Monitoring and Assertion-Checking of Real Time Specifications in Modechart

Monica Brockmeyer *
Farnam Jahanian
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122
email: {monicab,farnam}@eecs.umich.edu
tel. (313) 936-2974

Constance Heitmeyer
Bruce Labaw
Center for High Assurance Computer Systems
Naval Research Laboratory
Washington, DC 20375
email: {heitmeye,labaw}@itd.nrl.navy.mil

October 26, 1995

Abstract

This paper describes the integration of a monitoring and assertion checking tool into the Modechart Toolset, a suite of integrated tools providing system specification, formal verification, static analysis, and specification simulation for real-time systems. The monitoring and assertion checking tool, MAC, supports monitoring of symbolic execution traces generated by the Modechart simulator, permitting testing of specifications early in the design phase and providing a mechanism for evaluating properties of the system on a particular execution trace. These capabilities are provided by the automatic *translation of assertions in a declarative language (such as Real Time Logic) into monitoring fragments, written in Modechart, which augment the original specification to perform monitoring and assertion checking during simulation. In addition, we discuss several major issues of monitoring and assertion checking in this context. In particular, a key issue is that not all assertions can be evaluated with a bounded event history. A primary goal of this research is the identification of classes of assertions for which the number of events which must be recored for monitoring is independent of the length of the computation to be monitored. We address this problem by exploiting the semantics of Modechart to identify classes of assertions for which monitoring is simplified to evaluation of a state predicate on a finite event history having a bound which is established* a priori.

Index terms: real time systems, specification, symbolic execution, monitoring and assertion checking, formal methods.

*This work is supported in part by the Naval Research Laboratory under Grant N00014-94-P2015.

1 Introduction

Embedded real time systems, such as air-traffic control, patient-monitoring, and manufacturing, have stringent timing and dependability constraints. Ensuring that such systems meet their prescribed specifications is a growing challenge that faces the industry in the coming decade. Several studies have demonstrated that the cost of detecting and removing software errors increases significantly as the development process moves from requirements specification toward production [7]. In fact, the cost of removing an error from a system specification is an order of magnitude smaller than the cost of removing it from a system that is going through integration testing. Other studies have demonstrated that errors in the specification are the most frequent types of software errors and the most expensive to correct [3, 22]. The source of safety-critical failures can often be traced to requirements, specification, or design errors very early in the software development process [20]. It has been argued that formal specification methods can reduce the number of errors in a specification by preventing imprecision and ambiguity. Furthermore, formal specifications are amenable to formal analysis techniques including consistency and completeness checks, validation and testing, and verification.

The Modechart Toolset (MT)[5] is a collection of integrated tools developed by the Naval Research Laboratory together with researchers from the University of Texas to address these challenges. MT supports the formal specification of real time behavior in a language called Modechart [14] and formal analysis via formal verification, simulation, and completeness and consistency checking. The toolset includes facilities for creating and editing Modechart specifications. Users may symbolically execute the specifications with an automatic simulation tool to confirm that the specification is consistent with their intent. They may also invoke a verifier that uses model checking to determine whether the program specifications satisfy any of a broad class of safety assertions. Finally, they may perform a variety of static consistency and completeness queries on their specifications.

In this paper, we develop an approach to monitoring and assertion checking in the context of symbolic executions of Modechart specifications. This approach is intended to complement the existing analysis techniques in the Modechart Toolset. We develop a Monitoring and Assertion Checking Tool (MAC) which is tightly integrated into the Toolset. In particular, the MAC tool supports monitoring execution traces as generated by the Modechart Simulator. Furthermore, it provides a mechanism for evaluating properties of the system on a particular execution trace. For example, it can be used to detect an undesirable behavior or violation of a design assumption as an execution trace is generated. By combining evaluation of a property with a graphical representation of the execution trace, the MAC tool and the Modechart Simulator together conveniently provide some assurance of the user's intent during the specification and design phase. The MAC tool can also be used to invoke user-defined handlers upon detection of certain properties. The handlers can be used to change the simulator execution profile or even the system state before the computation resumes. Furthermore, the MAC tool can complement the Modechart Verifier in several novel ways. While the verifier can be used to prove the correctness of a specification with respect to a safety property, the MAC tool in conjunction with the Modechart Simulator allows the user to focus explicitly on a subset of the computations. By monitoring a specification simulation, the MAC tool can be used to trigger the verifier when certain conditions are observed in a computation.

Our implementation of the monitoring and assertion checking tool automatically translates assertions into Modechart fragments in order to provide a specification for the monitoring process. These monitoring fragments are symbolically executed together with the original specification to provide monitoring and assertion checking capabilities.

MT's specification language, Modechart, is a hierarchical, graphically-oriented specification language for real time systems and protocols [14]. A first-order language, Real Time Logic (RTL) [16], can be used to construct assertions which are checked against specification simulations generated from Modechart specifications. RTL allows the description of ordering and distance requirements between event occurrences. However, not all assertions expressible in RTL can be effectively monitored. Our tool addresses this challenge in several ways. First, it provides monitoring and assertion checking for simple RTL assertions by allowing the user to fill in parameters in a "forms-based" approach. These simple assertions are monitored by

translating the RTL assertions into Modechart monitoring fragments. Second, it provides monitoring and assertion checking for a wider class of RTL assertions based on comparisons between modes by composing the above-mentioned fragments into more elaborate Modechart monitoring fragments. And finally, the tool permits the user to incorporate his or her own monitoring fragments developed in Modechart into the original specification.

One model for formal specification and monitoring of constraints [13, 14, 15] in time-critical systems is based on timestamping events as they occur and maintaining an event history of such occurrences. The event history is then checked to determine whether a given assertion is satisfied by a computation. For a given assertion, the problem of monitoring and assertion checking can be understood in terms of three issues. The first of these is the determination of what information needs to be maintained by the monitoring and assertion checking system and for how long is it necessary to keep that information. In particular, as events are time-stamped and recorded for future evaluation, the event history can grow quite large. Establishment of a bound on the size of the event history by examination of the *syntax* of an assertion is crucial in order for monitoring and assertion checking to be effectively performed. The second difficulty of monitoring is resolution of the issue of when an assertion needs to be checked. Generally, an assertion does not need to be checked at every moment of a computation. Determining when the occurrence of an event (or its nonoccurrence) can make an assertion false is a critical element to efficient monitoring. Furthermore, it is desirable to ascertain the failure of an assertion as early as possible. This way it may be possible for the system to take some corrective action. Finally, an algorithm for checking the satisfiability of an assertion must be provided.

Our approach, that of using Modechart monitoring fragments to perform the monitoring and assertion checking during the symbolic execution of Modechart specifications, provides a tight integration of these three issues. For example, an explicit representation of the exact occurrence time of events may be unnecessary for many assertions, despite the fact that the RTL representation of these assertions contains explicit references to event occurrence times. Rather, it may be possible to monitor these assertions by recording the relationships of these event occurrences to each other and to the current time. In this way, the "what to keep" issue and "how to check" issue are more tightly integrated. Furthermore, by storing these relationships in a state machine written in Modechart, the "how to check" and "when to check" issues are bound together.

The remainder of this paper is organized as follows: Section 2 further describes the three main problems to be addressed in monitoring and assertion checking. Section 3 describes the integration of the monitoring and assertion checking techniques into the Modechart Toolset Simulation Tool. The Modechart language and the Monitoring and Assertion Checking Tool, MAC, are illustrated by discussion of a robot controller and manufacturing assembly line example in Section 3. In Section 4, we provide examples of assertions as well as summarize Real Time Logic, a first-order language for specifying assertions, and describe monitoring in the context of satisfiability of sentences in this language with regard to a particular computation. In Section 5, we introduce some primitive relationships between modes or intervals of time and describe a rich class of assertions which can be composed from these primitive relationships. Furthermore, we describe how these RTL assertions are translated into Modechart monitoring and assertion checking fragments. In Section 6 we discuss related work in monitoring and assertion checking and in Section 7 we describe the direction of our future efforts.

2 Three Problems of Monitoring

There are there major issues to be addressed in monitoring and assertion checking. The first of these is the determination of what event occurrences and other information needs to be stored and for how long that information is necessary. Secondly, it must be decided at what points the assertion must be checked. And finally, an algorithm must be defined for actually determining the satisfiability of the assertion for an execution prefix.

2.1 Event Histories

Since assertions are sentences which compare the time of the occurrences of various events in the system, it is generally the case that determining the satisfiability of an assertion involves recording certain event occurrences so that they can be used in comparisons with event occurrences that have not yet taken place. Consider the following Real Time Logic assertion, (eq. 1):

$$\forall i @(E,i) + 6 < @(E, i+1) \tag{1}$$

Informally, in this assertion, $@(E,i)$ represents the time of the ith occurrence of event `E`. Therefore this assertion states that at least 6 time units must elapse between each occurrence of event `E`. This type of assertion is known as a delay.

It is possible to monitor a delay by comparing the last two occurrences of the event. Although it is intuitively clear that this is a correct approach to monitoring a delay, the problem of syntactically determining from an assertion what event occurrences to record and maintain as well as when they can be discarded is the most significant problem of monitoring. Many assertions require the recording of more than just the last event occurrence.

An *event history* for a given event type is a set of the most recent occurrences of that event. The time and occurrence value for each occurrence is kept. It is desirable to be able to examine an assertion and to determine from its syntax a bound on the size of the event histories that must be maintained for each event. Furthermore, the bound should be independent of the particular computation over which the assertion is to be monitored.

This is important for several reasons. Not only would a lack of such a bound be impractical from a storage perspective, but it would imply that it would be impossible to bound the execution time of checking the satisfiability of an assertion at a given point in time. Put another way, if the event history were unbounded, that would mean that a given moment in time a potentially unbounded number of comparisons would be necessary to affirm or deny the assertion. This is potentially significant for run-time monitoring in a real time environment. The existence of a bound on the size of the event history forces a constant bound on the amount of work the monitoring process must do at each moment in time.

Such a bound does not exist for all assertions. For some assertions, additional assumptions about the possible execution traces must be made in order to restrict the event-history. For example, consider the assertion:

$$\forall i @(WRITE_LOC_1, i) < @(READ_LOC_1, i) \vee @(WRITE_LOC_2, i) < @(READ_LOC_2, i) \tag{2}$$

This assertion states that for each i, the ith read of location 1 must follow the ith write of that location or that the ith read of location 2 must follow the ith write of location 2. The event history for the monitoring of this assertions is potentially unbounded. Consider a point in a computation where for some i, the first comparison fails and neither $\mathtt{WRITE_LOC}_2$ nor $\mathtt{READ_LOC}_2$ has occurred for that i. In this case, the satisfaction of the assertion for this i cannot be determined until either $\mathtt{WRITE_LOC}_2$ or $\mathtt{READ_LOC}_2$ has occurred and the event history for $\mathtt{WRITE_LOC}_1$ and $\mathtt{READ_LOC}_1$ will need to be maintained until the entire assertion can be evaluated. However, during this period of time, arbitrarily many more $\mathtt{WRITE_LOC}_1$ and $\mathtt{READ_LOC}_1$ events could occur. Each of these could need to be saved until the corresponding events took place on location 2. Therefore, the event history is potentially unbounded.

2.2 When to Check

The second major issue in monitoring is that of when to check each assertion. The requirement that an assertion hold at every moment in time does not require that the assertion be checked at each moment in time. This is of great value because performing such checking at each moment of the computation would be prohibitively time consuming. Often it is the case that if, at a particular moment in time, no event

mentioned in the assertion occurs, then it is not necessary to check that assertion. Again, consider the example in Equation 1. The only events relevant to the satisfaction of this assertion are the is the event mentioned in the assertion. If at a given moment in time, this event doesn't occur, then nothing can happen to make that assertion become false.

This suggests that monitoring can be performed without checking the satisfiability of the assertion at every moment in time. Two approaches can be described. The *lazy* method involves checking the assertion only at those moments in time where an event mentioned in the assertion actually occurs.

Although many assertions can be monitored by such an approach, for some assertions, the lazy approach will permit undetected violations. Equation 3 contains an example of a deadline:

$$\forall i @(E, i+1) < @(E, i) + 100 \tag{3}$$

Informally, this assertion states that at most 100 time units must elapse between each occurrence of event `E`. Although the lazy approach is effective for the delay specified in Example 1, if a deadline expires and the monitored event never occurs, the deadline violation will remain undetected. Therefore, for some assertions, a more sophisticated approach than merely examining event occurrences is necessary.

Alternatively, an *active* approach is possible. This mechanism is based on the understanding that it is sometimes possible to detect the eventuality of a violation in advance of the occurrence of the events mentioned in the assertion.

For example, the violation of a *deadline* can be detected after the deadline expires; it is not necessary to wait until the late event occurs to raise an error. For assertions such as deadlines, *timers* are used to trigger the monitoring system to check the assertion.

In fact, sometimes it is possible to detect the eventual violation of an assertion *before* the deadline expires. Generally this is due to sophisticated assertions which contain *implicit deadlines* which are not directly stated.

2.3 How to Check

The third problem of monitoring is the actual determination of the satisfiability of an assertion when instantiated with actual event occurrence times. In general, determination of the satisfiability of an assertion reduces to the problem of testing the satisfaction of the assertion for particular event occurrences which have been recorded on the event history. In Equation 1, the assertion states that the inequality must hold for *all* pairs of subsequent event occurrences. That means, simply, that *each* pair of event occurrence times must be replaced into the inequality and the inequality test to see if it is satisfied for those particular occurrence times.

It is intuitively obvious how to do this for the delay example. However, a general mechanism that will handle more complex assertions must be used. In addition, an active monitoring approach (as described above in Section 2.2) may require testing the satisfiability of an assertion before all the mentioned events have occurred. This means that it must be possible to ascertain the satisfiability of inequalities containing one or more variable terms.

For such assertions, an elegant mechanism [4, 12] for checking satisfiability maps the inequality into a graph. Each event type is represented by a node and timing constraints between event types are represented by weighted edges between nodes. The graph is then checked for negative weight cycles using the Floyd-Warshall algorithm [1].

Alternatively, when a bound has ben established for the event history, it is also possible to define Modechart monitoring fragments or other automata to perform the actual checking. Instead of instantiating the RTL formula directly with times of event occurrences, event occurrences and timers trigger state changes in the Modechart monitoring fragments. This approach we have adopted for our tool and is described in more detail in Sections 3 and 5.

3 Integration of Monitoring with the Modechart Toolset

The Modechart Toolset (MT), developed by the Naval Research Laboratory provides a graphical user interface for entering Modechart specifications [5, 23]. In addition, the toolset provides formal verification, static completeness and consistency checking, and specification simulation. This section describes the Monitoring and Assertion-Checking Tool (MAC) and its integration with MT.

The Modechart Simulator provides symbolic execution of Modechart specifications. These symbolic executions produce execution traces which are examined by the MAC tool. The MAC tool is integrated with the Simulator and checks execution traces as they occur to ascertain the satisfiability of the assertions.

In Section 3.1, we illustrate the Modechart language using a robot controller for a manufacturing assembly line. This example is used for purposes of illustration throughout the paper. In Section 3.2, we describe our approach to monitoring and assertion-checking and the integration of the MAC Tool into the Modechart Toolset. Finally, in Section 3.3, we return to our robot example to demonstrate our approach.

3.1 Overview of the Modechart Language: A Robot Controller Example

Modechart [14] is a graphical specification language based on concurrent finite state diagrams. It provides a compact and structured way to represent real time systems. Although similar to Harel's Statecharts [10], Modechart is specifically designed for the specification of real time systems. It allows for the specification of modes which represent control information which impose structure on the operation of a system.

Modechart is extended from Statecharts with constructs for expressing timing constraints. It has a visual hierarchical structure and a small set of well defined constructs for the definition of event-driven real time systems. These constructs include *modes, mode-transitions, events*, and *timing constraints*. During a non-zero moment in time, a mode can be either active or inactive. Informally, the state of a real time system is described by the collection of modes which are active.

Modes are hierarchically arranged in a tree structure; there is a top level mode from which all modes are descended and each mode has only one parent. The children of each mode can execute serially or in parallel, allowing Modechart to capture both concurrent and sequential behavior. The hierarchical nature of Modechart also allows specifications to be evaluated at different levels of abstraction.

The behavior of a real time system is captured by *mode transitions*. Mode transitions are expressions which control the exit from one mode and entry into another. These transitions can be specified by timing constraints or can be triggered by events in the system or by predicates on the behavior of modes in the specification. Events in Modechart include *mode-entry*, *mode-exit* events, transition events, and external events. When a mode becomes active it is said that a mode entry event corresponding to that mode occurs. When a mode becomes inactive it is said that a mode exit event occurs. Mode transition events occur where one mode is exited and another is entered. Finally, external events represent something happening in the environment which can affect the behavior of the system. If $\mathtt{M}_1$, and $\mathtt{M}_2$ are modes, the exit of $\mathtt{M}_1$ is indicated by $\mathtt{M}_1\!\rightarrow$, the entry of $\mathtt{M}_2$ is indicated by $\rightarrow\!\mathtt{M}_2$, and the transition from $\mathtt{M}_1$ to $\mathtt{M}_2$ by $\mathtt{M}_1\!\rightarrow\!\mathtt{M}_2$. External events are represented by single letters, e.g. `E`. These form the atomic units of a system; assertions are described in terms of timing and ordering relationships between instances of these type of events.

Transitions can also be controlled by predicates on modes. For example, the predicate $<\mathtt{M})$ indicates that the mode `M` has been active for at least one unit of time. More precisely, `M` was entered before the current moment of time, and since then, `M` has not exited before the current moment. Further discussion of the possible predicates on modes can be found in [23, 14]. Finally, more complex mode transition expressions can be formed from triggers and timing constraints. More elaborate triggers can be composed by taking the conjuncts of trigger expressions, and these conjuncts can be disjuncted together with timing expressions.

Consider the robot controller for a manufacturing assembly line illustrated in Figure 1. A producer process generates items to be processed by a robot. The producer process places the items on a conveyer belt and advances the conveyer belt so that the item approaches the robot. When the item is in the proper position, the robot picks up the item, rotates away from the belt and processes the item in some way; for example, it paints the item. When the robot is done processing the item, it attempts to place the item on

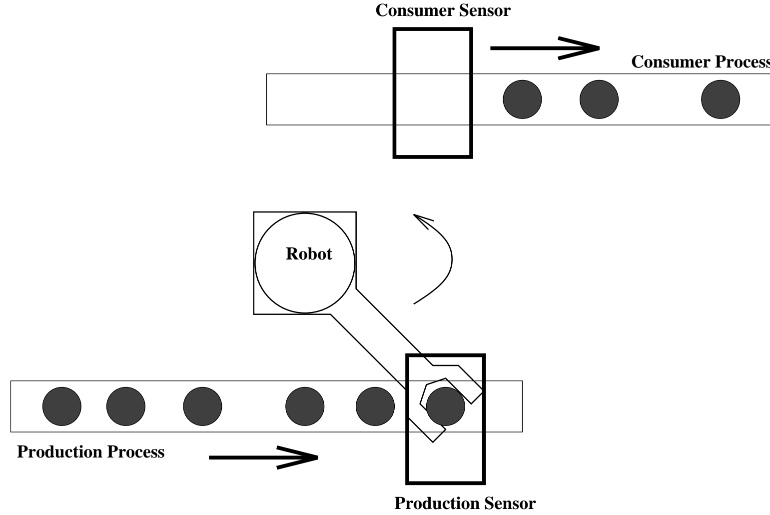


Figure 1: Diagram of a Robot Controller for a Manufacturing Assembly Line

another conveyer belt where it is removed by a consumer process.

The specification models two main components, the system and the environment. The system includes the robot controller and the conveyer belt controller for the producer process. This system has no control over the conveyer belt for the consumer process; that is considered part of the environment. The environment includes the robot, the producer conveyer belt, and the consumer conveyer belt, together with various sensors. The sensors in the environment send signals which are received by the controllers in the system and the system sends control signals to the environment.

Figure 2 contains a Modechart specification for this example. At the top level, we introduce two parallel modes which model the environment and the system. The `System` mode is a parallel mode; its two child modes `S.Producer_Belt` and `S.Robot_Controller` are active simultaneously. Similarly the `Environment` mode has seven children which are active simultaneously, with each of the child modes representing some aspect of the environment in which the controller must function.

The `S.Robot_Controller` mode is responsible for sending signals to the environment to instruct the robot to take an item from the processor belt, process it, and place it on the consumer belt. The signals sent to and received from the environment are represented by various mode entry events as described in Table 1. The mode `S.Robot_Controller` also responds to various events in the environment indicating that the robot has performed the last command and is ready for the next one. The `S.Robot_Controller` mode is a serial mode. When it becomes active, only one of its child modes become active.

The serial children of the `S.Robot_Controller` mode become active in turn by transitions. The box with the thicker outline represents that the `S.RC.Wait_to_Grab` mode is the *initial* mode for the `S.Robot-_Controller` mode. That is to say, upon entering the `S.Robot_Controller` mode, Modechart also enters the `S.RC.Wait_to_Grab` mode.

Several kinds of mode transition expressions can be observed in this specification. The transition between the `S.RC.Wait_to_Grab` mode and the `S.RC.Extend` mode is controlled by the expression `(E.P.Ready)`. This expression indicates that the transition from `S.RC.Wait_to_Grab` into `S.RC.Extend` is to be taken if the mode `E.P.Ready` is active. The transition from `S.RC.Extend` to `S.RC.Grab` is controlled by the expression →`E.AE.Extended`. That is to say the transition is taken when the mode `E.AE.Extended` is entered. Finally, the transition from `E.AE.Extending` to `E.AE.Extended` is controlled by a timing condition, `(1,5)`. This indicates that the transition occurs at least one unit of time and not more than five units of time after the mode `E.AE.Extending` becomes active. A summary of the timing constraints is depicted in Table 2.

This system concisely demonstrates many of the usual properties of real time systems. Many of the

SYSTEM (Parallel)

S.PRODUCER_BELT (Serial)

S.PBC.STOP

->E.P.NOT_READY

->E.P.READY

S.PBC.START

S.ROBOT_CONTROLLER (Serial)

S.RC.PROCESS

E.RP.PROCESSING->E.RP.NOT_PROCESSING

S.RC.ROTATE

->E.AR.90_DEG

S.RC.GRAB

->E.HP.HOLDING

->E.AR.180DEG

S.RC.WAIT_TO_DROP

S.RC.ROT.ROT1

S.RC.ROT.ROT2

->E.AE.EXTENDED

{E.C.READY}

S.RC.EXTEND

S.RC.DROP

{(E.P.READY)}

->E.HP.EMPTY

->E.AE.0DEG

->E.AE.RETRACTED

S.RC.WAIT_TO_GRAB

S.RC.ROT.ROT3

S.RC.RETRACT

ENVIRONMENT (Parallel)

E.ARM_ROTATION (Serial)

E.AR.0DEG

E.AR.1_89_DEG

->S.RC.ROTATE

(4,6)

(2,3)

E.AR.90.DEG

E.AR.181_359DEG

->S.RC.ROTATE

->S.RC.ROTATE

(2,3)

E.AR.180DEG

E.AR.91_179DEG

E.ARM_EXTENSION (Serial)

E.AE.RETRACTED

E.AE.EXTENDING

->S.RC.EXTEND

(3,5)

(3,5)

->S.RC.RETRACT

E.AE.RETRACTING

E.AE.EXTENDED

E.HAND_POSN (Serial)

E.HP.EMPTY

E.HP.GRABBING

->S.RC.GRAB

(1,5)

(1,5)

E.HP.DROPPING

E.HP.HOLDING

->S.RC.DROP

E.ROBOT_PROCESS

E.RP.NOT_PROCESSING

(5,10)

->S.RC.PROCESS

E.RP.PROCESSING

E.P_READY (Serial)

E.P.NOT_READY

(1,15)

E.P.READY

({E.AR.90DEG}&{E.AE.EXTENDED}
&{E.HP.HOLDING})

E.PRODUCER_BELT (Serial)

E.PB.BELT_STOPPED

->S.PBC.START

->S.PBC.STOP

E.PB.BELT_MOVING

E.C_READY (Serial)

E.C.NOT_READY.

E.C.READY

({E.HP.EMPTY}&{E.AR.180DEG}
&{E.AE.EXTENDED}

{E.CB.BELT_MOVING}

(3,3)

E.C.MOVING_AWAY

E.CONSUMER_BELT (Serial)

E.CB.BELT_STOPPED

(1,100)

(3,100)

E.CB.BELT_MOVING

Figure 2: Modechart Specification of a Robot Controller for a Manufacturing Assembly Line

System ⟶ Robot	**Modechart Representation**
rotate robot arm	→ `S.RC.Rotate`
extend robot arm	→ `S.RC.Extend`
retract robot arm	→ `S.RC.Retract`
grab with robot hand	→ `S.RC.Grab`
drop with robot hand	→ `S.RC.Drop`
process with robot hand	→ `S.RC.Process`
System ⟶ Producer Conveyer Belt	**Modechart Representation**
start producer belt	→ `S.PBC.Start`
stop producer belt	→ `S.PBC.Stop`
Environment ⟶ System	**Modechart Representation**
robot arm is at 0 degrees	→ `E.AR.0Deg`
robot arm is at 90 degrees	→ `E.AR.90Deg`
robot arm is at 180 degrees	→ `E.AR.180Deg`
robot hand is holding item	→ `E.HP.Holding`
robot hand is empty	→ `E.HP.Empty`
robot hand is processing item	→ `E.HP.Processing`
robot hand is done processing item	`E.HP.Processing` → `E.HP.Holding`
robot arm is extended	→ `E.AE.Extended`
robot arm is retracted	→ `E.AE.Retracted`
producer item is ready	→ `E.P.Ready`
producer item is not ready	→ `E.P.Not_Ready`
consumer process is ready for item	→ `E.C.Ready`
consumer process is not ready for item	→ `E.C.Not_Ready`

Table 1: Signals and Corresponding Modechart Events

Time	Minimum	Maximum
Robot arm rotation time (90 deg)	2	3
Robot arm rotation time (180 deg)	4	6
Robot arm extension time	3	5
Robot arm retraction time	3	5
Robot hand grabbing time	1	5
Robot hand dropping time	1	5
Robot hand processing time	5	10
Producer item ready time	1	15
Time to move item away on consumer belt	3	3
Consumer belt moving time	3	100
Consumer belt stopped time	1	100

Table 2: Timing Constraints for Robot

typical safety (including real time) properties can be tested for such a system. For example, one may want to know "Does the robot ever enter the mode `E.HP.Processing` when there is no item in position?", "Does an item ever move from the Producer to the Consumer without being processed?", or "Does an item ever wait in position more than 20 time units before being processed?"

3.2 Approach to Monitoring and Assertion-Checking Monitoring in the Modechart Toolset

In this section, we discuss the implementation of the Monitoring and Assertion Checking tool (MAC). The MAC tool complements the existing tools in MT, allowing testing to take place for program specifications and assertions for which formal verification is impractical. This tool provides valuable feedback to system designers during the initial design process. The overall architecture of the Modechart Toolset is displayed in Figure 3.

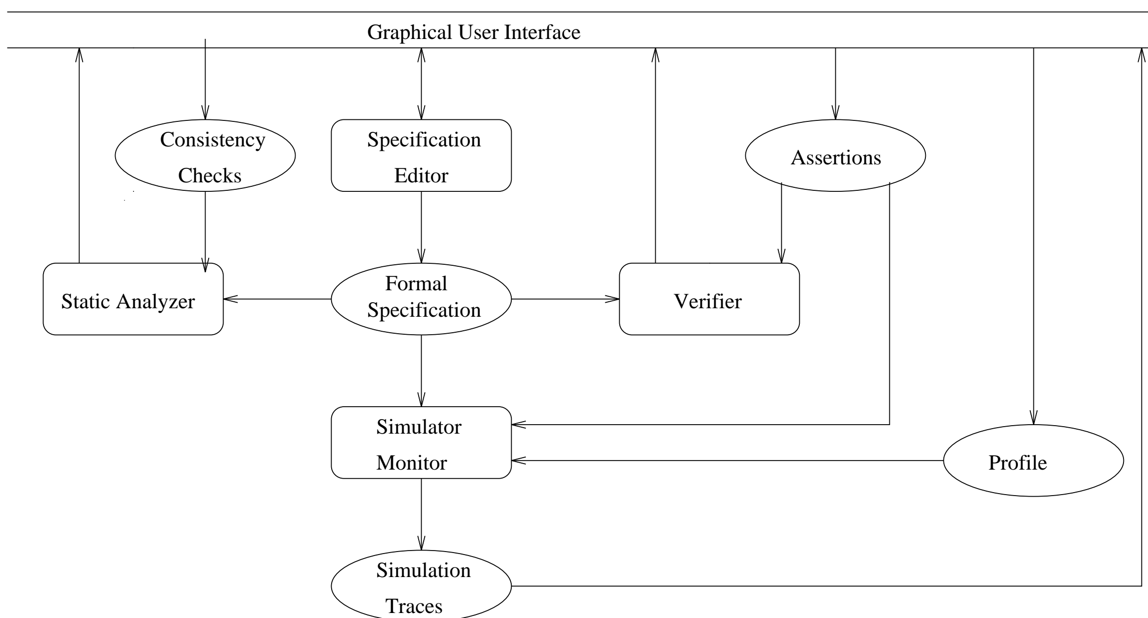


Figure 3: The Architecture of the Modechart Toolset

Users of the Modechart Toolset enter specifications via the *specification editor*. The specification editor has a graphical interface via which the user draws the modes and transitions which comprise the specification. Generally, once a user has entered a specification she or he will want to perform certain application-independent consistency checks. The *static analyzer* checks the specification to guarantee that the specification is well-formed and fully specified. It also checks for various conditions often associated with errors in specifications such as *sink modes*.

After these static checks have been performed, analysis of the more dynamic properties of the system may be desirable. The two tools for checking dynamic properties of the specification are the Modechart Verifier and the Modechart Simulator. The verification tool provides a model checking approach for verifying properties of specifications by examination of a computation graph using a model-checking approach. The verified properties are stated in Real Time Logic (RTL). The verifier is currently implemented for specific classes of RTL assertions that are likely to be of interest to system developers. These classes of assertions are also handled by the MAC tool.

The verifier is complemented by the Simulator Tool. The simulator generates a symbolic execution of the Modechart specification. One use of the simulator is to ascertain whether a Modechart specification

satisfies the developer's intent. The tool user can generate an execution trace and examine it for errors or variations from informal specifications of the system. That is, simulation of execution traces is often very useful during the design phase of system development because it allows the developer to informally evaluate system behavior as well as to test for specification violations for particular execution traces.

The MAC Tool is a component of the Modechart Simulator. The key idea is to augment a Modechart specification with a *monitoring fragment* which represents the assertion to be checked. The monitoring fragment, also expressed as a Modechart specification, is used to represent the satisfaction or violation of an assertion. The goal is to use the Modechart simulator to produce a possible execution trace of the augmented specification. The monitoring fragment is symbolically executed together with the original specification generating an execution trace that highlights the violation of the original assertion. What is attractive about this approach is that instead of generating an operational description (or algorithm), an indicator of the status of the assertion is provided.

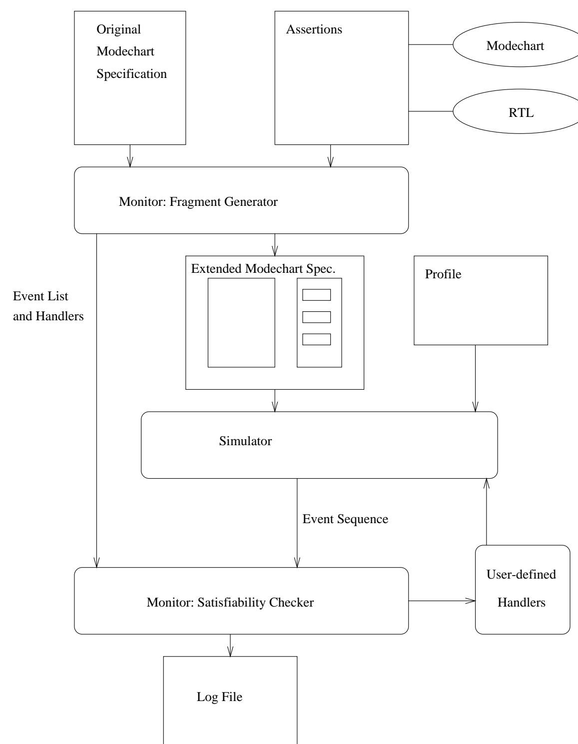


Figure 4: Implementation Design

Figure 4 illustrates the MAC Tool and its relationship to the Modechart Toolset. As described later in this section, there are several ways to specify an assertion. The *Fragment Generator* produces a Modechart specification designed to perform monitoring during simulation. This supplemental specification (or fragment) is appended to the original specification. The MT *simulator* produces possible execution traces from a *Modechart specification* and a user-defined *profile*. The monitoring specification is symbolically executed along with the original specification. The execution traces generated by the MT simulator are examined by the *Satisfiability Checker* to test for violations of system constraints. Satisfiability checking is simplified to checking for particular events generated by the newly created monitoring fragment. The tool also supports *user-defined handlers* that are invoked by the monitor when a particular property is satisfied or a violation is detected.

There are several ways in which assertions can be specified by the user:

(a) The user may provide his or her own monitoring fragment as a Modechart specification. This allows

the user the maximum flexibility in terms of what kind of properties can be monitored. Modechart monitoring fragments can be developed in the specification editor in much the same manner as are ordinary system specifications. The only real difference is that the user must identify to the MAC Tool which mode represents a violation of the assertion and what handler should be invoked upon entry to the violating mode.

(b) Classes of assertions which are supported by the MT verifier are also supported by the MAC tool. These assertions include:

- Separation
- Mutual Exclusion
- Cover Mode
- Under Mode
- Inner Universal
- Outer Universal
- Reachability
- Elapsed Time

These assertions are specified in Real Time Logic via a graphical, forms-based interface. (One such a form is displayed in Figure 5.) The user fills the relevant information into the RTL formula, but is not required to write RTL formulae from scratch. Each of these assertions corresponds to a Modechart monitoring fragment. The necessary information is filled in the fragment and the resulting fragment is attached to the original Modechart specification.

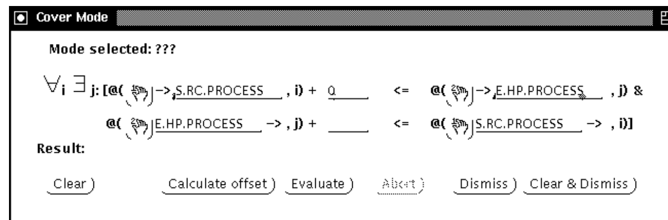


Figure 5: Specifying an Assertion via the Forms Interface

(c) The user can specify more complex assertions, composing primitive relationships described above in part (b). Again the user can fill mode names into a form to specify the primitive assertions; these are then very easily composed using the usual logical connectives. In this case, the MAC Fragment Generator composes monitoring fragments written in Modechart to develop a complex monitoring fragment tailored to the user's request. The resulting fragment is appended to the original specification and simulated together with it.

3.3 A Monitoring Example: The Robot Controller Revisited

In Figure 6, the robot example described in Figure 2 has been supplemented with a monitoring fragment. In this example, the assertion to be monitored is designed to test whether the mode `S.RP.Process` is ever active for fewer than 6 time units. Such delay assertions are further discussed in Section 4.3.

The mode `Monitor` contains the monitoring fragment. This fragment could be provided by the user or it could be generated by the tool to provide monitoring for an assertion specified in RTL. As long as the mode

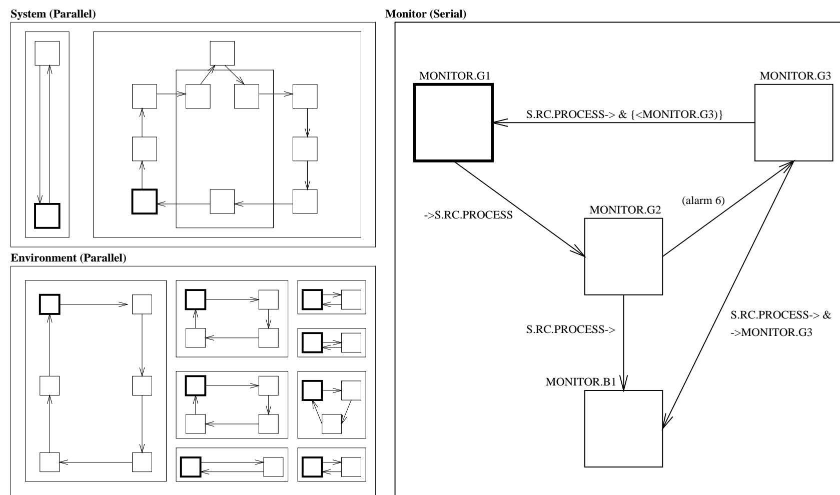


Figure 6: A Modechart Specification of a Assembly-Line and Robot Arm Controller with Monitoring Component

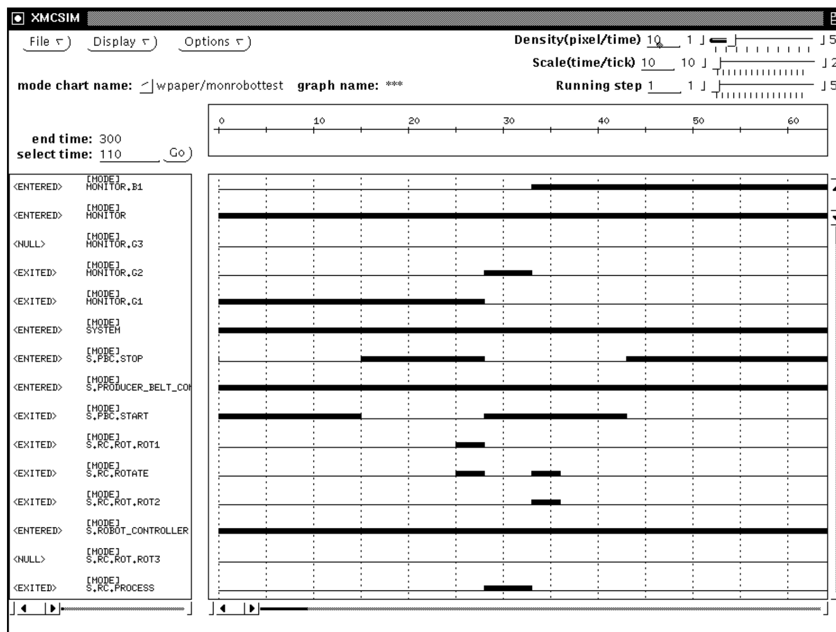


Figure 7: Simulation of Robot with Monitoring Component

`S.RP.Process` stays active for at least 6 units of time, the system stays in modes `Monitor.G1`, `Monitor.G2`, or `Monitor.G3`. However if the mode exits earlier, the system will enter mode `Monitor.B1`.

It may be noted that there are many possible Modechart fragments which could be used to monitor this assertion. The particular fragment shown in Figure 6 is designed for simplicity. Later, we demonstrate another approach to monitoring a delay which can be composed with other monitors to monitor more complex assertions. This is described in Section 5.

Figure 7 contains a simulation of the robot together with the monitoring fragment as generated and displayed by the Modechart Simulator. Mode names are listed on the vertical axis on the left and the elapsed time is displayed across the top in this "time-process diagram" [21]. A thick horizontal line indicates the period of time during which each mode is active. The particular execution displayed violates the delay assertion. It can be seen from this diagram that at time point 27, S.RC.Process becomes active and exits at time point 32. Moreover, at time point 32, the mode `Monitor.B1` becomes active, while the mode `Monitor.G2` becomes inactive.

4 Specification of Assertions in RTL

In the previous section, we described how assertions may be specified by the user in either Real Time Logic (RTL) or in Modechart. Users may prefer specifying a property of interest in a more concise notation using an assertional language like RTL over a language based on state transition diagrams. Moreover, it is desirable for the MAC tool to handle many of the same classes of RTL assertions which are supported in the verifier tool.

In this section we provide an informal overview of Real Time Logic. In order to define the idea of the satisfiability or violation of an assertion, we introduce the concept of a computation in terms of event occurrences. This leads to the definition of a *computation prefix* which is used to provide interpretations for RTL. These interpretations are used to determine whether or not an assertion is violated during simulation of a specification. Finally, we illustrate several examples of assertions specified in RTL and their corresponding Modechart monitoring fragments. These examples correspond to the primitive relationships from which more complex assertions are composed.

4.1 RTL Overview

Rather than creating Modechart monitoring fragments directly, the user may want to specify assertions for which the monitoring tool can generate the relevant monitoring fragments. The general specification language for assertions is Real Time Logic (RTL). RTL is currently used by the toolset users to specify certain classes constraints which are verified to hold for all possible executions. Moreover, RTL appears to facilitate the specification of assertions as constraints which must always hold over the computation.

Real Time Logic is a first order language designed to describe order relationships and timing constraints about event occurrences in real time computations.

The language elements of RTL include event constants, positive and negative integer constants, a binary function, $+$, a binary relation, $<$, the logical symbol, $=$, and a binary function, @.

In this model, a real time system is a collection of event types. It is assumed that the collection of event types is finite with each event type having a unique name. Each event can have multiple occurrences. The event occurrences for each event type are ordered. Each occurrence of a particular event is described by an *occurrence index* which indicates its ordinarily, i.e. whether it is the first, second, third, etc. occurrence of that event type. Furthermore, connected with each event occurrence is the time at which that event occurred. In addition, it is assumed that an event cannot occur more than once at a particular moment in time. (Time is modeled by a strictly increasing monotonic sequence with a starting point, such as the natural numbers.) An event occurrence can be represented by a triplet (E, i, t) indicating the name of the event, its occurrence index, and the time when that occurrence took place.

4.2 Computation Prefixes and Interpretations for RTL

Here we introduce the concepts of a computation and a computation prefix. The computation prefixes provide an interpretation to RTL by assigning values to the occurrence function. Monitoring of assertions is defined in terms of checking the satisfiability of an RTL sentence by the sequence of computation prefixes which compose a computation. Because each computation prefix is consistent with the previous computation, only the impact of the most recent events on the satisfiability of the assertion needs to be considered.

A real time computation is defined to be a generally infinite sequence of sets of occurrences of events:

$$\Sigma = \Sigma_1, \Sigma_2, \Sigma_3, \ldots$$

where each Σ_t represents the finite set of event occurrences which took place at time $t = i$. Each $\sigma_j \in \Sigma_t$ is an event occurrence represented by some (E, i, t) subject to the constraint that for a fixed event type, E, the occurrence index i's strictly increase as the time values, t's strictly increase. Moreover, the same event may not occur more than once at each moment in time, that is, for a given t, if $(E, i, t) \in \Sigma_t$, then the i is unique for that t. Notice that no ordering of event occurrences can be determined for events which took place at the same moment in time; we model simultaneous events with true concurrency rather than by using an interleaved model.

A prefix of a computation is an initial subsequence of sets of event occurrences such that all the events occurrences in the prefix occurred at or before some time, t, indicated by P_t. Each prefix of a computation is a subset of any prefix that follows it and of the entire computation. Therefore, a computation can be seen to be the union of a chain of computation prefixes defined by each moment in time. Each prefix of a computation will partially define an interpretation for RTL. Each such interpretation will assign to the universe the positive and negative natural numbers (to which the corresponding constants are assigned), and additional elements for each event type that can occur in the system.

The function, $+$, the binary relation, $<$, and the logical symbol, $=$ are interpreted in their usual ways on the integers for each interpretation defined by a prefix. The function symbol @ is defined on

$$E \times \mathcal{Z}^{+} \rightarrow \mathcal{N}$$

and is interpreted by each so that $@(E, i)$ indicates the time of the ith occurrence of event E, for $i > 0$. Therefore, any prefix, P_t, has an interpretation of the @ function is consistent with every prefix which ended before P. The reduct of @ to P_t is exactly the reduct of @ to $P_{t-1} \cup \{@(E, i) : @(E, i) = t$ for some event $\mathtt{E}\}$. That is, each prefix P_t interprets the @ function exactly as its predecessor on event occurrences that happen before time t and newly interprets @ on event occurrences that happen at time t.

For $@(E, i)$ where the ith occurrence of event E has not happened at time t, it is unclear how the occurrence function should be interpreted by P_t. Here we consider that there are very many, probably infinitely many, possible *extensions* of the computation prefix to complete computations. In each of these possible extensions, the ith occurrence of E may take place at different times. All that is known at time t is that the ith occurrence happens in the future. Therefore, any natural number greater than time t can be assigned to $@(E, i)$.

In summary,

$$@(E, i)_{P_t} = \begin{cases} t & \text{if the i'th occurrence of } \mathtt{E} \text{ happens at time t} \\ @(E, i)_{P_{t-1}} & \text{if the i'th occurrence of } \mathtt{E} \text{ happens before time t} \\ t' \in \mathcal{N} - \{0, 1, \ldots t\} & \text{otherwise} \end{cases}$$

To *monitor* an assertion is to check its satisfiability over a computation. In order to do this, monitoring is performed on each prefix of the computation. The computation is the union of all of its prefixes so the ascertaining the satisfiability of an assertion over a computation can be achieved by ascertaining the satisfiability of that assertion for each prefix of the computation.

To check the satisfiability of an assertion on a computation prefix, we check whether the assertion is satisfiable on some extension of that prefix into a complete computation. Because the interpretation provided by the computation prefix P_t is consistent with interpretations of prefixes preceding P_t, the satisfiability of an assertion by an interpretation P_t involves checking only the satisfiability of the assertion with regard to event occurrences which happen at time t and following.

Assertions are limited to sentences, that is, formulas with no free variables.

The following is an example of a common type of assertion, a delay:

$$\forall i \, \{ @(\rightarrow \text{Process}, i) + 3 < @(\rightarrow \text{Process}, i+1) \} \quad (4)$$

This assertion states that each occurrence of event E must be separated from the subsequent occurrence by at least 6 time units. Note that if E first occurs at time $t = 2$ and occurs the second time at $t = 10$, the assertion will be satisfiable under the interpretation provided by the prefix P_{10}. In addition, subsequent interpretations will not need to re-check the assertion for $i = 1$ since all subsequent interpretations are supersets of P_{10} and consistent with P_{10}.

4.3 Examples of Assertions

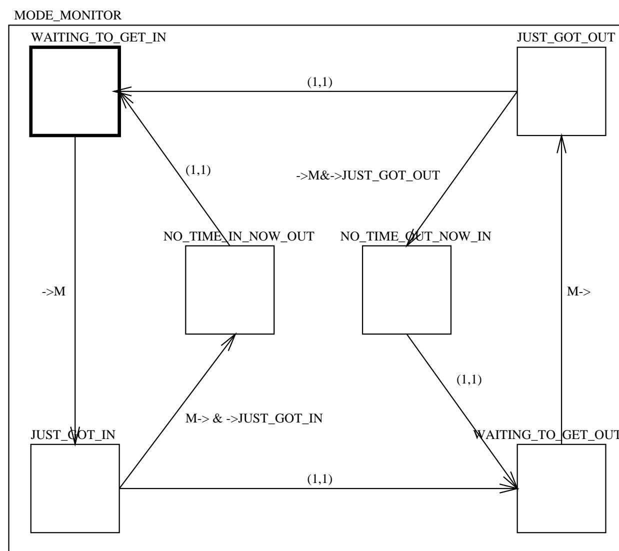


Figure 8: Modechart Monitoring Fragment for a Single Mode

In this section, a variety of example assertions are presented in order to further illustrate the problems in bounding the event history and in deciding when in a computation it is necessary to check the satisfiability of an assertion. These example assertions correspond to some of the primitive monitoring fragments described above. They also demonstrate how monitoring is simpler for assertions describing the behavior of modes than for assertions describing the behavior of general Modechart events. In this section we use capital letters to indicate general Modechart events. These events could be mode-entry, mode-exit, or external events.

In general, we will consider assertions concerned with the behavior of modes. We describe timing constraints on individual modes, as well as restrictions on the behavior of pairs of modes. Therefore, it is necessary to address carefully the behavior of a single mode. As described above, a given mode may be *active* or *inactive*. A mode is active from the time that its mode entry event occurs until the time its mode exit event occurs (and, respectively, inactive from the time that its mode exit event occurs until its mode entry event occurs). However, since the semantics of Modechart allow simultaneous events, it is possible for the

mode entry event and mode exit event to occur at the same moment of time. When these two events occur simultaneously it can be difficult to ascertain whether the mode is active or inactive at the next moment in time.

For this reason, we introduce a monitoring fragment (see Figure 8) which is useful in indicating the precise behavior of a mode–whether it is active or or in active and whether it exited and entered at the same moment of time. In this example, the modes `Just_Got_In`, `No_Time_In_Now_Out`, and `Waiting_To_Get_Out` indicate that mode M is active. The remaining three modes indicate that mode M is inactive. This monitoring fragment used to specify the more complex monitoring fragments described in the remainder of this section.

Delay Assertion.

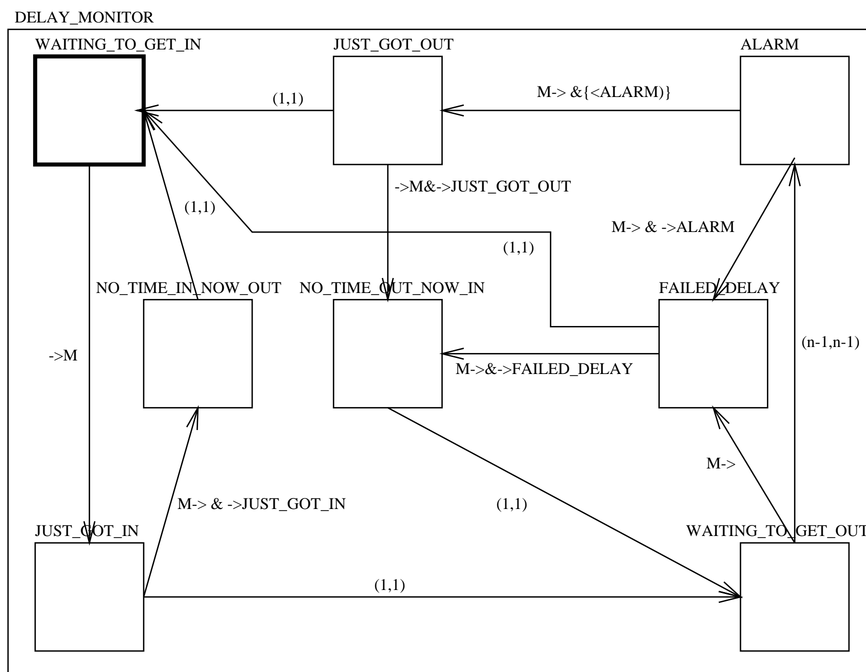


Figure 9: Modechart Monitoring Fragment for a Simple Delay

Consider the following assertion:

$$\forall i @(E, i) + 6 < @(E, i+1) \tag{5}$$

This assertion, a *delay*, states that every occurrence of the event E must be separated by at least 6 time units. As described above, it is sufficient to maintain the time of the last occurrence of the event E in order to monitor this assertion. Each time an E event occurs, the time of that event is compared to the time of the last event stored in the event history for this assertion. The satisfiability is checked by instantiating the formula with the actual times of the two occurrences. Finally, the previous event occurrence is discarded from the event history and replaced by the record of the time of the most recent event occurrence. Here it is easy to see that the event history does not need to maintain more than one "old" occurrence of any event. It is sufficient to compare the time of any event occurrence with the time of the last occurrence of that event type.

There is, however, another approach to monitoring this assertion. For a given occurrence of event E, it possible to determine the truth of the assertion after 6 time units have passed, whether or not the next occurrence has taken place. If it has taken place, then, in fact, the assertion has been violated. If, however, it has not taken place, then it can be seen that if it ever happens, then it will happen at least six moments after the last occurrence, because if it hasn't occurred yet, it can only occur in the future.

A Modechart fragment to monitor a similar assertion, one involving the duration of a mode, using this approach is depicted in Figure 9. This assertion states that the mode is active for at least n time units. The corresponding RTL assertion is:

$$\forall i @(\rightarrow M, i) + n < @(M \rightarrow, i) \tag{6}$$

This Modechart monitor extends the technique depicted in Figure 8 by adding an alarm mode and a mode (`Failed_Delay`) to indicate that the delay has been violated. While the previous monitor indicated whether a mode is active or not, this monitor indicates whether a mode is active long enough to satisfy the delay. (In the figure, n indicates the duration of the delay.) Also, note that entering the mode `No_Time_In_Now_Out` also indicates a violation of the delay.

Another type delay is shown below:

$$\forall i @(E_1, i) + 4 < @(E_2, i) \tag{7}$$

Monitoring of this delay is more complicated than monitoring of the apparently similar delay involving modes described in Equation 6. Unlike mode entry and exit events, the events E_1 and E_2 are unrelated. As a consequence, it is insufficient to keep each occurrence of E_1 and wait for the corresponding E_2 event to happen before checking. It is impossible to determine without additional assumptions about the execution trace or the system that there is any bound of the number of E_1 events that can occur (and would need to be recorded) before the corresponding E_2 event. (In contrast, we know that a mode exit event will occur before a subsequent mode entry event. This is described in more detail in Section 5.) For Equation 7, the *only* way to make a bound on the event history for E_1 is to record each E_1 event, wait four time units and determine whether the assertion is satisfied. After four time units the satisfiability of the assertion for that particular i is known and the record of that particular E_1 event occurrence can be discarded. As a consequence, the bound on the event history is four since no occurrence of E_1 needs to be kept longer than four moments. This example demonstrates that the establishment of a bound on the size of the event history is sometimes closely tied to the determination of a bound on the *duration* for which an event occurrence must be maintained.

$$\forall i : @(E, i) + 100 < @(E, i + 3) \tag{8}$$

Equation 8 demonstrates yet another type of delay. In this case there is an added complication in that the delay is enforced not for the subsequent event occurrence but for the 3th subsequent event occurrence. In this case, it is necessary to keep each occurrence in the event history for 100 time units or until there are 3 of them. That is, the maximum size of the event history is 3 because there can be up to 3 i's for which the satisfaction of the delay is unknown. For each event occurrence, if 100 moments pass before the 3th subsequent occurrence takes place, the assertion is satisfied. Alternatively, if the 3th subsequent event occurrence happens before 100 time units have elapsed, the assertion will have been violated for that i.

A Modechart monitoring fragment for such a delay would require separate fragments to monitor each of the three occurrences of event `E` for which the satisfaction of the delay would be unknown at a given moment in time.

Deadline Assertion. Consider the following assertion:

$$\forall i : @(\rightarrow M, i) + n > @(M \rightarrow, i) \tag{9}$$

This type of assertion, a *deadline* is generally monitorable in much the same fashion as are delays. This assertion states that the mode M must exit within n time units of when it is entered. A monitor for this deadline is displayed in Figure 10. Variants similar to those described above for delays are also possible and are monitored in much the same manner.

Mutual Exclusion Assertion. Consider the following assertion:

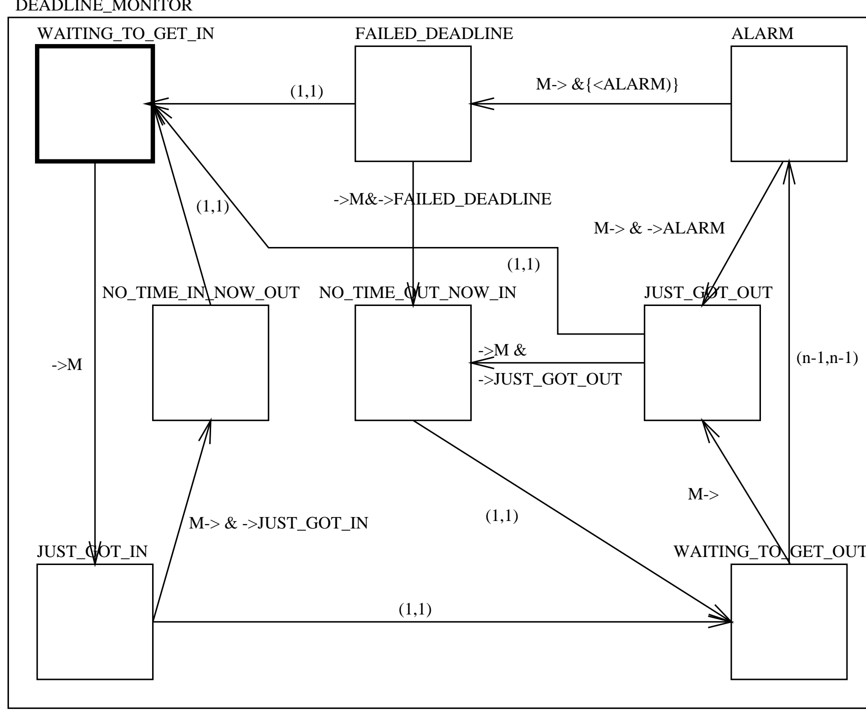


Figure 10: Modechart Monitoring Fragment for a Deadline

$$\forall i, \forall j : [@(M_2\rightarrow, j) < @(\rightarrow M_1, i) \vee @(M_1\rightarrow, i) < @(\rightarrow M_2, j)] \tag{10}$$

This assertion, *mutual exclusion*, states that two modes, indicated by $\mathtt{M}_1$ and $\mathtt{M}_2$ must be disjoint. Here the transitivity of order relations is used to establish a bound on the event history. For a fixed occurrence of the mode $\mathtt{M}_1$, if the last $\mathtt{M}_2$ mode exited before that $\mathtt{M}_1$ started, then all previous $\mathtt{M}_2$'s exited before that $\mathtt{M}_1$ started. That is, despite the fact that the RTL form of this assertion has two universal quantifiers, it is not necessary to compare every occurrence of $\mathtt{M}_1$ with every occurrence of $\mathtt{M}_2$. Rather, as each occurrence of $\rightarrow\mathtt{M}_1$ takes place, it is sufficient to check that $\mathtt{M}_2$ is not active and that throughout the period of time that $\mathtt{M}_1$ is active, that $\mathtt{M}_2$ does not become active. This is demonstrated in Figure 11. Observe how the monitor enters the mode `Just_Became_Bad` under precisely these circumstances. That is to say, $\rightarrow$ `Just_Became_Bad` occurs if `M1` is active and `M2` starts or if `M2` is active and `M1` starts.

Similarly, the satisfiability of the assertion with regard to each $\mathtt{M}_2$ interval can be determined by watching during each $\mathtt{M}_2$ interval for the occurrence of any $\mathtt{M}_1$ intervals. Therefore, it is only necessary to observe and keep the last entry and exit events for each of $\mathtt{M}_1$ and $\mathtt{M}_2$. Notice that a violation can be observed comparing the last entry and exit events for each interval. This idea is further developed in Section 5.

Is_Contained Assertion. Consider the following assertion:

$$\forall i \exists j : [@(J_1, j) < @(I_1, i) \wedge @(I_2, i) < @(J_2, j)] \tag{11}$$

This assertion gives a very general example where one interval is required to be contained within another interval. Here, the events $\mathtt{J}_1$ and $\mathtt{J}_2$ can be seen as forming an interval of time referred to as the "j-interval." Similarly, the events $\mathtt{I}_1$ and $\mathtt{I}_2$ form the "i-interval." This assertion indicates that each "i-interval", comprised of the events $\mathtt{I}_1$ and $\mathtt{I}_2$ must be contained in some surrounding "j-interval" comprised of the events $\mathtt{J}_1$ and $\mathtt{J}_2$. Intervals and modes are described in more detail in Section 5.1.

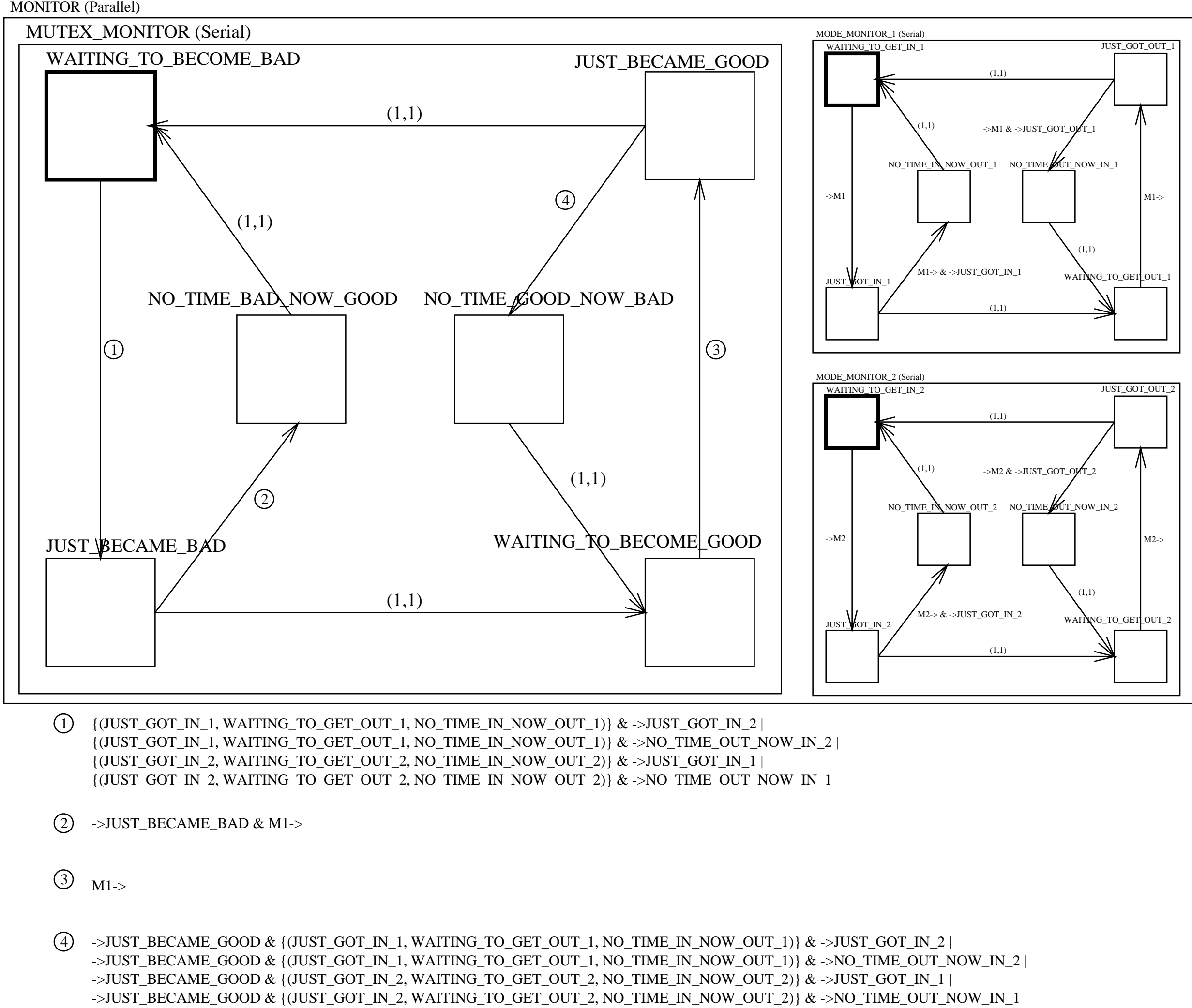


Figure 11: Modechart Monitoring Fragment for Mutual Exclusion

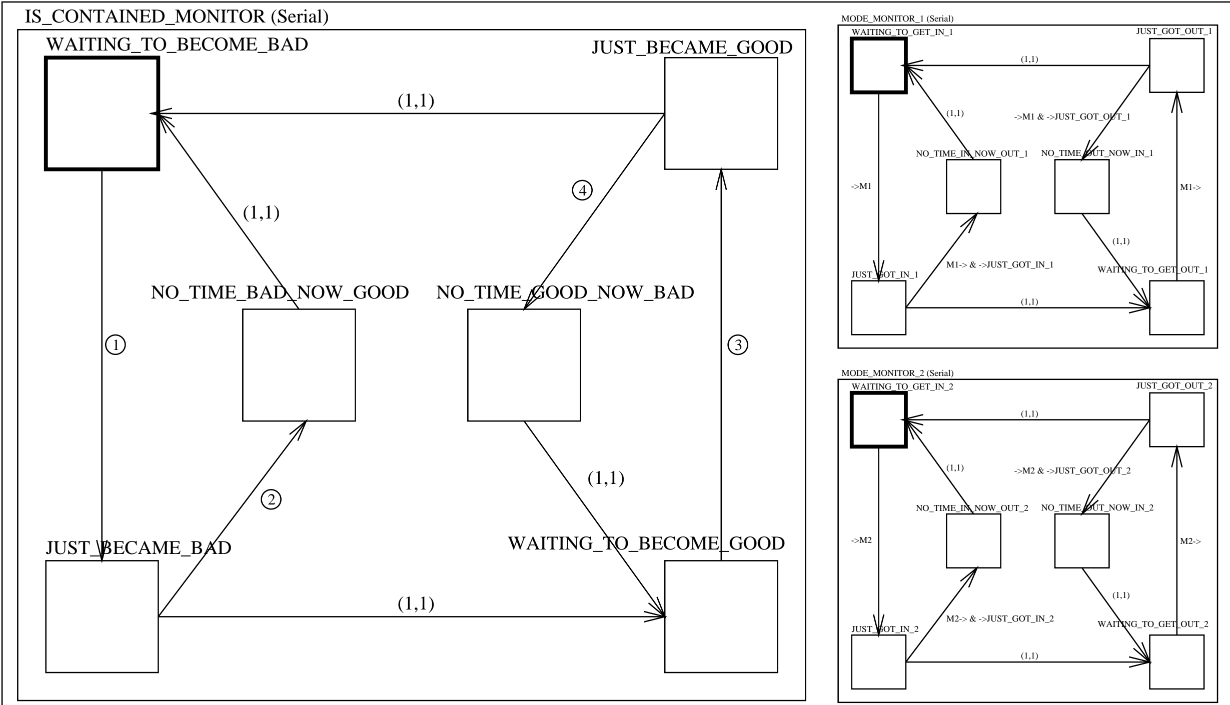
The event history is potentially unbounded unless additional restrictions are made. Consider the computation:

$J_1, I_1, J_1, I_1, J_1, I_1 \ldots$. This sequence indicates a computation where there are multiple “j-intervals” and “i-intervals” open simultaneously. That is to say, the intervals overlap each other.

In order to monitor such a computation for the Is_contained relationship, for each I_1 event occurrence, the preceding J_1 event occurrence must be preserved. This is because at every point in the above sequence there is insufficient information to affirm or refute the assertion. The first J_1 *could* begin an interval that encloses all of the “i-intervals”. Alternatively, that “j-interval” could end before the end of *any* of the “i-intervals.”

However, for each open “i-interval”, we need keep only one J_1 event occurrence. Consider the J_1 occurrence that is the last one occurring before the opening of an “i-interval.” All previous J_1 event occurrences will occur before the beginning of the “i-interval.” Therefore all such J_1 occurrences are candidates for the surrounding interval. When a J_2 event occurs after the “i-interval”, if its index is less than or equal to the last J_1 index preceding the interval, then a “surrounding” interval is detected.

In summary, the event history for Is_contained is unbounded on the “i” dimension, but bounded on the “j” dimension. For each open “i-interval”, the event history is bounded. Therefore an overall bound can



① {(JUST_GOT_IN_1, WAITING_TO_GET_OUT_1, NO_TIME_IN_NOW_OUT_1)} & ->JUST_GOT_OUT_2 |
{(JUST_GOT_IN_1, WAITING_TO_GET_OUT_1, NO_TIME_IN_NOW_OUT_1)} & ->NO_TIME_IN_NOW_OUT_2 |
{(JUST_GOT_OUT_2, WAITING_TO_GET_IN_2, NO_TIME_OUT_NOW_IN_2)} & ->JUST_GOT_IN_1 |
{(JUST_GOT_OUT_2, WAITING_TO_GET_IN_2, NO_TIME_IN_NOW_OUT_2)} & ->NO_TIME_OUT_NOW_IN_1

② ->JUST_BECAME_BAD & M1->

③ M1->

④ ->JUST_BECAME_GOOD & {(JUST_GOT_IN_1, WAITING_TO_GET_OUT_1, NO_TIME_IN_NOW_OUT_1)} & ->JUST_GOT_OUT_2 |
->JUST_BECAME_GOOD & {(JUST_GOT_IN_1, WAITING_TO_GET_OUT_1, NO_TIME_IN_NOW_OUT_1)} & ->NO_TIME_IN_NOW_OUT_2 |
->JUST_BECAME_GOOD & {(JUST_GOT_OUT_2, WAITING_TO_GET_IN_2, NO_TIME_IN_NOW_OUT_2)} & ->JUST_GOT_IN_1 |
->JUST_BECAME_GOOD & {(JUST_GOT_OUT_2, WAITING_TO_GET_IN_2, NO_TIME_OUT_NOW_IN_2)} & ->NO_TIME_OUT_NOW_IN_1

Figure 12: Modechart Monitoring Fragment for the Is_contained Relationship

only be obtained if a bound is established for the number of open "i-intervals."

There are several approaches to bounding the event history for this type of assertion.

- The user may assume the existence of such a bound. In this case monitoring will fail should the assumption be violated.
- Assert a deadline on the duration of the "i-interval". If the "i-interval" is constrained in length, then only a finite number of them can be open simultaneously.
- Restrict the "i-intervals" to be modes. A mode has the property that only one instance of the mode may be open at a given time. That is to say, each mode-entry event must be followed by a mode-exit event before a subsequent mode-entry event can occur. Modes are described in more detail in Section 5; this kind of constraint is the basis for the development of a large class of assertions which can be monitored with a bounded event history.

A Modechart monitoring fragment for monitoring the Is_contained relationship for the case where all intervals are modes is presented in Figure 12; the RTL assertion is provided in Equation 12.

$$\forall i \exists j : [@(\rightarrow M_2, j) < @(\rightarrow M_1, i) \wedge @(M_1 \rightarrow, i) < @(M_2, j)] \tag{12}$$

Here, the fact that modes do not overlap is exploited to facilitate monitoring. In fact, to show that every occurrence of mode `M1` is contained within some occurrence of mode `M2` is equivalent to showing that every occurrence of mode `M1` excludes the occurrence of the intervals when mode `M2` is **not** active. Note the similarity of this monitor with the monitor described for mutual exclusion in Figure 11.

Contains Assertion. Consider the following assertion:

$$\forall i \exists j : [@(I_1, i) < @(J_1, j) \wedge @(J_2, j) < @(I_2, i)] \tag{13}$$

This assertion states that the interval indicated by $\mathtt{I}_1$ and $\mathtt{I}_2$ must contain an occurrence of the interval indicated by $\mathtt{J}_1$ and $\mathtt{J}_2$. Just as for the Is_Contained relationship, the sequence $\mathtt{I}_1, \mathtt{J}_1, \mathtt{I}_1, \mathtt{J}_1, \mathtt{I}_1, \mathtt{J}_1 \ldots$ demonstrates that the event history is potentially unbounded. Furthermore, the event history is bounded only on the dimension indicated by the existential quantifier, but is unbounded if the number of intervals indexed by the universal quantifier is unrestricted. The same approaches to bounding the number of open intervals can be considered.

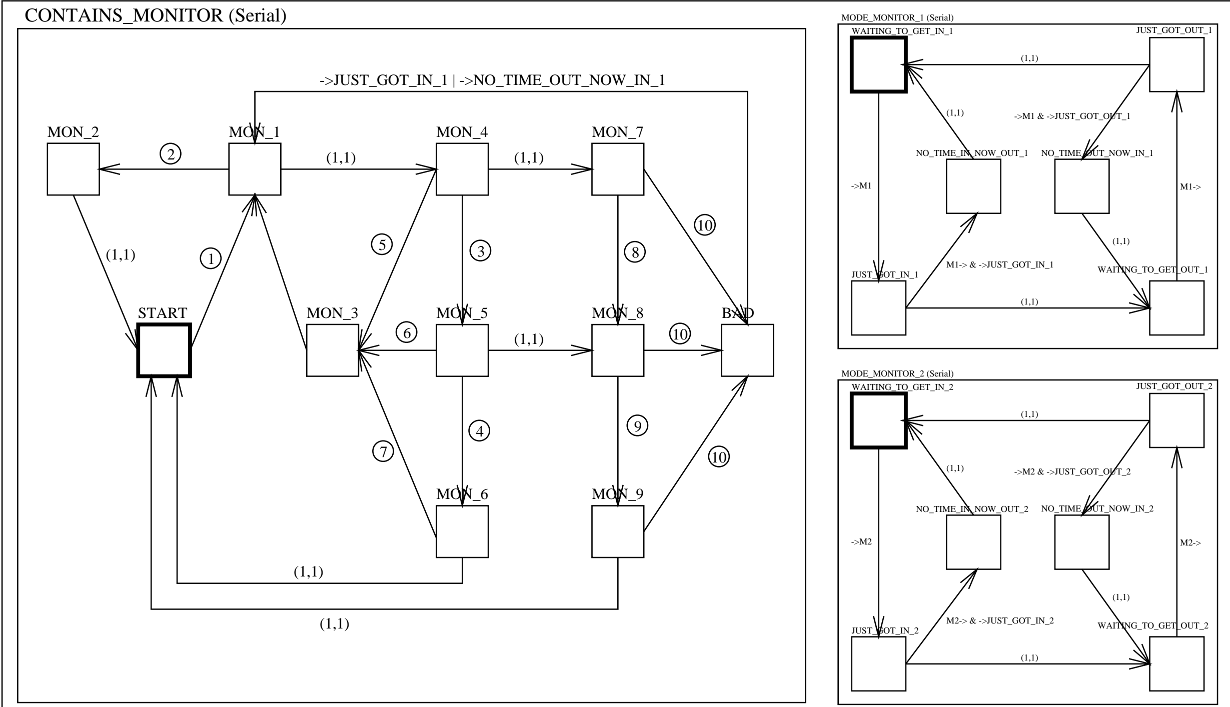
A Modechart monitoring fragment for monitoring the contains relationship for the case where all intervals are modes is presented in Figure 13. This monitor evaluates the RTL assertion:

$$\forall i \exists j : [@(\rightarrow M_1, i) < @(\rightarrow M_2, j) \wedge @(M_1 \rightarrow, j) < @(M_2 \rightarrow, i)] \tag{14}$$

Note that we require the relationship to be strict; that is, the contained mode, $\mathtt{M}_2$, must start strictly after $\mathtt{M}_1$ starts and end strictly before $\mathtt{M}_1$ ends.

5 Translating RTL Assertions into Modechart Specifications

In this section we describe the Fragment Generator part of the MAC tool. The Fragment Generator translates assertions in Real Time Logic into monitoring fragments written in Modechart. As described in Section 3.1, modes are one of the most basic elements of the Modechart language. As a consequence, many of the safety and real time properties likely to be of interest for Modechart specifications are those concerning the behavior of modes. In this section we show that an expressive class of assertions can be composed from primitive relationships between modes and from timing constraints. In addition, we show that the semantics of Modechart can be exploited to guarantee that this particular class of assertions can be monitored with a bounded



(1) ->JUST_GOT_IN_1 | NO_TIME_OUT_NOW_IN_1

(2) ->NO_TIME_OUT_NOW_IN_1

(3) ->MON_4 & ->JUST_GOT_IN_2 |
->MON_4 & ->NO_TIME_OUT_NOW_IN_2

(4) ->MON_5 & ->JUST_GOT_OUT_2 |
->MON_5 &->NO_TIME_IN_NOW_OUT_2

(5) ->MON_4 & ->JUST_GOT_OUT_1

(6) ->MON_5 & ->JUST_GOT_OUT_1

(7) ->MON_6 & ->JUST_GOT_OUT_1

(8) ->JUST_GOT_IN_2

(9) ->JUST_GOT_OUT_2

(10) ->JUST_GOT_OUT_1

Figure 13: Modechart Monitoring Fragment for the Contains Relationship

event history. (As described in 4.3, not all assertions which can be specified in RTL can be monitored with a bounded event history; similar assertions describing more arbitrary events and intervals are not likely to be monitorable with a bounded event history.)

In this section, we first give a formal definition of modes and intervals This is followed by a description of ways in which two modes can be related as well as definitions of RTL assertions capturing these relationships (Section 5.1). We also describe Modechart monitoring fragments for each of these relationships. This is followed by a discussion of how more complex RTL assertions can be composed from RTL formulas describing these primitive relationships (Section 5.2). Finally, we present our approach for automatic composition of the corresponding Modechart fragments to generate Modechart fragments to monitor these more complex RTL assertions.

5.1 RTL Fragments Describing Relationships Between Two Modes

As described above, modes are one of the basic constructs of the Modechart language. A mode can be viewed in terms of the interval of time during which the mode is active, that is, the time elapsed between a mode-entry and the corresponding mode-exit event. In this section, we provide an explicit RTL representation for both intervals and modes. We also describe a set of assertions based on relationships between pairs of modes; these relationships correspond to the monitoring fragments described above in Section 4.3.

Since modes are the primary control mechanism in Modechart, expression of assertions which describe the behavior of modes is likely to be very useful in that environment. Consider the Is_contained relationship discussed in Section 4.3. This assertion describes the relationship between two intervals. Each "j-interval" must be contained within some "i-interval." As described above, in its general form, this assertion cannot be monitored with a bounded event history. Moreover, the reason it is potentially unbounded is the possibility of an arbitrary number of "j-intervals" can be open at a given time. When the "j-intervals" are restricted to be modes, a bound can be easily established for the event history.

This example suggests that it may be easier to establish bounds on assertions describing relationships between modes rather than those describing relationships between arbitrary intervals or even more general statements about events. By restricting our assertions to those describing the manner in which modes can be related to each other, we can ensure that all expressible assertions are monitorable.

An interval can be represented in RTL as a pair of terms (corresponding to the starting and ending points) having the following form:

$@(E_1, i + c_1) + C_1$ and $@(E_2, i + c_2) + C_2$

Each of the terms is called an endpoint. The two endpoints share a single free variable in the occurrence index. One endpoint is designated as "opening" the interval and the other as "closing" the interval. By that is meant that the following axiom is added for each interval:

$$\forall i @(E_1, i + c_1) + C_1 < @(E_2, i + c_2) + C_2$$

That is to say, each occurrence of an interval must open before it closes.

A mode can be represented as one type of interval with the mode entry event indicating the first endpoint and the mode exit event indicating the second endpoint. Such an interval has the additional property that no two occurrences of that interval can be open simultaneously. The following axiom is added for each mode:

$$\forall i @(E_2, i) + C_2 < @(E_1, i + 1) + C_1$$

Allen [2] introduced a taxonomy of temporal relationships for two intervals of time. We introduce in Table 3 a set of primitive relationships loosely derived from his system. The relationships we describe represent a simplification of his scheme; we do not disambiguate pairs of simultaneous events from pairs of that happen in succession. For example, the relationship `A` contains `B` included the possibility that `A` and `B` are equal; the relationship need not be strict. A more detailed set of relationships more closely parallel to his is possible.

Table 3 also contains the RTL *definitions* for the primitive relationships. Each relationship describes the relationship between a *base* mode and another mode, called the *relative* mode. In Table 3, the base mode is always indicated as `A` and the relative mode is indicated by `B`. Note that in these expressions the occurrence index variable for each base mode is a free variable and that the index variable for each relative mode is quantified. In these examples, each of these primitive relationships describes how other modes relate to a given occurrence of mode `A`.

As described in Section 4, RTL assertions must be sentences. Therefore, each of these relationships forms the basis for a primitive assertion formed by universally quantifying the variable connected with the base mode. For example, one assertion based on a primitive relationship, taken from the robot example described above, states that every occurrence of `E.HP.Holding` contains some occurrence of the mode `E.RP.Processing`. In other words, this captures the constraint that every time an item is held by the robot, it is processed by the robot during that period of time.

In RTL, this is represented as:

$$\forall v \exists j @(\rightarrow \text{E.HP.Holding}, v) < @(\rightarrow \text{E.RP.Processing}, j) \wedge @(\text{E.RP.Processing}\rightarrow, j) < @(\text{E.HP.Holding}\rightarrow, v) \quad (15)$$

which is formed by substituting the mode `E.HP.Holding` into the RTL representation for the contains primitive relationship and universally quantifying the resulting expression.

Moreover, these primitive relationships can be composed into more complex assertions as described in Section 5.2.

(B [ ) A]	A left_overlaps B	$\exists j$	$@(\rightarrow B, j) \leq @(\rightarrow A, v_A) \wedge$ $@(\rightarrow A, v_A) \leq @(B\rightarrow, j) \wedge$ $@(B\rightarrow, j) \leq @(A\rightarrow, v_A)$
[(B) A]	A contains B	$\exists j$	$@(\rightarrow A, v_A) < @(\rightarrow B, j) \wedge$ $@(B\rightarrow, j) < @(A\rightarrow, v_A)$
[A (B])	A right_overlaps B	$\exists j$	$@(\rightarrow A, v_A) \leq @(\rightarrow B, j) \wedge$ $@(\rightarrow B, j) \leq @(A\rightarrow, v_A) \wedge$ $@(A\rightarrow, v_A) \leq @(B\rightarrow, j)$
([A] B)	A is_contained_in B	$\exists j$	$@(\rightarrow B, j) < @(\rightarrow A, v_A) \wedge$ $@(A\rightarrow, v_A) < @(B\rightarrow, j)$
[A] (B) or (B) [A]	A excludes B	$\forall j$	$@(B\rightarrow, j) < @(\rightarrow A, v_A) \vee$ $@(A\rightarrow, v_A) < @(\rightarrow B, j)$
[←max n, A→]	A deadline n		$@(A\rightarrow, v_A) < @(\rightarrow A, v_A) + n$
[←min n, A→]	A delay n		$@(\rightarrow A, v_A) + n < @(A\rightarrow, j)$

Table 3: RTL Definitions of Simple Primitive Relationships

We have described Modechart monitoring fragments for most of the primitive relationships in Section 4.3 (Figures 9 to 13). These fragments are used by the monitoring and assertion-checking tool to monitor primitive relationships as well as to compose larger fragments to monitor more complex RTL assertions; this is discussed in Section 5.2.

5.2 Composition of Primitive Relationships

The primitive relationships described above can be composed into more complex assertions. We have described how each primitive relationship has a corresponding Modechart monitoring fragment; similarly, complex assertions have monitoring fragments composed from the Modechart monitoring fragments for primitive relationships. Primitive Relationships concerned with the same *base* mode can be composed using the logical connectives into a *single-base-mode* assertion. For example, a *single-base-mode* assertion composed from two primitive relationships concerned with the same base mode might be "Every occurrence of the mode `S.RC.Process` has a deadline of 9 and has a delay of 6."

When *single-base-mode* assertions are composed, the result is a *complex* assertion. For example, here is a complex assertion taken from the robot specification: "Every occurrence of the mode `S.RC.Process` has a deadline of 9 and every occurrence of the mode `E.RP.Processing` has a delay of 6." Note that the complex assertion describes the behavior of different modes over the computation.

Single-base-mode assertions are fully quantified and, therefore, so are complex assertions. As a consequence, single-base-mode assertions and complex assertions describe constraints on the *entire* computation. In contrast, each primitive relationship describes the behavior of each occurrence of the base mode because it has a free variable as the occurrence index variable for the base mode. The free variable in primitive relationships permits the the composition of more complicated assertions about the behavior of the base mode.

Composition of RTL Assertions. The rules for generating a composed RTL assertion from the primitive relationships are:

1. Let $\alpha(M, v_M)$ and $\beta(M, v_M)$ represent primitive relationships with `M` the base mode and v_M the free variable in α and β. The primitive relationships are specified in RTL as indicated in Table 3.

2. Primitive relationships can be composed using the usual logical connectives to from the matrix of a single-base-mode assertion, thus:

   - $\psi(M, v_M) = \alpha(M, v_M)$
   - $\psi(M, v_M) = (\phi(M, v_M) \wedge \varphi(M, v_M))$ where $\phi(M, v_M)$ and $\varphi(M, v_M)$ are matrices of single-base-mode assertions.
   - $\psi(M, v_M) = (\phi(M, v_M) \vee \varphi(M, v_M))$ where $\phi(M, v_M)$ and $\varphi(M, v_M)$ are matrices of single-base-mode assertions.
   - $\psi(M, v_M) = (\neg\phi(M, v_M))$ where $\phi(M, v_M)$ is a matrix of a single-base-mode assertion.

   Note that without loss of generality it is possible to assume that the primitive relationships are composed in disjunctive normal form.

3. Replace each *single-base-mode assertion* with its universal closure:

   $\phi^* = \forall v_M \phi(M, v_M)$ where $\phi(M, v_M)$ is a matrix of a single-base-mode assertion.

4. Single-mode assertion may be composed together in disjunctive normal form using the usual logical connectives. For example,

   $\phi^* \wedge \varphi^*$ where $\phi^* = \forall v_A \phi(A, v_A)$ and $\varphi^* = \forall v_B \varphi(B, v_B)$ is a complex assertion.

   Again, without loss of generality it is possible to assume that the single-base-mode assertions are composed in disjunctive normal form.

For example, the second assertion described above is specified in RTL as:

$$\forall i @(S.RC.Process\!\rightarrow, i) < @(\rightarrow\! S.RC.Process, i) + 9 \wedge @(\rightarrow\! S.RC.Process, i) + 6 < @(S.RC.Process\!\rightarrow, i) \tag{16}$$

Incidentally, the simple phrases, "contains," "excludes," "left_overlaps,", etc. can be used to describe these composed assertions in a straightforward manner. These phrases correspond to the templates presented to the user in the "forms" interface for specifying assertions. In this way the burden of writing RTL directly is avoided–the user has only to fill in correct mode names. Furthermore, the task of writing assertions that conform to the above requirements for composed assertions is simplified by permitting the user to state assertions in terms of these phrases linked with the propositional connectives.

Composition of Modechart Monitoring Fragments. Monitoring fragments for composed RTL assertions (such as the one described in Equation 16) can be constructed by using Modechart monitoring

fragments corresponding to each of the primitive components of the assertion. Each time any of the monitoring fragments has a mode change, the monitoring system combines the values of the modes for each primitive relation using the logical connectives to determine the satisfiability for the composite assertion.

The composition of the Modechart Monitoring fragments is structurally similar to that of the RTL assertions; that is, monitoring fragments are instantiated for each primitive relationship and composed to form monitoring fragments for single-base-mode assertions. The resulting Modechart monitoring fragments for single-base-mode assertions are then composed to develop Modechart monitoring fragments capable of monitoring the entire complex assertion.

Modechart monitoring fragments for primitive relationships are based on the fragments depicted above in Figures 9 to 13. To simplify the construction of the monitoring fragments, each of the fragments for primitive relationships is composed in parallel with a control mode. That is to say, a parent mode is created having two serial children, one of which is the relevant monitoring fragment instantiated for the particular primitive relationship and the other of which is the control mode. The control mode has two atomic children, one representing all of the "good" or satisfiable modes in the template and one representing the disjunction of the "bad" or unsatisfied modes of the template. A transition from the satisfiable to the unsatisfied mode is triggered by the disjunction of all of the entry events into unsatisfied modes in the template mode; similarly, a transition back into the satisfiable mode is triggered by the disjunction of all of the entry events into satisfiable modes in the template mode. Thus, the control mode contains a simplified representation of the template mode, entering a satisfiable mode any time the template is in a satisfiable mode and entering an unsatisfied mode any time the template is in an unsatisfied mode. The purpose of the control mode is merely to simplify the representation of the behavior of the template mode to facilitate parallel composition with monitoring fragments for other primitive relationships.

For each single-mode assertion consisting of two or more primitive relationships, a higher level monitoring fragment is created. This parallel mode has as children each of the relevant monitoring fragments for primitive relationships (as described above with control modes) as well as its own control mode. The control mode for each simple assertion has two modes, one for satisfiable modes and one for unsatisfiable modes, just as the control mode for primitive relationships does. However, there is just one transition, from the satisfiable mode to the unsatisfied mode. The trigger condition for this transition is the negation of the matrix of the RTL specification of the single-mode assertion. Each primitive assertion is replaced by the name of the corresponding unsatisfied mode for its monitoring fragment and the resulting expression is negated to derive the trigger condition for the transition into the unsatisfied mode for the simple assertion.

For example, the generation of the Modechart monitoring fragment for the assertion stated in Equation 16 is depicted in Figure 14. Here, the assertion to be monitored states that for each occurrence of the mode `S.RC.Process` that occurrence does not exit in less than 6 time units and it exits within 9 time units.

Note that this monitoring fragments is structured according to the above description with monitoring fragments corresponding to each of the primitive relationships. Each of the two primitive relationships, `S.RC.Process` has a delay of 6", and `S.RC.Process` has a deadline of 9" are monitored by the corresponding modes `A1.Deadline` and `A1.Delay`. `A1.Deadline.Controls` and `A1.Delay.Controls` are the relevant control modes which summarize the behavior of each of the monitoring fragments for each primitive relationship. The single-base-mode assertion which is composed of these two primitive relationships is monitored by the mode `A1` having control mode `A1.S.RC.Process.Controls`. `A1.S.RC.Process.Controls` is a new control mode introduced to monitor the behavior of the single-base-mode assertion composed of the deadline and the delay relationships. This control mode has two children `A1.S.RC.Process.Satisfiable` and `A1.S.RC.Process.Unsatisfied`. The transition from `A1.S.RC.Process.Satisfiable` to `A1.S.RC.-Process.Unsatisfied` is controlled by a transition triggered by the disjunction of the two modes representing the unsatisfied modes of the primitive relationships. As this is a single-base-mode assertion, this monitoring fragment is sufficient. If it were a complex assertion, a further level of controls would be introduced to monitor the composition of the single mode assertions.

Modechart monitoring fragments for single-mode assertions can be composed into monitoring fragments for complex assertions in exactly the same way. The monitoring mode for a complex assertion has as children monitoring fragments for each simple assertion (as described above) as well as its own control mode. This

Figure 14: Composed Modechart Monitoring Fragments for example 16.

control mode has satisfiable and unsatisfied modes with a single transition from the satisfiable mode to the unsatisfied mode defined in exactly the same manner as for the composition of single-mode assertions.

6 Related Work

The foundations for our work are found in [12] which develops the framework for understanding a system computation in terms of an event-based model. The problem of bounding event histories is described here as well as the distinction between synchronous and asynchronous monitoring. In addition, this work presents several classes of assertions which can be monitored with bounded event histories, one of which (Exclusion/Inclusion/Overlapping of Intervals) serves as the impetus for the primitive relationships we describe. A more formal approach to real time monitoring, including the development of the idea of a computation prefix is provided in [13].

More attention has been given to the issues of on-line or run-time monitoring than to that of monitoring and assertion checking of *symbolic executions* of real time systems. Run-time monitoring systems must address problems related to the *probe effect* including perturbation and event reordering, in addition to the issues we describe. Moreover, the research has focused on monitoring facilities for parallel and distributed systems, rather than for real time systems. See [28] for many of the major contributions with regard to distributed systems as well as several which address monitoring in the context of real time systems.

Special hardware support for collecting run-time data in real time applications has been considered in a number of recent papers [8, 19, 27]. These approaches introduce specialized co-processors for the collection and analysis of run-time information. The use of special-purpose hardware allows non-intrusive monitoring of a system by recording the run-time information in a large repository, often for post analysis. A related work [9] studies the use of monitoring information to aid in scheduling tasks. The underutilization of a CPU due to the use of scheduling methods based on worst-case execution times is addressed by the use of a hardware real time monitor which measures task execution times and delays due to resource sharing. The monitored information is fed back to the operating system for achieving an adaptive behavior. These approaches emphasize event detection rather than the issues connected with monitoring more complex behaviors of a system.

[24] gives an overview of the important issues of run-time monitoring for real-time distributed systems and illustrates these issues in terms of the test methodology developed for the MARS system [19]. This approach uses special monitoring nodes between system nodes to collect communication events which are saved for later analysis. Because the monitoring nodes do not contribute any network traffic, interference is avoided.

A work closer to our approach is a system for collection and analysis of distributed/parallel (real time) programs [18]. The work is based on an earlier system for exploring the use of an extended E-R model for specification and access to monitoring information at run-time [25]. The assumption is that the relational model is an appropriate formalism for structuring the information generated by a distributed system.

A real time monitor developed for the ARTS distributed operating system is presented in [26]. The proposed monitor requires certain support from the kernel, such as notification of the state changes of a process, including waking-up, and being scheduled. In particular the ARTS kernel records these events that are seen by the operating system as the state changes of a process. These events are sent periodically by the local host to a remote host for displaying the execution history. The invasiveness of the monitoring facility is included in the schedulability analysis.

Monitoring and detecting violations of certain predefined timing constraints have been proposed in real time languages, such as FLEX [17]. The FLEX language provides the constructs for specifying delay and deadline constraints in a program.

For a discussion of an approach to run-time monitoring of RTL assertions in distributed real time systems see [15]. This paper discusses the additional issues of early detection of violations, minimization of the number of messages required for monitoring, and clocks and timer granularity.

The STATEMATE system [11], based on Statecharts [10], provides symbolic execution of system spec-

ifications. In an approach similar to ours, these executions can be monitored via special "watchdog" code which is defined by the user in Statecharts. There is no automatic generation of watchdog code. It is also possible to set breakpoints in the Statechart code. In addition, the STATEMATE system provides a simulation control language to provide additional control over the generation of execution traces.

History-checking of TRIO specifications is provided in the recent work of [6]. TRIO is a first order temporal logic which deals with time in a quantitative way by providing a metric to indicate distance in time between events. History-checking is provided by applying a tableaux-based algorithm to a history (execution trace) of a TRIO specification. In addition to the fact that a specification of real time systems in a temporal logic such as TRIO looks very different from one specified in the state-transition diagrams of Modechart, one important way that their approach differs from ours is that their history-checker examines an entire history which has been stored in a data structure, rather than examining a computation as it is generated as we do. In contrast, an important goal in our research was identifying assertions for which a bound on the event history could be established independently from the duration of the compuation, whereas the bound in the TRIO approach results from an assumption of a finite temporal domain.

7 Future Work and Conclusions

In this paper we have presented a framework for monitoring execution traces generated from Modechart specifications. We have presented an assertion language, Real Time Logic, and defined monitoring in terms of determining satisfiability of those assertions under a sequence of interpretations generated by computation prefixes. In addition, we have identified the three primary challenges of monitoring, namely, bounding of the amount of information that must be recorded, when checks must be performed, and how to check a particular instance of an assertion. We have exploited the semantics of Modechart to guarantee that monitoring is possible with a bounded event history for an expressive class of assertions based on comparisons between two modes and on timing constraints for individual modes. We have described an approach to monitoring these assertions based translating assertions specified in Real Time Logic into monitoring fragments specified in Modechart. An implementation of a monitoring and assertion checking tool (MAC) for for the Modechart Toolset has been completed.

Future work includes enriching the Modechart Toolset by extending simulator options to allow developers more control over the generation of execution traces. Ideally, the monitoring system would be capable of providing feedback to the simulation to permit "steering" of an execution. In addition, we would like to augment the user interface to the simulator tool, supplementing the current "time-process" display with an animated display of the execution of the specification. An animated display would use color or highlighting to indicate active modes and enabled transitions. These extensions to the current toolset would further integrate the monitoring and assertion-checking tool with the rest of the toolset and would make the tool more useful for system designers.

Finally, we would like to incorporate *actions* into the Modechart Toolset. Actions are procedures attached to modes that are executed upon the occurrence of a mode-entry event. In addition to providing Modechart with general data handling capabilities, actions would provide a valuable extension to the monitoring and assertion-checking tool. For example, actions connected to monitoring fragments could compile performance statistics concerning the simulated execution.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, Reading, MA, 1974.

[2] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[3] B. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[4] S. Chodrow, F. Jahanian, and M. Donner. Run-time monitoring of real-time systems. In *Proc. of 12th Real-Time Systems Symposium*, pages 74–83, December 1991.

[5] 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.

[6] Miguel Felder and Angelo Morzenti. Validating real-time systems by history-checking trio specifications. In *Procedings of the 14th International Confernence on Software Engineering*, New York, NY, 1992. ACM Press.

[7] D.A. Gabel. Technology 1994: Software engineering. *IEEE Spectrum*, 31(1):38–41, january 1994.

[8] D. Haban and K. G. Shin. Application of real-time monitoring to scheduling tasks with random execution times. *IEEE Trans. Software Engineering*, 16(12):1374–1389, December 1990.

[9] D. Haban and D. Wybranietz. A hybrid monitor for behavior and performance analysis of distributed systems. *IEEE Trans. Software Engineering*, 16(2):197–211, February 1990.

[10] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.

[11] D. Harel et al. Statemate: A working environment for the development of complex reactive systems. *IEEE Trans. Software Engineering*, SE-16(4), April 1990.

[12] F. Jahanian. Run-time monitoring of real-time systems. In Sang H. Son, editor, *Advances in Real-Time Computing*, chapter 18. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[13] F. Jahanian and A. Goyal. A formalism for monitoring real-time constraints at run-time. In *Proc. of Fault-Tolerant Computing Symposium (FTCS-20)*, June 1990.

[14] F. Jahanian and A. K. Mok. Modechart: A specification language for real-time systems. *IEEE Trans. Software Engineering*, 20(10), 1994.

[15] F. Jahanian, R. Rajkumar, and S. Raju. Monitoring timing constraints in distributed real-time systems. *Journal of Real-Time Systems*, 7(3), 1994.

[16] Farnam Jahanian and Aloysius Ka-Lau Mok. Safety analysis of timing properties in real-time systems. *IEEE Trans. Software Engineering*, SE-12(9):890–904, September 1986.

[17] Kevin B. Kenny and Kwei-Jay Lin. Building flexible real-time systems using the flex language. *IEEE Computer*, 24(5):70–78, May 1991.

[18] C. Kilpatrick, K. Schwan, and D. Ogle. Using languages for capture, analysis, and display of performance information for parallel or distributed systems. In *Int'l Conf. on Computer Languages*, 1990.

[19] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The MARS approach. *IEEE Micro*, 9(1):25–40, February 1989.

[20] N. G. Leveson. Software safety. 11th IEEE Workshop on Real-Time Operating Systems and Software, May 1994. Invited Talk.

[21] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. In Jeffrey J. P. Tsai and Steve J. H. Yang, editors, *Monitoring and Debugging of Distributed Real-Time Systems.* IEEE Computer Society Press, Los Alamitos, CA, 1995.

[22] R. Pressman. *Software Engineering A practitioner's Approach.* McGraw Hill, New York, NY, 3rd edition, 1992.

[23] 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.

[24] Werner Schütz. Fundamental issues in testing distributed real-time systems. *Real-Time Systems*, 7(2):129–157, September 1994.

[25] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Trans. Computer Systems*, 6(2):157–196, May 1988.

[26] Hideyuki Tokuda, Makoto Kotera, and Clifford W. Mercer. A real-time monitor for a distributed real-time operating system. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):68–77, January 1989.

[27] J. J. P. Tsai, K.-Y. Fang, and H.-Y. Chen. A non-invasive architecture to monitor real-time distributed systems. *IEEE Computer*, 23(3):11–23, March 1990.

[28] Jeffrey J. P. Tsai and Steve J. H. Yang, editors. *Monitoring and Debugging of Distributed Real-Time Systems.* IEEE Computer Society Press, Los Alamitos, CA, 1995.