a transition does however impose a requirement that the source mode of the transition must be exited at or before the time indicated by the deadline of the condition. The delay on a timing constraint, and the condition of a triggering condition, can therefore be regarded as permitting the transition to be taken, while the deadline on either type of condition is a requirement for the source mode to be exited by some transition, not necessarily the transition with the deadline. This distinction is only relevant, though, if there is more than one transition that exits the source mode. A deadline of infinity therefore imposes no restriction at all, and indicates that the transition need never be taken. Therefore, a timing condition $(0, \infty)$ indicates a transition that may be taken with complete discretion.

Referring to Figure 1.1, $rest1 \rightarrow move1$ may be taken at any time the event $\rightarrow busy1$

occurs, that is when the mode *busy1* is entered, and any time $\rightarrow busy1$ occurs, the source mode of the transition, *rest1*, must be exited. Since there is only the one transition leaving *rest1*, the transition to be taken must be $rest1 \rightarrow move1$. Similarly, the transition $move1 \rightarrow reset1$ may be taken one time unit after mode *move1* is entered, and must be taken no later than 50 time units after *move* is entered.

As indicated, the effect of taking a transition is to exit the source mode of the transition and to enter the destination mode. Taking a transition must maintain the relationships among serial and parallel modes and their children established previously. A transition actually exits the child of the serial mode that is the least common ancestor of the source and destination modes that contains the source mode, and likewise enters the child of common ancestor that is the ancestor of the destination mode. In addition, a transition exits all active children of any mode that is exited. A transition enters all of the ancestors of the destination mode that are also descendants of the common serial mode. Also, for each parallel mode entered, all of its children are entered (and considered active). For each serial mode entered its initial mode is entered, unless one of its children is an ancestor of the destination mode. A mode that is exited or entered by a transition is said to be *explicitly* exited or entered if the edge representing the transition originates or ends at the mode, or a descendant of the mode. Any other mode exited or entered by the transition is said to be *implicitly* exited or entered. In the case of Figure 1.1, no transition implicitly enters or exits any mode. The transition $rest1 \rightarrow move1$ explicitly exits the mode *rest1*, and explicitly enters the mode *move1*. In Figure 1.10, *control_fl*, *init_cf*, *rod1_fl*, *init_flr1*, *rod2_fl* and *init_flr2* are all entered implicitly by the transition $ready\_ch \rightarrow full$, which explicitly exits *ready_ch*, *control_hf*, and *half*, and implicitly exits *rod1_hf*, *rod2_hf*, and their active children.

Finally, to complete this discussion of the semantics of modechart specifications, the *root mode* is the outermost mode in the specification, and the only mode which has no parent. Each computation begins with the entry of the root mode at time 0, together with those modes that would be entered if the root mode were explicitly entered by a transition, that is, all children of each parallel mode entered, and the initial child of each serial mode entered. The remaining events in the computation are the result of transitions that are taken.

The root mode of the specification in Figure 1.1 is *nuke*. The modes initially entered are *nuke*, *controller1*, *idle1*, *rod1*, and *rest1*, so any computation of this modechart must include the first occurrences of the events $\rightarrow nuke$, $\rightarrow controller1$, $\rightarrow idle1$, $\rightarrow rod1$, and $\rightarrow rest1$ among those that occur at time zero. The computation that consists solely of the entry events of the initial modes is one computation of this specification, since the only transition that can be satisfied by this collection of events, and has an active source mode, is the transition $idle1 \rightarrow req1$. This transition has timing condition $(deadline \infty)$, therefore no other event need even occur. Another possible computation would result if the transitions $idle1 \rightarrow req1$, $req1 \rightarrow busy1$, and $rest1 \rightarrow move1$ all occur at time 100, $move1 \rightarrow reset1$ and $busy1 \rightarrow done1$ both occur at time 150, and $reset1 \rightarrow rest1$ and $done1 \rightarrow idle1$ both occur at time 152.

The Modechart Toolset, MT, includes a tool for creating, modifying, and browsing Modechart specifications. The tool can also automatically generate Postscript versions of specifications, and this facility has been used to generate the modechart in Figure 1.1. The tool consists of four windows. The Modechart window controls the top level behavior of the tool. The Specification window controls specification level behavior, such as selecting an analysis tool to apply to the specification. The Locator window provides an overview of the entire specification, and serves as a scrollbar. The Work window provides a detailed view of the

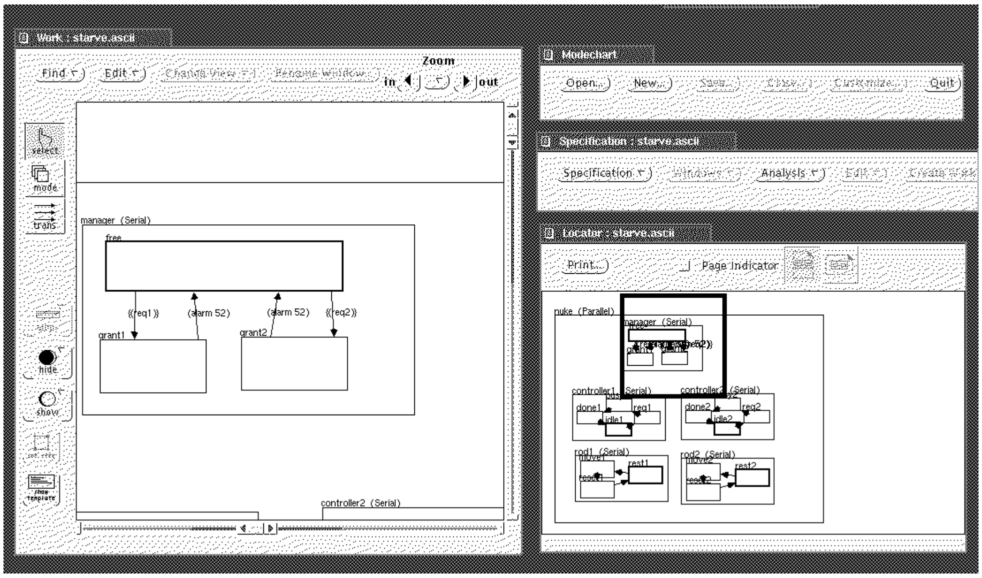


**Figure 1.4** The Modechart Toolset

portion of the specification specified by the Locator window, and is used to manipulate the specification. The nuclear reactor example, a part of which is represented by the modechart of Figure 1.1, consists of two rods and two controllers, and a manager which ensures that the two rods are not in motion at the same time. Figure 1.4 shows one possible specification for the entire nuclear reactor system loaded into the toolset.

The modechart in Figure 1.4 consists of the parallel root mode *nuke*, which has five serial children. The two rod controller pairs, *controller1* and *rod1* and *controller2* and *rod2* are similar to *controller1* and *rod1* from Figure 1.1, and differ only from the specification in Figure 1.1 in the condition on the transition from *reqi* to *busyi*. The fifth serial mode, *manager*, is new, and its function is to prevent the two rods from moving simultaneously. It does so by granting permission to the controllers to begin moving. The manager consists of three modes. The mode *free* represents neither rod having permission to move. The mode *grant1* represents *rod1* having permission to move, so that the condition on the transition $req1 \rightarrow busy1$ is $\rightarrow grant1$. Similarly, *grant2* represents *rod2* having permission to move. Since each control rod moves for at most 50 time units once it has permission to move, the condition on the transitions back into *free* is the timing condition *(alarm 52)*. Choosing 52 rather than 50 ensures that there is a one time unit margin before and after a rod's movement during which the other rod does not have permission to move. 50 time units is sufficient to guarantee that there is no positive length interval during which both rods could be moving, but would allow for the possibility that one rod would begin moving the instant that the other stopped moving.

The design decision that must be addressed carefully is the conditions on the transitions into each granting mode. One possibility would be the entry event for the corresponding requesting mode. This is not a good choice, since an event condition of this type is only satisfied at the instant that the requesting mode is entered. Therefore, if a rod requested permission to move when the manager was unable to grant it, that rod would never gain permission to move, since the requesting mode could not be entered again until it had already received permission and moved the rod. In this case, a predicate triggering condition is necessary. There are a number of possible choices, one being a mode list predicate consisting only of the request mode. Accordingly, as in the figure, the condition on the transition $free \rightarrow grant1$ is $\{$*(req1)*$\}$, and similarly for the condition on $free \rightarrow grant2$.

## 1.4 THE MODECHART SIMULATOR

The preceding section concluded by introducing one possible specification for the nuclear reactor system. All of the elements required by the specification, two rods, two controllers, and a manager, are present. What are the behaviors of this specification? The intended behavior is that each rod controller should seek permission to move its individual rod, wait for permission from the manager, begin moving the rod, and then return to the idle state, eventually issuing another request. Also, the manager should ensure that at most one rod is moving at any time. Does the specification in fact exhibit this behavior? The answer to this question requires analysis of the computations of the specification.

As indicated, the Modechart Toolset supports two main semantic analysis tools, a simulator and a verifier. The simulator allows examination of hopefully representative computations, while the verifier considers all computations. First the simulator will be used to generate an individual computation for examination, to see if the computation conforms to the expected behavior.

The Modechart Simulator generates a single execution of the system as specified by a modechart specification. It begins by generating the entry events for all of the initial modes of the specification. Then, each time instant, including time 0, it attempts to take eligible transitions. Each time instant, the set of transitions with satisfied conditions is generated. A transition from this set is chosen and executed, changing the active modes and the current events. After each transition is executed, the set of transitions with satisfied conditions is updated by removing transitions with source modes that are no longer active, and adding transitions with conditions newly satisfied by the events caused by the just executed transitions, and those transitions from newly active modes with satisfied transitions. This process is repeated until either no more transitions with satisfied conditions are available, or no transition with a satisfied condition has a zero deadline, and the simulator elects not to take any of the remaining eligible transitions. In either case, the clock is then incremented to reflect the passage of a single time unit, and the process of taking transitions is begun again.

Since the simulator produces only a single execution, and a modechart specification will generally have more than one execution, the simulator must select a single execution from multiple candidates. As indicated in the preceding section, there are two types of choice the simulator must make. The first is a choice among competing transitions which exit the same source mode. The second is when not to take a transition with a satisfied condition which has an unexpired deadline. This is equivalent to deciding when to schedule a transition with a timing condition.

The simulator accepts as input, in addition to the specification, an *options file*. This file allows the user to give instructions to the simulator on how such non-deterministic choices are to be made. Four choices are available to the user for each transition. The user may designate that a transition is always to be taken as soon as possible (as soon as the delay transpires), as late as possible (only when the deadline is reached), at a specific time (always after 10 time

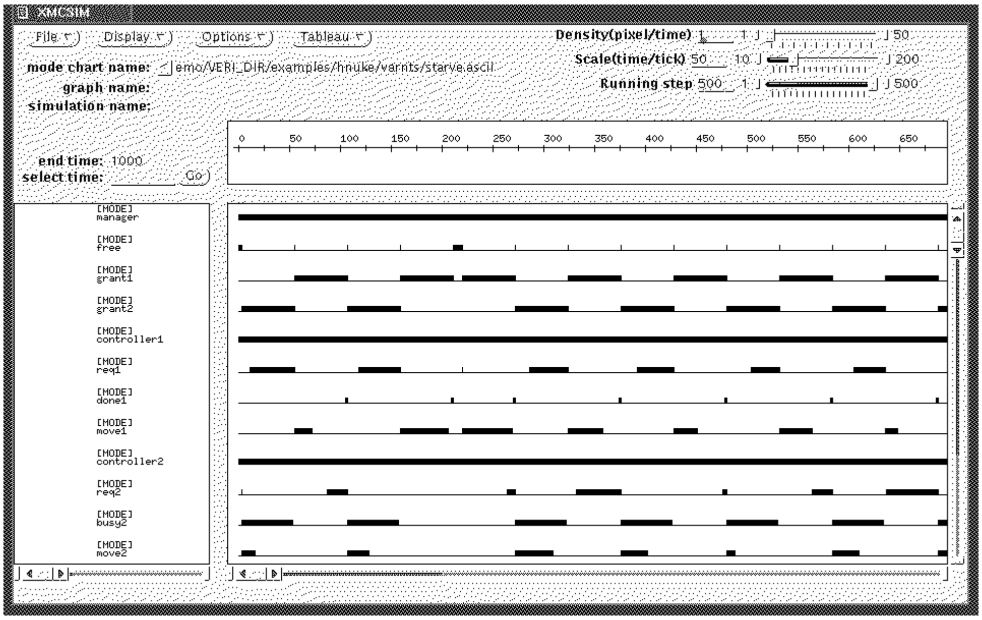


**Figure 1.5** Simulating the Nuclear Reactor

units), or randomly, in which case the simulator makes a choice from a random distribution. The options file is supplied by the user at the beginning of a given run of the simulator.

The options file also provides the user with a means for determining the length of the simulation. The user may supply an absolute length, a step size, and breakpoints. The run length determines how long the simulation will run without user direction for further execution. The step size allows the user to step through the execution at a desired time granularity. Breakpoints, analogous to breakpoints in debuggers, allow the user to designate events the occurrence of which will cause the simulation to halt.

Figure 1.5 shows the result of simulating the nuclear reactor specification of Figure 1.4 for 1000 time units. The simulator window contains four major areas. The top area contains the various controls and menus for the simulator. The three display areas show the results of the simulation graphically. The top display area shows a simple time-line, the scale of which can be varied by the user. The bottom two areas display the activity of the system specified during the simulation. The left window is a legend for the main display area. For each mode and transition in the left window there is a line representing the behavior of that mode or transition in the right window. In the case of a transition, the line represents when the transition's condition is satisfied, and when it is taken. For a mode, a thin line indicates that the mode is not active, while a thick line indicates that the mode is active. Which individual modes and events are tracked in the display area is also determined by the options file.

In the case of Figure 1.5, all of the objects being tracked are modes. The top four modes are *manager*, *free*, *grant1*, and *grant2*. Since *manager* is a child of *nuke*, which is the root mode, and a parallel mode, it will always be active, indicated in the figure by the line for the mode being thick throughout the simulation. Accordingly, no information is conferred by including it in the monitored set of modes. However, doing so can be helpful by providing a visual separator. The three lines directly below the first solid line are the lines for the children of that mode. Note that the manager is seldom in *free*, and appears to grant requests by both rods.

The middle four lines in Figure 1.5 represent some of the modes associated with the first rod. The modes in question are *controller1* as a separator, and then *req1*, *done1* and *move1*. Examining this portion of the display, it would appear that the rod only moves following a request, and that all of the requests are being granted, although generally after a delay, that the rod then moves, and only enters done after the rod has finished moving, as intended.

The final four lines in Figure 1.5 represent modes associated with the second rod. In this case the modes in question are *controller2*, again as a separator, *req2*, *busy2*, and *move2*. In this case, examining the display indicates that the second rod also follows the request-move pattern, although it tends not to wait as long as the first rod for its requests to be granted. Also, the controller does move the rod only when mode *busy2* has been entered.

This computation behaves as anticipated. Examining the display in its entirety, each rod moves only while it has been granted permission by the manager, since each move mode is active only while the corresponding grant mode is active. The active intervals for the two move modes do not appear to overlap. Also, each of the rods gets permission to move each time permission is requested; neither control rod appears to starve. All of these, however, are only observations based on examining a single execution. This illustrates the advantages and disadvantages of simulation. It provides insight into the behavior of the system being simulated, and allows for the formation of hypotheses, but it does not make any guarantee about universally valid attributes of the specification, as indicated in the next section.

## 1.5 THE MODECHART VERIFIER

The previous section used the Modechart simulator to examine the nuclear reactor specification. Although such examination may provide valuable insight into the behavior of the system being specified, and be valuable in other ways, it can not guarantee the correctness of the specification. Simulation produces a single presumably typical execution. Correctness requires the examination of all executions. Since most modecharts have a number of computations, simulation is inadequate for the task of verifying the correctness of a specification. The Modechart Toolset provides a verification tool for this purpose. Figure 1.6 shows the graphical interface to the verification tool.

There are a number of verification techniques available for real-time systems. The Modechart Verifier performs verification using *model checking*. The model checking approach is to construct a finite representation, or model, that encompasses all of the computations of the system being verified. The desired correctness properties are then checked against the model. If the property holds for the model, then it must hold for all of the computations of the specification. Model checking therefore requires both a finite representation of the computations of a specification, and a way of representing the properties to be checked.

As indicated in the introduction, the model for the computations of a Modechart specification is the *computation graph*. A computation graph is a directed graph. The vertices, called *points*, represent points in time at which events occur. Accordingly, points can also be considered to represent the configuration of the system after the occurrence of the event represented by the point. Vertices are labeled with the modes active at the time represented by the point, as in Figure 1.3. Modes that are always active are omitted from the labels of points in the figures. The edges represent the occurrence of transitions. The mode entered at the point, by the transition or transitions entering the point, is indicated in the figure by having its name appear in **bold**. Point number *1* is labeled 1: **req1**, rest1, indicating that in point 1, the modes *req1* and *rest1* are active, and the point represents the entry of *req1* as a result of taking the transition $idle1 \rightarrow req1$.

Since transitions are instantaneous, and there may be several transitions at the same time, time must pass in the configurations represented by the points of the computation graph. The amount of time that may be spent in a given point is determined by the *timing constraints* imposed on the transitions which can be taken from the point. These timing constraints include any timing conditions on transitions from the point, and along a sequence of transitions, or path, leading to the point, as well as the effect of the implied deadline on any triggering conditions. For any path through the graph, these cumulative effects are captured using a separation graph, and may be calculated as a path weight in such a graph. Using this information, the transitions which can actually be taken from a given point, and so appear as edges from the point in the computation graph, may be identified. Points may be distinguished by their label, or by being subject to different timing constraints. For example, in Figure 1.3, points 12 and 14 have the same label, but are subject to different timing constraints: point 16 may happen 2 time units after point 12, but only 1 time unit after point 14.

As an example, returning to Figure 1.3, there are two transitions from modes active in point 6, the transition $move1 \rightarrow reset1$, which appears in the figure, and the transition $done1 \rightarrow idle1$, which does not appear. The reason it does not appear, is that the timing constraints that lead up to point 6 force the transition $move1 \rightarrow reset1$ to happen at the same time as point 6, while the timing condition on $done1 \rightarrow idle1$ ensures that it must happen strictly later than point 6, so that $done1 \rightarrow idle1$ may not happen at point 6, otherwise the transition $move1 \rightarrow reset1$ would miss its deadline.

Model checking also requires a formal representation for the properties to be checked. The Modechart Toolset uses the first order logic RTL, Real Time Logic ( [JM86]), to formally represent properties of Modechart specifications. In fact Modechart was originally developed as a way of organizing system specifications written in RTL. RTL is a first order logic, consisting of the usual first order connectives and quantifiers, plus the relations and operators of integer addition. Multiplication is not allowed, except for multiplication by a constant as an abbreviation for repeated addition.

The link between the logic and the computations of a modechart specification is provided by the *occurrence function*. The occurrence function, denoted by @, is a function with two arguments. The first argument is an event, and the second argument is an integer. The value of the function is the time of the indicated occurrence of the event. For example, $@(\rightarrow req1, i{+}4)$ is the time of the *i + 4th* entry into the mode *req1*.

The key correctness property of the nuclear reactor example, that both of the rods may not be in motion at the same time, can be written

$$\forall i \forall j @(move2 \rightarrow, j) \leq @(\rightarrow move1, i) \vee @(move1 \rightarrow, i) \leq @(\rightarrow move2, j).$$

It states that for every interval in which rod one is moving, so that *move1* is active, every interval in which *move2* is active either ends before the activation interval for *move1* begins, or starts after the interval for *move1* has ended. The result of instantiating this formula as a query for the verifier of the Modechart Toolset can be seen in Figure 1.7

The current verifier implements a number of queries which check the computation graph to determine the satisfiability or validity of formulas which match templates supplied by the verifier for modecharts satisfying appropriate technical restrictions. Formula templates provided cover a wide range of useful properties, including mode activation interval exclusion, such as the correctness formula for the nuclear reactor example above (Figure 1.7), mode activation interval inclusion, event minimum and maximum separation, and the minimum and maximum activation intervals of a mode, which will be used to determine the presence or absence of starvation for the nuclear reactor example at the end of this section (Figure 1.8). Future work will extend the subset of RTL for which such templates are provided, although arbitrary RTL formulas can not be checked, since it was shown in [Jah89] that the model checking problem for arbitrary modecharts and arbitrary RTL formulas is undecidable.

Figure 1.6 shows the verifier tool of the Modechart Toolset after constructing the computation graph for the nuclear reactor modechart. The top section of the tool includes the various command menus, and displays information about the specification and the computation graph. The middle section is used to select and identify various elements of the specification and computation graph that can be used to build queries which can be applied to the computation graph. The bottom section is an output window for text output. In the figure, it contains statistical information about the computation graph, including the number of points and edges in the graph, 100 and 334 respectively.

Figure 1.7 shows a query window for the formula template corresponding to the mutual exclusion property for rod movement, instantiated to the formula above, and with the result indicated that the formula is in fact true, as intended, and as indicated by the simulation. Figure 1.8 shows a second query window. This one indicates the minimum and maximum times the mode *req1* may be active. Although the minimum time is 0, the maximum time is infinity, indicating that rod one could starve, contrary to the indications of the simulation. This example illustrates the advantages of simulation, that it can provide insight into the operation of the system being specified, and indications of its correctness, but that it can also be misleading, as in this case, where the simulation did not exhibit starvation that verification did uncover.

## 1.6 A SPECIFICATION FOR A NUCLEAR REACTOR

The previous sections introduced the Modechart specification language and the Modechart Toolset using the nuclear reactor example. This section will further develop the nuclear reactor specification to illustrate how Modechart and the Toolset can be used to develop a specification.

The nuclear reactor specification of Figure 1.4 was correct in that it prevented both control rods from moving simultaneously. However, as revealed by the verifier in the preceding section, it is possible for the manager to starve either of the control rods. Examining the specification, the reason for this becomes clear. The *manager* is ready to grant permission to a control rod at 52 time unit intervals. It also takes 52 time units for a rod controller to make the cycle from request to request, so that it is possible for a controller to submit a new request at the beginning of every decision cycle for the manager. Since the manager non-deterministically selects between the competing rods, it could forever ignore one of the rods on some computation, because the other rod could continue asking for and receiving permission to move as frequently as possible.

How can a manager be written to prevent starvation? There are several approaches. One approach is for the manager to remember which rod is supposed to next receive permission to move if both are requesting permission. A possible manager modechart that does this is

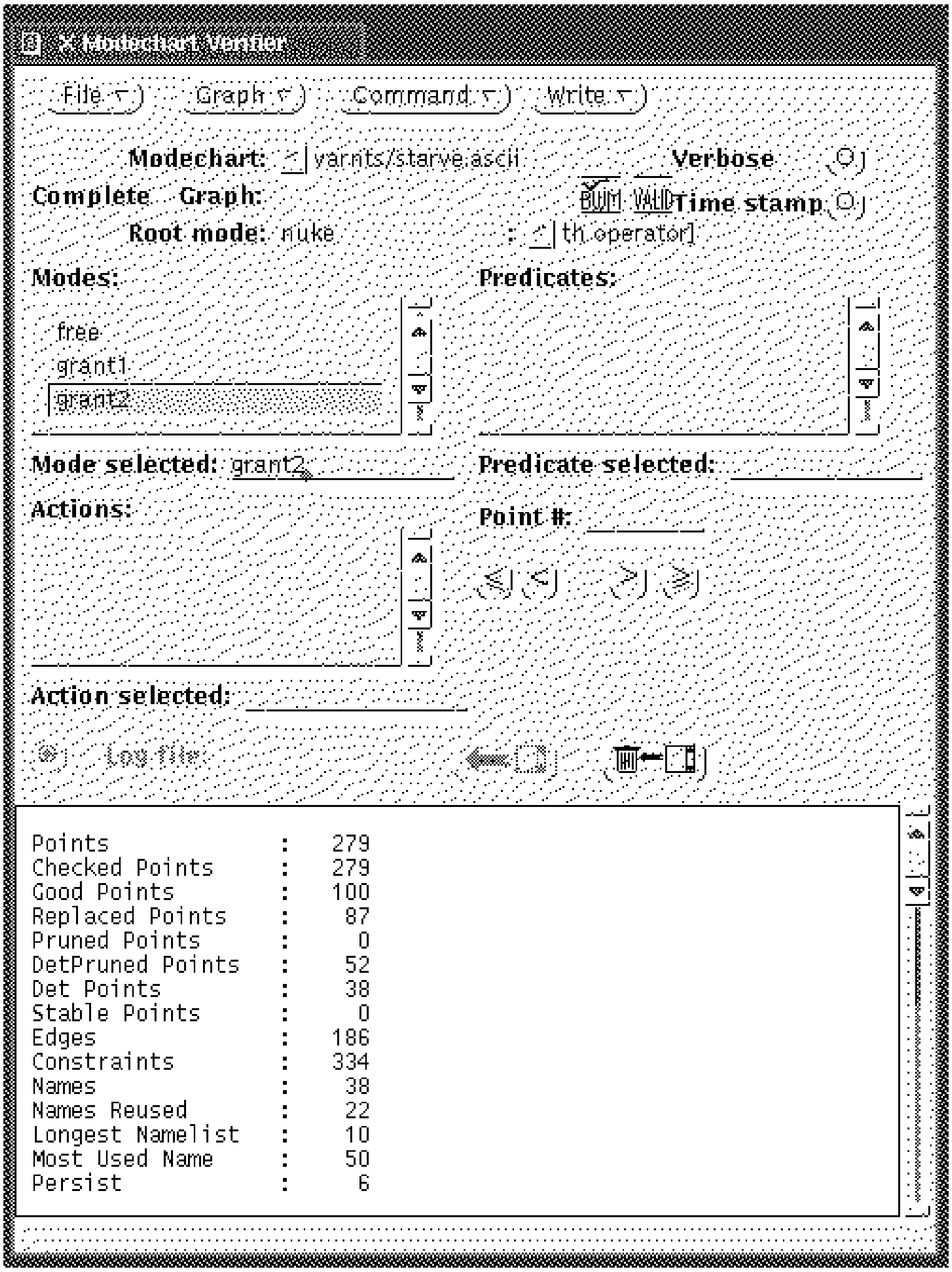


**Figure 1.6** The Modechart Verifier

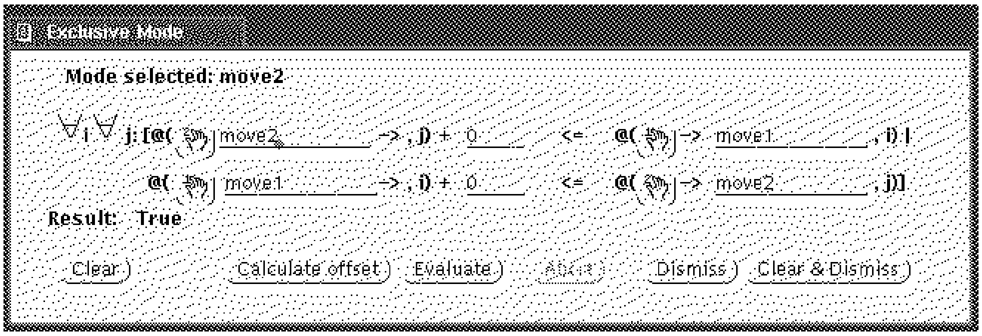


**Figure 1.7** The Modechart Verifier

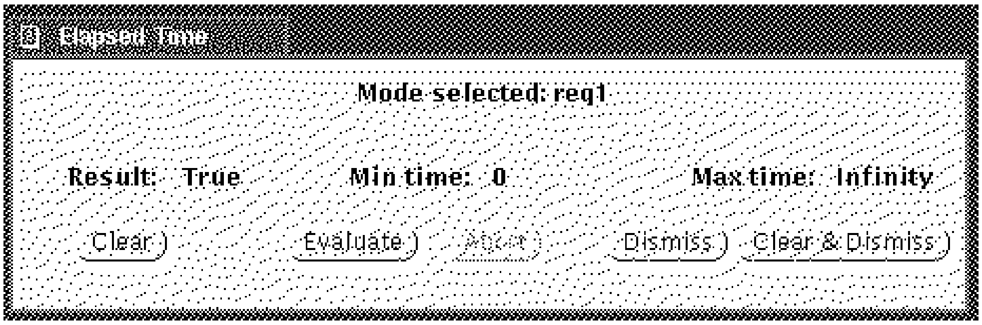


**Figure 1.8** The Modechart Verifier

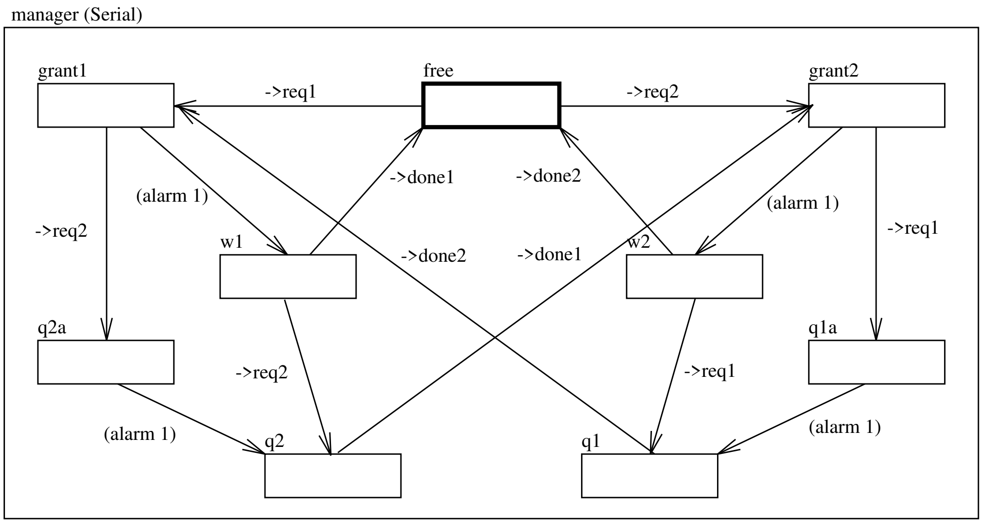


**Figure 1.9** A Starvation Free Manager

given in Figure 1.9. In this modechart, once permission is granted to move a particular rod, if a request for the other rod arrives before the moving rod is finished, the manager moves to a mode reflecting the queuing of this request, in the case of a request for rod2 while rod1 is moving, to *q2*. When the moving rod finishes, either the manager returns to *free* if there is no queued request, or to the appropriate grant mode if there is a queued request. The verifier confirms that this solution is still correct, and also does not permit starvation.

This modechart also illustrates a number of unusual features of Modechart that must be considered when writing Modechart specifications. Note that the queuing modes are essentially split in two. For example, there are modes *q2* and *q2a* that both represent a queuing of *req2*. The second is entered directly from *grant1* on the arrival of a request. The first is entered if a request arrives while in *w1*, which is itself entered one time unit after the request is granted, if a request for rod2 does not arrive earlier. The purpose of this construction is to eliminate the possibility of a *zero cycle*. A zero cycle is a sequence of transitions that can be taken simultaneously, such that the destination mode of the final transition is the source mode of the first transition. Such a cycle could require an infinite number of events to happen in zero time, since the cycle could be repeated as often as desired. Such computations are considered illegal, and a purported Modechart containing such a cycle is not considered to be a valid Modechart. In this case, if the *(alarm 1)* transitions were not present, and there was an edge directly from *grant1* to *free* with condition $\rightarrow done1$, and rod1 requested permission to move immediately when it finished, there would be a zero cycle from *grant1* to *free* and back, since both conditions, $\rightarrow done1$ and $\rightarrow req1$ would be true at that time. Also, if rod1 and rod2 both requested permission at the same time as rod1 finished, the manager could move from *grant1* to *free* to *grant1* to *q2* to *grant2* all at the same time, giving both rods permission to move, violating mutual exclusion of rod movement.

This manager, while it results in a correct and starvation free specification, is complex. It also requires communication between the controllers and the manager both when permission is sought to move a rod, and also when movement is completed, which was not needed by the previous specification. Examining the earlier specification, it can be seen that starvation occurs because a request is granted immediately to any requesting controller, without regard to how long the request has been outstanding. One simple modification to change this results in a specification that is both correct and starvation free. Changing the transition condition on the transition $free \rightarrow grant1$ to $\{< req1)\}$, and changing the condition on the transition $free \rightarrow grant2$ to $\{< req2)\}$ accomplishes this. The reasoning behind this change is that when a a controller re-requests permission at the same time it completes the previous move, the request arrives at the same time as the manager enters *free*. However, since a before list requires the mode in the list be active for the preceding time unit, a rod that has been waiting for permission will satisfy the condition, while the new request will not. This specification is smaller and simpler, and may be better than the one in Figure 1.9.

The modecharts in all of the examples to this point have all been *flat*. Each has consisted of a parallel root mode with a number of serial children, each of which had only atomic children. This section will conclude with an example of a modechart which is not flat, which has a more complex hierarchical arrangement. The previous versions of the nuclear reactor example have all dealt only with ensuring the mutual exclusion and non-starvation of the movement of the control rods in the reactor. The causes of requests to move the control rods have been ignored. Each controller decides with complete freedom when to request permission to move its control rod. This final example will add an operator, shown in Figure 1.10, which decides when the control rods will be moved.

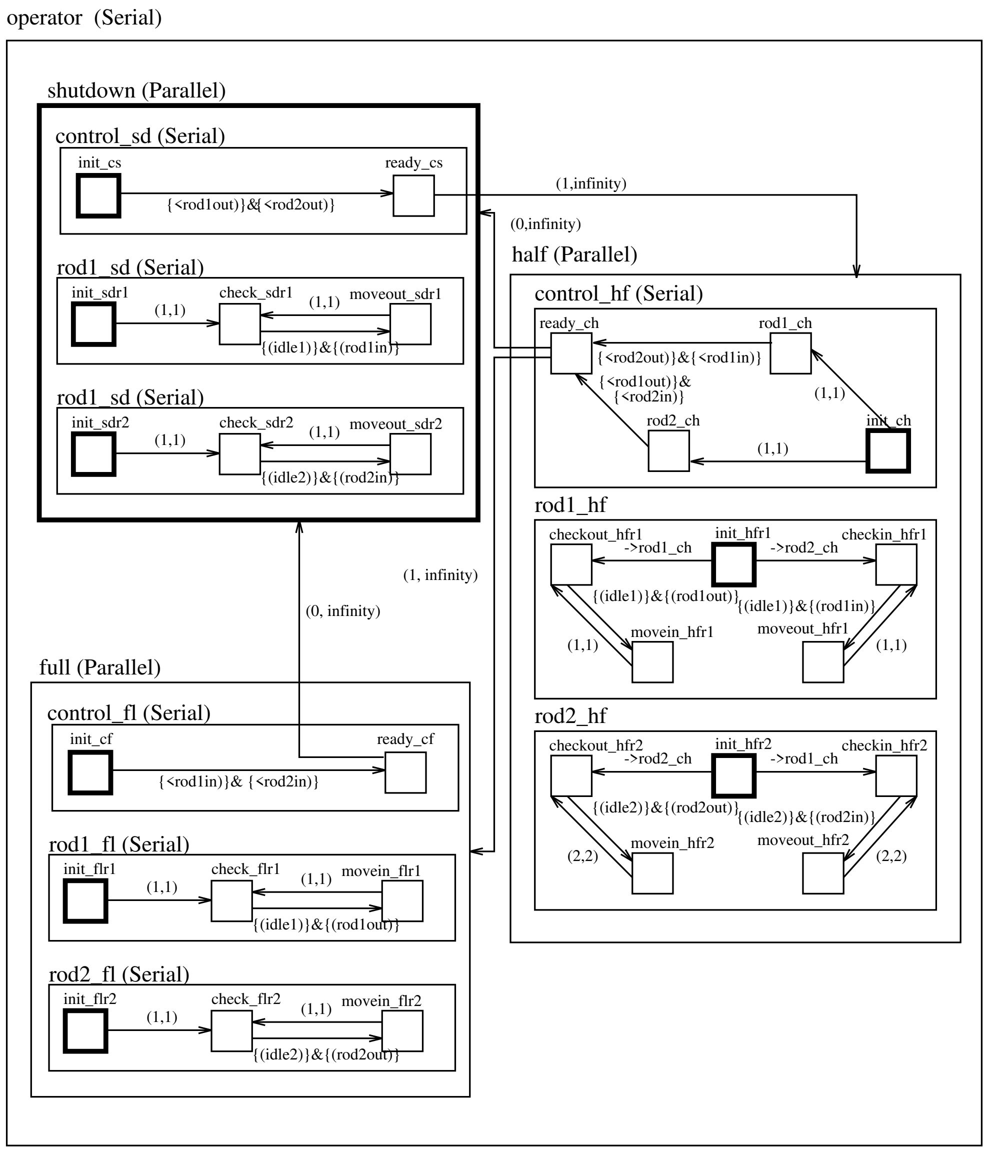


**Figure 1.10** Nuclear Reactor Operator

Since there are two control rods, the operator has three choices for the power level produced by the reactor. Either no rods are moved in, one rod is moved in, or both rods are moved in. In the first case, the reactor is operating at full power, in the second half power, and if both rods are in, the reactor is shutdown. The operator will therefore be represented by a serial mode with three children, one for each mode of operation of the reactor. In this case, however, these three children will not be atomic, since they must govern the operation of the control rods. Since each mode is responsible for two control rods, as well as deciding what to do, each of these modes will be a parallel mode with three children. Looking first at the *shutdown* mode, each control rod must be inserted, so the modes responsible for each rod must just ensure that its rod is in. The mode *control_sd*, which controls the behavior of the shutdown mode, is just a simple serial mode with an initial mode, and a *ready_cs* mode to be entered when both rods are in. The *full* mode is similar. The *half* mode is slightly more complicated, since the decision mode, *control_hf*, must decide which rod will be in and which will be out. Similarly, the controllers for the individual rods must first find out which position their rod should be in before taking any action. Finally, since the current position of the mode is necessary, not just whether or not it is moving, a serial child of the root mode is added for each rod to monitor its location. The modes required for this are not shown.

Note that there are transitions in *operator* which occur within the lowest level serial modes, but also which come from the lowest level atomic modes, and leave the serial mode that contains them, such as the transition from *ready_cs* to *half*. Such a transition leaves not only the source mode, but also any active modes in parallel with it. In the given example, the transition leaves not only *ready_cs* but also *rod1_sd* and *rod2_sd* and their active children implicitly, and *shutdown* itself explicitly.

This final specification has also been subjected to analysis using the Modechart Toolset. We have used the Verification tool to show that this version of the specification is also correct and without starvation.

## 1.7 CONCLUSION

This chapter has illustrated how Modechart can be used to specify a real-time system through various stages of refinement. In this process, two powerful analysis techniques, simulation and verification, were used to examine the correctness of the specification. This analysis lead to insights which were used to create a better final specification. This chapter has also provided a brief introduction to the Modechart Toolset, which includes an interface tool for creating, editing, and browsing specifications, a simulation tool, and a verification tool. The toolset was used to mechanically examine the specification, validating expectations about typical behaviors, and verifying the correctness of the specification over all behaviors. More detailed information about the Modechart specification language can be found in [JLM88] and [JM94]. Simulating Modechart specifications is covered in [CHLR93]. The theoretical framework for the verifier is in [JS88] and [Stu90]. Finally, [RPC94] is a detailed user's guide for the Modechart Toolset.

In the introduction, a methodology was proposed envisioning specification, verification and validation as a collaborative process involving the user and mechanical support. What has been outlined in this chapter is the kernel of such an approach, although there are indications of the collaboration even in this introduction. The property that *rod1* was in mode *move1* only while *controller1* is in *busy1* is verified in the introduction. The same property holds for all of the modecharts in this chapter, and for both control rods, and can be established by verification of the property for each of the control rods for each specification. However, the only impact the remainder of the specification has on this property is in the timing of the transitions $idle \rightarrow req$ and $req \rightarrow busy$. The verification performed in the introduction ensures that the property holds if these transitions occur at arbitrary times. Therefore, the property will also hold if the transitions occur only at designated times, since such computations are included in those already checked, and so verification of this property for the modechart of Figure 1.1 is sufficient to guarantee that it also holds for all of the modecharts of this chapter. This technique of considering only those portions of the specification that can affect a particular property under consideration is known as *focussing*. Also, since all of the modecharts are symmetric in the two control rods, it is also sufficient to check the property for a single control rod.

## ACKNOWLEDGMENTS

The authors would like to acknowledge the efforts of Connie Heitmeyer's group at NRL and Paul Clements, Alex Ho, Jin Yang, Dennis Chen and Helen Wang at the University of Texas in the design and implementation of the Modechart Toolset.

## REFERENCES

[CHLR93] P. C. Clements, C. L. Heitmeyer, B. G. Labaw, and A. T. Rose. MT: A toolset for specifying and analyzing real-time systems. In *Proc. IEEE Real-Time Systems Symposium*, December 1993.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Programming*, (8):231–274, 1987.

[Jah89] F. Jahanian. Verifying properties of systems with variable timing constraints. In *Proceedings of the Tenth Real-Time Systems Symposium*, Santa Monica, California, December 1989.

[JLM88] F. Jahanian, R.S. Lee, and A.K. Mok. Semantics of modechart in real time logic. In *Proc. of 21st Hawaii International Conference on System Sciences*, January 1988.

[JM86] F. Jahanian and A.K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Trans. Software Engineering*, SE-12(9):890–904, September 1986.

[JM94] F. Jahanian and A. K. Mok. Modechart: A specification language for real-time systems. *IEEE Trans. Software Engineering*, 20(12), 1994.

[JS88] F. Jahanian and D. A. Stuart. A method for verifying properties of modechart specifications. In *Proc. IEEE Real-Time Systems Symposium*, pages 12–21, Huntsville, AL, December 1988.

[RPC94] A. Rose, M. Perez, and P. Clements. Modechart toolset user's guide. Technical Report NRL/MRL/5540-94-7427, Center for Computer High Assurance Systems, Naval Research Lab, Washington, D.C., February 1994.

[Stu90] D. A. Stuart. Implementing a verifier for real-time systems. In *Proc. IEEE Real-Time Systems Symposium*, pages 62–71, Orlando, FL, December 1990.