A Method for Verifying Properties of Modechart Specifications

Farnam Jahanian† and Douglas A. Stuart†

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Key words: *real-time systems, specification, verification, timing constraints, Modechart, RTL*

Abstract

As software control of time-critical functions in *embedded systems* becomes more common, a means for the precise specification of their behavior becomes increasingly important. Modechart is a graphical specification language introduced to meet this need. This paper presents a method for verifying properties of systems specified in Modechart. The proposed approach makes use of a *computation graph* which takes advantage of the structuring inherent in a Modechart specification. Two classes of properties are presented for which decision procedures are developed.

1. Introduction

Modechart is a graphical specification language developed to provide a compact and structured way to represent real-time systems [Jahanian & Mok 88]. Although similar in some ways to Harel's Statecharts [Harel 86], Modechart is specifically tailored to representing time-critical systems. The semantics of Modechart is given in Real Time Logic (RTL), a logic for the specification and analysis of such systems [Jahanian & Mok 86]. The goal of this research is to use the structuring of Modechart to produce a decision procedure for classes of properties expressed as RTL formulas. The method proposes to use the structuring information present in the Modechart description of a system to generate a *computation graph* of the system, and to employ this graph to determine the satisfiability of formulas for which this method is appropriate.

Although Modechart is graph-oriented, the model of computation it uses considers a computation to be a sequence of sets of events, i.e., a partial ordering of events. Since this model of

† Supported in part by a research grant from the Office of Naval Research under ONR contract number N00014-85-K-0117, by a University Research Initiative grant under ONR contract number N00014-86-K-0763, and also by a DoD-University Research Instrumentation Program grant under ONR contract number N00014-86-G-0199

computation does not include the notion of state, there is no concept of an invariant holding in all the states, as in most of the verification literature (e.g., [Lamport 83], [Ostroff 87], [Shankar & Lam 87]). Although this is a high price, most of the techniques which use invariance rely on transition systems as their model of computation. Transition systems may not be appropriate models for hard real-time systems, because transition systems generally rely on interleaving to model concurrency, which can pose problems when reasoning about timing properties of systems, as discussed in [Jahanian & Mok 86]. This paper is an attempt to reconcile these two differing approaches, so that the advantages of the invariant will be available for reasoning about this new model of computation.

The presentation will follow the same general outline as the preceding discussion. Sections 2 and 3 present an overview of RTL and Modechart. Then in sections 4 through 6 computation graphs will be introduced. Section 7 presents an example illustrating these concepts. Lastly, in section 8 two classes of RTL formulas will be introduced and the corresponding decision procedures will be developed.

2. Real Time Logic

Real Time Logic views a real-time system as a class of event types. Usually these will include both events that are a part of the system, such as those denoting the start and stop of the actions of the systems, and those associated with the changing of values of the various attributes of the system, and events that are a part of the environment of the system, which are the usual external events. The logic views the computations of such a system as a sequence of events of these various types, known as *event occurrences*, and each event occurrence has both a type and a time. The type indicates of which event it is an occurrence, and the time is an integer which represents the absolute time of the event occurrence, measured from the beginning of the computation.

The class of events is generally subdivided as indicated above into the following subclasses, and each likewise has a standard notation. One such subclass consists of the start and stop events for each action of the underlying system. For each action A, its corresponding start and stop events are represented by $\uparrow$A and $\downarrow$A, respectively. Another subclass consists of the transition events for each state variable of the system. For simplicity, each state variable S is assumed to span the Boolean domain, and is represented by the pair of events (S:=T) and (S:=F), which denote the change of the value of the variable from false to true and true to false, respectively. The last subclass that is usually distinguished is that of the external events. Each such event is represented in RTL by its name, prefixed by Ω. For example the external event corresponding to the landing of an aircraft would be ΩLAND.

RTL is an extension of integer arithmetic without multiplication (Presburger arithmetic) that adds a single uninterpreted binary function representing the relationship between events of a system, and their times of occurrence. This function, called the *occurrence function*, is denoted by @. As an example of the use of @, the time after the beginning of the computation that an event *e* occurs for the twelfth time is represented as @(*e*,12). The occurrence function is a mapping

$$@ : \ \mathbf{E} \times \mathbf{Z}^{+} \rightarrow \mathbf{N}$$

(where **E** is a set of event constants, $\mathbf{Z}^{+}$ is the set of positive integers, and **N** is the set of natural numbers) such that @ is strictly increasing in its second argument, i.e., if @(E,i) = t and @(E,i+1) = u, then t < u. Also, RTL includes only integer variables. There are no event variables, and therefore quantification over events is not permitted. Likewise, RTL is a first order logic, and contains no uninterpreted predicate symbols. The formulas of RTL are therefore boolean combinations of the various equality and inequality predicates of standard integer arithmetic, where the arguments of the relations are integer valued expressions involving variables, constants, and applications of the function symbol @.

$$\forall i \ \ @(\Omega\text{CTRL_D},i) \leq @(\uparrow\text{LOGOUT},i) \ \wedge$$
$$@(\downarrow\text{LOGOUT},i) \leq @(\Omega\text{CTRL_D},i) + 20$$

is an example of an RTL formula, which states that each time CTRL_D is pressed, the action LOGOUT begins, and the action must end twenty time units later.

3. Modechart

Modechart is a language for constructing *modes*, which can be regarded as control information that impose structure on the operation of a system. Modes are arranged hierarchically. Each mode is either *primitive*, *parallel*, or *serial*. A primitive mode has no internal structure. A parallel mode is constructed from a collection of child modes by parallel composition, and the children are said to be *in parallel*. The parallel relationship among several modes indicates that a system operates in all of these modes simultaneously. The entry of a parallel mode causes the simultaneous entry of all of its children. Similarly, the exit of the mode causes the simultaneous exit of all of the child modes. In comparison, a serial mode is constructed from a collection of child modes by serial composition, and the children are said to be *in series*. The serial relationship among several modes indicates that (when in the parent mode) the system operates in exactly one of these modes at any time. The entry and exit of a serial mode causes the simultaneous entry and exit of exactly one child mode. Modechart also allows at most one action to be associated with

each mode. The action is executed upon mode entry.

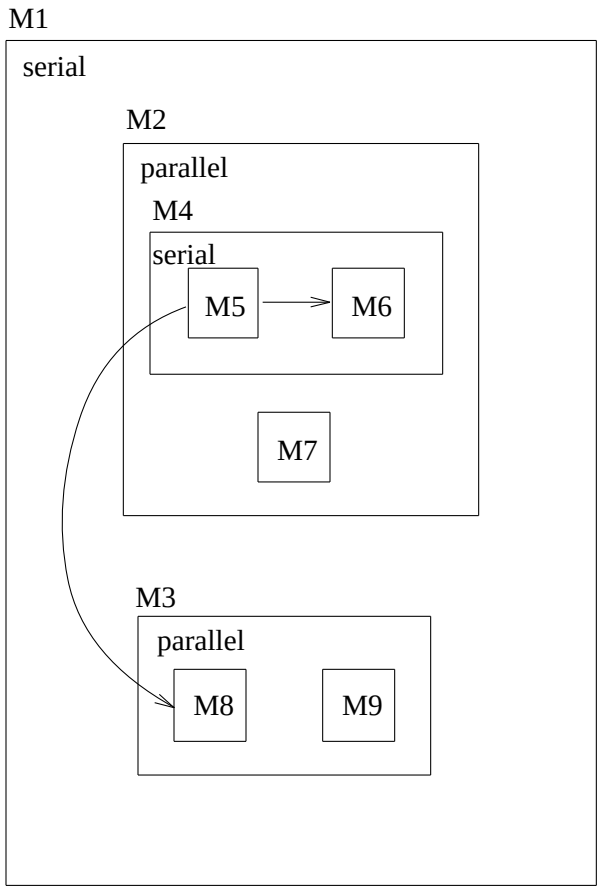


FIGURE 1.

The example of Figure 1 shows a system, serial mode M1, that is made up of two parallel modes, M2 and M3. Mode M2 contains the primitive mode M7, and the serial mode M4, which contains the primitive modes M5 and M6. Mode M3 is a parallel mode composed from the primitive modes M8 and M9. Modes M2 and M3 are in series, and modes M4 and M7 are in parallel.

Modechart also provides for the specification of transitions between modes. A transition is represented graphically by a directed edge between the two modes. A transition has three components, a source mode, a destination mode, and a condition (omitted in the figure). A transition between two modes represents a change in the control information of the system. A mode transition is an instantaneous event which takes zero time units.

When describing modechart specifications in RTL, two additional subclasses of events are distinguished. The events associated with transitions are of the form (M1-M2), where M1 is the source mode of the transition, and M2 is the destination mode. Mode entry and exit events are

similar to the transition events for state variables. The events (M:=T) and (M:=F) are the events corresponding to entering and leaving mode M.

The condition for a mode transition is of the form

$$c_1 \vee c_2 \vee \cdots \vee c_n$$

where each disjunct c_k is either (1) a triggering condition for taking the transition, or (2) a lower/upper bound restriction on when the transition may be taken.

(1) Triggering Condition: A disjunct c_k denoting a triggering condition is of the form

$$p_1 \wedge p_2 \wedge \cdots \wedge p_m$$

where the $p_j s$ specify a condition for taking the transition depending on the occurrence of an event and/or the truth values of certain predicates. In particular, each subcondition p_j is of one of the three forms shown below. (E denotes an event, S is a state variable and t denotes the time at which the transition is taken.)

(a) S (or $\bar{S}$)
Enabling subcondition is a state variable S being true (or false) at time t.

(b) in $\{M_1, M_2, \cdots, M_m\}$
Enabling subcondition is the system being in one of the specified modes at time t.

(c) E
Enabling subcondition is the occurrence of an event E at time t where E can be any event.

A transition from a mode is taken at a time t iff the condition is satisfied at time t when the system is in that mode.

(2) Lower/Upper Bound Condition: A condition c_k denoting a lower/upper bound restriction is of the form (r,d), where r is a non-negative integer denoting a delay, and d is a positive integer or ∞ denoting a deadline. If a lower/upper bound is specified as the condition for a transition from a mode, the transition can be taken after r time units and before d time units has elapsed since entering the mode. The conditions *delay r* and *deadline d* are abbreviations for (r,∞) and (0,d), respectively.

Separation of lower/upper bound and triggering conditions simplifies generation of computation graphs in section 6, by associating timing constraints with mode entry. This does not result in a decrease in expressive power because the result of a conjunction of a lower/upper bound condition and a triggering condition can be achieved by introducing additional modes.

In order to ensure that at all times if a system is in a parallel mode it is in all of its children, and if a system is in a serial mode, it is in exactly one of its children, each transition must satisfy the following condition. There must be some serial mode containing both the source mode and the destination mode of the transition, and each must be contained in a different child mode of that common serial mode. Again informally, when a transition is taken, the system exits all modes which do not contain both the source and destination modes. Furthermore, it enters all modes that contain the destination mode, but not the source mode. The definition of parallel modes may require other modes to be entered and exited as well (modes in parallel with modes that are entered or exited). For example, in Figure 1, the transition from M5 to M8 is possible because M5 is contained in M2, and M8 is contained in M3, two distinct children of the serial mode M1. Taking the transition causes modes M5, M4, M7, and M2 to be exited, and M3, M8, and M9 to be entered.

The formal definition of Modechart is beyond the scope of this paper. More formal treatment of the syntax and semantics of Modechart appeared in [Jahanian & Mok 88], [Jahanian et al 88b].

4. The Approach

The objective is to use the Modechart specification to consider only a finite number of event occurrences to verify a system property. First, we define the computations of a system specified in Modechart as a directed tree augmented with a set of relations expressing the lower/upper bounds on pairs of events. If we can represent the potentially infinite computation tree by a finite graph, then if a procedure can decide a class of properties for this graph, it is a decision procedure for the original Modechart specification.

Figure 2 illustrates the proposed approach. A Modechart specification is used to generate a finite *computation graph*. While generating the graph, the corresponding RTL formulas can be used to prune the vertices that are unreachable due to the timing constraints. These RTL formulas pose little difficulty in an analysis because constants are used for event instances in the occurrence functions. The key is in generating the computation graph so that it accurately reflects the modechart specification. A model for a modechart specification is an assignment of times to events. Likewise, a model of a computation graph is an assignment of times to the events on a path in the graph. A model for a Modechart specification, i.e., for the RTL formulas that capture the formal semantics of the specification, must be shown to be a model for the computation graph. Then if we prove a property about the computation graph, it also holds for the original Modechart specification. However, if we can also show that a model for the computation graph is

a model for the Modechart specification, then a decision procedure for a class of properties for the computation graph is also a decision procedure for the Modechart specification.

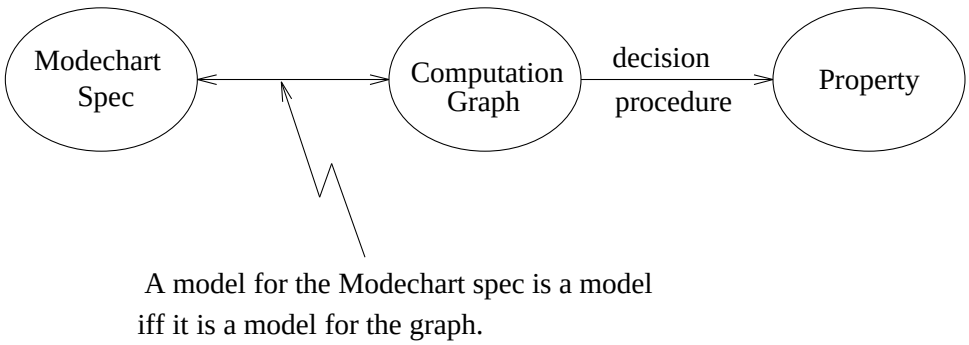


FIGURE 2.

Although the approach proposed in this paper is similar to the reachability analysis for timed-petri nets, e.g.,[Razouk & Phelps 84], [Zuberek 1980], [Ramchandani 74], [Merlin & Farber 76], there are several major differences. The model of computation used in the petri net approach relies on interleaving. The restrictions imposed on the types of nets (such as decision free nets, free choice nets and nets with conflict sets) and the restrictions on the timing constraints limit the systems that can be described. Also, the reachability algorithms, for example the one presented in[Razouk & Phelps 84], rely on the presence of counters in the nodes of the reachability graph to measure elapsed times. The approach in this paper allows nodes with different timer values to be merged into a single node, resulting in a potentially smaller graph. Lastly, a node in a reachability graph represents a system state and determines the truth value of predicates. A node in a computation graph more closely represents a point in time at which an event occurs, and does not assign a truth value to predicates.

5. Computations of a System

The computations of a system specified in Modechart can be viewed as a directed tree in which the nodes are labelled with events and the edges denote causality. The nodes in a computation tree are referred to as *points* throughout this report. The sink node of an edge represents the occurrence of an event caused by the events on the path from the root to that point. A computation tree is augmented by a set of timing requirements representing the lower/upper bound requirements on separation between pairs of (not necessarily consecutive) events in a path from the root. A lower bound separation requirement imposes a non-negative minimum separation on

the two events. An upper bound separation requirement imposes a maximum separation on the two events; it can be either a natural number or ∞. (The timing requirements are obtained from the Modechart specification.)

A *computation of a system* is defined to be an assignment of time values to the events on a path (perhaps infinite) from the root of the tree such that it is consistent with lower/upper bound requirements on the events. Consider a path consisting of a sequence of points P_0, P_1, ... Each point P_i represents an event occurrence. An edge from P_i to P_{i+1} represents the occurrence of an event at point P_{i+1} which was caused by the events from P_0 to P_i. The timing requirements of the system, as described by the Modechart specification, impose minimum/maximum separation on the points (actually event occurrences) in the path. For a pair of points P_i and P_j on a path where i<j, the timing requirements can be represented as

$P_i + I \leq P_j$ (lower bound separation requirement)

$P_j \leq P_i + I$ (upper bound separation requirement)

where I is a non-negative integer. If no restriction is imposed on the two points, it is assumed that the minimum separation from P_i to P_j is at least zero, and the maximum separation is ∞. So there is at least zero minimum separation between each pair of consecutive points, i.e., $P_i \leq P_{i+1}$.

A point can be labelled with more than one event if the events are simultaneous. For example, a point labelled with the mode transition event (M1-M2) represents the simultaneous occurrence of two events, the mode exit event (M1:=F) and the mode entry event (M2:=T). However, two simultaneous events are not necessarily labelled at the same point. Two distinct points can be labelled each with one of the simultaneous events, but the lower/upper bound separations must then permit the events to happen at the same time.

Example 1:

Consider the computation tree in Figure 3. Each point (node) is labelled with a set of events that happen at that point. The sequence of points P_0, P_1, P_2, $\cdots$ denotes a path from the root. The point P_0 is the root indicating that the system is initially in modes M1 and M3 and the state variable S is false. The point P_1 corresponds to the occurrence of the event (M1-M2) which denotes that a mode transition from M1 to M2 is taken at that point. The following lower/upper bound separations may be imposed on the points in the path segment P_0 to P_5. The inequalities are intended to denote bounds on the instances of events corresponding to each point.

$P_0 + 10 \leq P_1$ delay on transition M1 to M2

$P_2 + 20 \leq P_3$ computation time of action A

$P_1 + 150 \leq P_5$	delay on transition from M2 to M4
$P_3 \leq P_1 + 100$	deadline on action A
$P_4 \leq P_3$	(S:=T) happens at same time as ↓A

The lower and upper bound separation between P_3 and P_4 are zero, thus the two points (i.e., the corresponding events) are simultaneous. An assignment of time values to the event occurrences in the above path such that they do not violate the lower/upper bound requirements is a computation of the system.

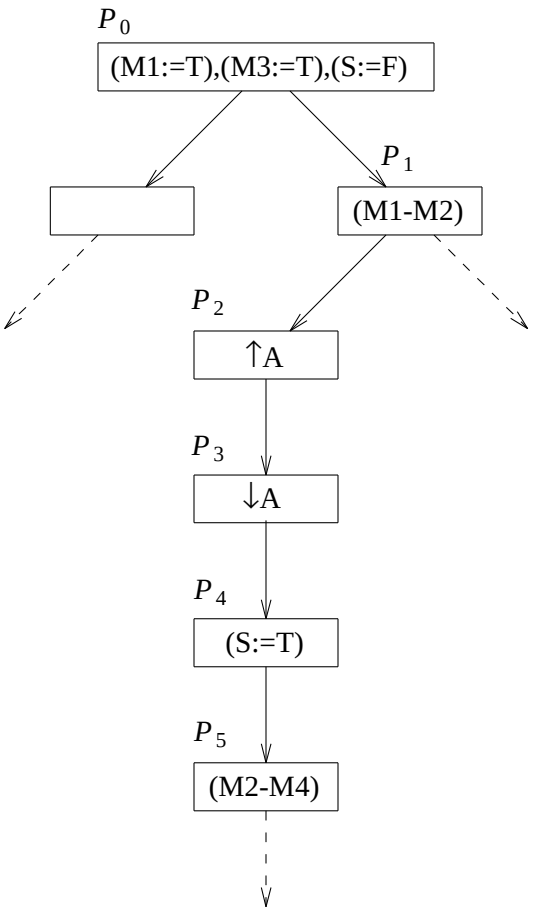


FIGURE 3.

The set of bound separations on a path in a computation tree can in turn be represented by a directed graph, referred to as a *separation graph*. The nodes in this graph are the points in the corresponding path and the weights on the edges denote the specified separation between the nodes. Specifically, given a pair of points P_i and P_j on a path, a separation between the points,

$$P_i + \mathrm{I} \leq P_j$$

is represented in the graph by an edge from P_i to P_j with the weight I. Figure 4 illustrates the corresponding separation graph for the path in Example 1. Notice that the edges corresponding to deadlines have non-positive weights.

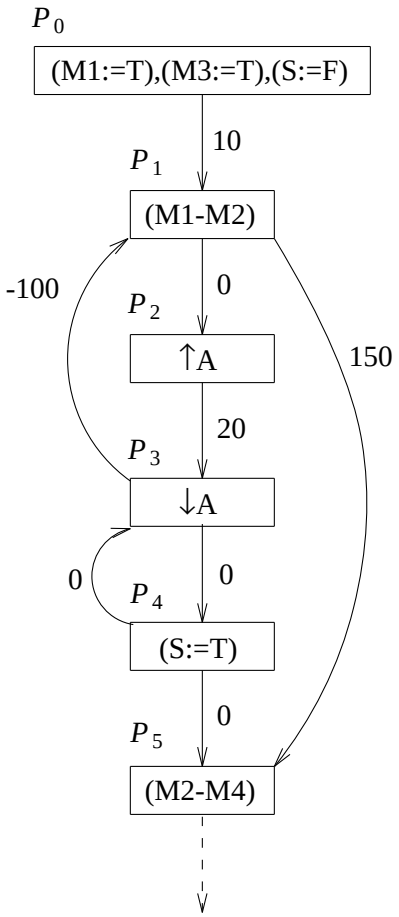


FIGURE 4.

Theorem 1: The weight of a cycle in a separation graph is non-positive.

Proof: Suppose A is a point in a separation graph such that there is cycle of n edges through the points $B_1, \cdots, B_{n-1}$ back to A. The following inequalities correspond to the edges in the cycle with I_1, ..., I_n as the weights on the edges:

$$A + I_1 \le B_1$$
$$B_1 + I_2 \le B_2$$
$$\cdots$$
$$B_{n-1} + I_n \le A$$

where each I_i is a natural number. We obtain the following inequality by adding the two sides of

the above inequalities:

$$A + I \leq A$$

where $I = I_1 + I_2 + \cdots + I_n$. Clearly, I must be non-positive for A to be a point in a computation. Hence, the sum of weights on the transitions forming the cycle is non-positive. □

Definition: The *distance* from point A to point B in a separation graph, $distance(A,B)$, is defined to be the longest (largest weight) path from A to B. If there is no path, $distance(A,B) = -\infty$.

The distance from A to B is the relative separation of A and B measured from point A. Intuitively, if A precedes B on a path in a computation tree, then $distance(A,B)$ in the separation graph is the earliest relative time B can happen after the occurrence of A; $distance(B,A)$ is the earliest relative time A can happen before the occurrence of B (the longest time B can happen after the occurrence of A). As stated in Theorem 2, if A precedes B $distance(A,B)$ is a non-negative integer and $distance(B,A)$ is a non-positive integer.

Referring again to example 1 and Figures 3 and 4, we will use the above definitions to determine the distance from P_3 to P_5, or the minimum time that must elapse between the completion of action A and the entry of mode M4. The longest path from P_3 to P_5 is the maximum weight of all paths from P_3 to P_5, a non-negative integer. As shown in Figure 4, the longest path between the two points is the path from P_3 to P_1 to P_5. Hence, $distance(P_3,P_5)=50$. The distance from P_2 to P_1 is the largest weight on a path from P_2 to P_1, a non-positive integer. Again, refering to Figure 4, the longest path travels from P_2 to P_3 and then to P_1. Hence, $distance(P_2,P_1)=-80$.

Theorem 2: Given two points A and B in a path in a computation tree such that A precedes B, (a) $distance(A,B) \geq 0$, and (b) $distance(B,A) \leq 0$.

Proof:

(a) It is possible that a negative path from A to B exist in a separation graph. However, $distance(A,B)$ is defined to be the longest path from point A to point B. Since A precedes B, $A \leq B$, there is a path with the weight of at least zero from A to B in a separation graph. Hence, $distance(A,B) \geq 0$.

(b) Suppose the distance from B to A is an integer I. Therefore there is a path from B to A with weight I, so $B + I \leq A$. Furthermore, since A precedes B, we know $A \leq B$ from part (a). Hence, there is a path from B to itself with the weight of I, so

$$B + I \leq B$$

By Theorem 1, each cycle in a separation graph must has a non-positive weight. Hence, $I \leq 0$. □

6. Generating a Computation Graph

In Section 5, the notion of an augmented computation tree for a Modechart specification was introduced. Since a system computation is often infinite, our goal is to construct a finite *computation graph* that represents the computations of the system. In the subsequent section, we will describe how to verify a desired property of the system with respect to the corresponding computation graph. Since a mode is control information and may be *viewed* as a state, it is crucial to discuss why a global state reachability graph is not sufficient for our purposes. One reason concerns the timing constraints which are imposed on the actions and transitions. We use two examples to illustrate several key issues in justifying why a global reachability graph is inadequate here. This section concludes with an algorithm for generating a computation graph from a Modechart specification.

Example 2:

Consider the simple Modechart specification shown in Figure 5(a). It consists of two parallel components. The initial modes for the two components are M1 and M4. The transition from mode M1 to mode M2 must be taken after a delay of 5 and before a deadline of 10 after starting in mode M1. The transition from M2 to M3 must be taken after a delay of 16 time units after entering M2. The transition from M4 to M5 must be taken by a deadline of 20 from starting in mode M4.

Figure 5(b) shows a typical global state graph for this system. Although the enabling conditions are true, not every global ‘state’ is reachable. For instance, while the system is in modes M2 and M4, the transition from M2 to M3 cannot be taken before the transition from M4 to M5 because the time after the minimum delay for transition M2-M3 is later than the deadline for the other transition, M4-M5! Hence, it is necessary to consider the relative time ordering to determine what is reachable. In the computation graph proposed at the end of this section, timing constraints are used to prune the vertices in the graph which are unreachable due to the timing constraints.

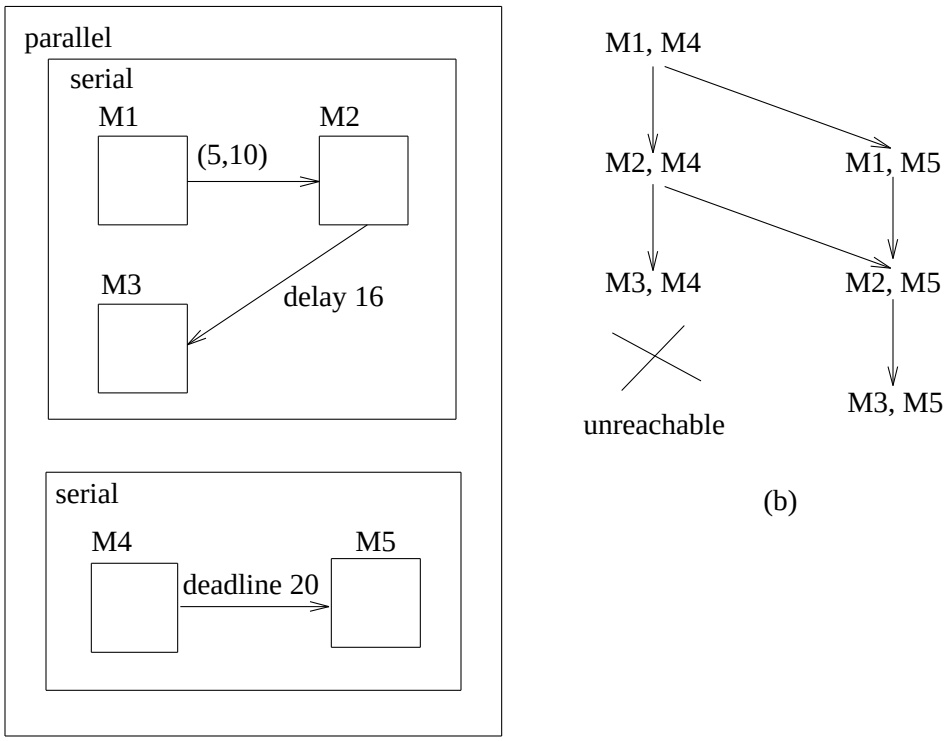


FIGURE 5.

Example 3:

Consider the Modechart specification in Figure 6(a). The system consists of two components with the modes M1 and M4 as the initial modes. The enabling conditions on all the transitions are true. The transitions M1-M2, M2-M3, and M3-M1 have the deadlines of 100, 200, and 1, respectively. The transition from mode M4 to M5 cannot be taken until the delay of 1000 time units after starting in M4. Figure 6(b) shows a reachability graph for this system in which the delays and the deadlines are considered to discard the unreachable vertices in the graph. Observe that the graph in Figure 6(b) shows the mode M5 to be unreachable because the transition M4-M5 is never taken. However, if the system goes from M1 to M2 to M3 and back to M1 several times, the delay of 1000 will eventually expire and the transition from M4 to M5 can be taken. Hence, the graph in Figure 6(b) does not represent the computations of the system. It fails to distinguish between the first and the second time that the system entered M1 while it remained in M4. In particular, the edge labelled α in Figure 6(b) should go from vertex 'M3,M4' to another instance of vertex 'M1,M4'. Figure 6(c) illustrates the revised graph. Observe that the vertices P_1,P_2 and P_3 have the same label. However, each represents a different instance of mode M3. The transition from M4 to M5 which could not have been taken from the vertices P_1 and P_2 can in

fact be taken from P_3.

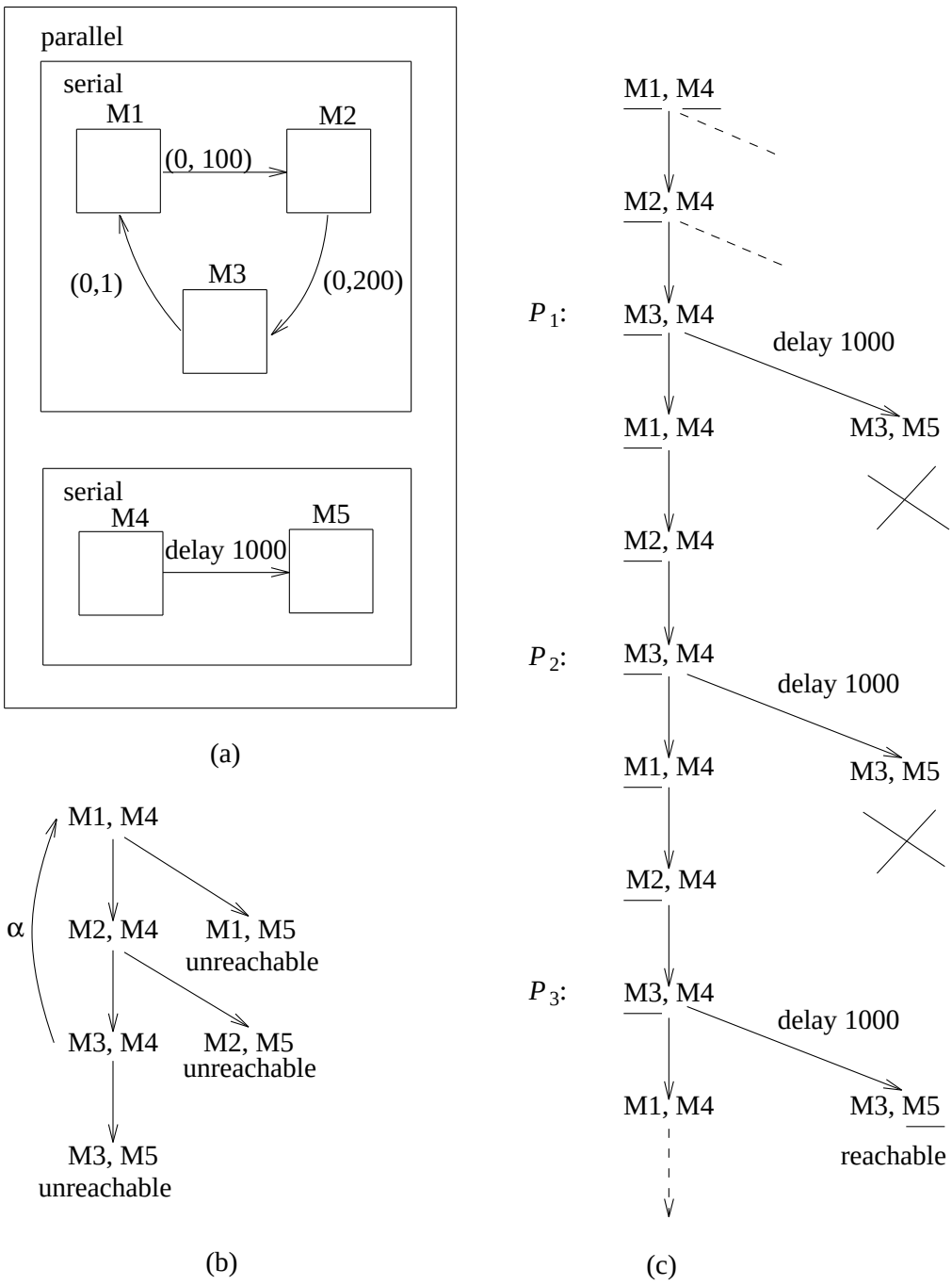


FIGURE 6.

Having looked at some of the key issues, we can now discuss how to construct a computation graph from a Modechart specification. Since a computation graph is intended to represent the computations of a given system, its construction is similar to that of a computation tree.

However, as a new point (vertex) is generated, it can be shown under certain conditions that the new point P_n is an *instance* of a point P_m already in the graph; that is, the computation subtree below P_n is identical to the subtree below P_m. Hence, the point P_n is not added to the graph. Instead, an edge from the immediate predecessor of P_n to P_m is added to denote the occurrence of another instance of P_m in the computations of the system. As we shall illustrate shortly, the resulting computation graph is finite, but each point in the graph may correspond to infinitely many occurrences of events.

As in a computation tree, the points in a computation graph are labelled with events, and the edges represent causality. Furthermore, for convenience, each point is labelled with a set of predicates denoting the modes the system is in and the values of the state variables. Informally, an edge in a computation graph corresponds to a mode transition or a change in the value of a state variable as prescribed by the corresponding Modechart specification. After each *point* the system remains in the same modes and the values of the state variables remain the same until the next *point*. We shall refer to the modes and state variables in a point as the *components* of that point. Furthermore, a mode or a state variable is underlined in each point to indicate the component that changed from the previous point.

For example, the computation graph in Figure 7 denotes a system which is initially in modes M1 and M3 (in parallel) and the state variable S is true. The first edge takes the system from mode M1 to M2. The vertex labelled P_1 denotes a point at which a mode transition from M1 to M2 is taken while the system is also in mode M3 and state variable S is true. The edge from P_1 to P_2 denotes a change in the value of the state variable S. The vertex labelled P_2 represents a point at which the value of S changed from T to F while the system is in modes M2 and M3.

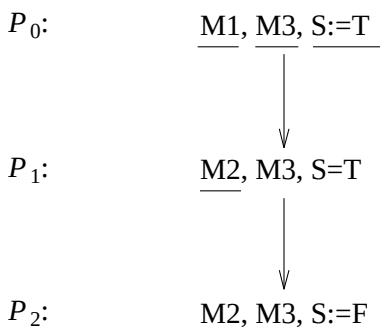


FIGURE 7.

We now present the conditions under which a point can be shown to be an instance of another point in the graph.

Definition: The *event set* of a point P is the set of events in P or its predecessors such that the events are assigned the same time value in at least one computation of the system:

$$E = \{e \mid e \in P \vee [\exists Q \text{ a predecessor of } P \wedge e \in Q \wedge \mathit{distance}(Q,P) = 0\ ]\ \}$$

Definition: The *event set reference points* for a point P are all of those points Q up to and including P such that $\mathit{distance}(Q,P)=0$.

For example, in Figure 4 the event set of point P_5 is {(M2-M4), (M4:=T), (M2:=F), (S:=T), ↓A} and the event set reference points are P_3, P_4, and P_5.

Definition: A *reference point for a mode component M* of point P is the point in the computation tree (or graph) closest to P such that $(M:=T)$ is part of the label of that point, and for no points between that point (exclusive) and P (inclusive).

Notice that by this definition, if component M changes at P, then the reference point for M at P is P itself, otherwise, it is the most recent predecessor of P at which M changed. For example, in Figure 7, the reference point for M2 is P_1 at both points P_1 and P_2.

Definition: The *reference points for a point P* are all of the reference points for the mode components of the point, together with its event set reference points.

Definition: A *potential successor* of a point P is a point that differs from P in at most one component. If that component is a mode component, then there must be a mode transition from the mode of P to that of the potential successor. If that component is a state variable component, then there must be an action that can be executed in one of the modes of P that can result in the state variable changing its value from the value it has in P to that in the potential successor.

Definition: An *actual successor* of a point P is a potential successor that appears in the computation tree as a successor of P.

Note that a potential successor of a point is a point that could be reached if timing requirements were ignored, while actual successors are those that do not violate the real-time constraints.

Definition: Suppose P_m and $P_{m'}$ are two points in a computation graph sharing the same components and event sets. Let P be a reference point for P_m and P' be the corresponding reference point for P'_m. Also, suppose Λ is the set of potential successors of P_m and Λ' is the set of potential successors of P'_m, (Λ and Λ' are isomorphic since P_m and P'_m have the same mode

components and event sets.) The *distance equivalence (deq) condition* for reference point P at P_m and P' at P'_m holds if, for each pair of corresponding potential successors $A \in \Lambda$ and $A' \in \Lambda'$:

(a)

$$distance(P,A) = distance(P',A')$$
$$distance(A,P) = distance(A',P')$$

OR the following conditions hold:

(b) If for the reference points P and P' corresponding to a mode component M, $d = \infty$, then

$$distance(P,A) \geq r$$
$$distance(P',A') \geq r$$

where d is the smallest deadline on any lower/upper bound transition from M, and r is the largest delay on any lower/upper bound transition from M.

AND

(c) If for the reference points P and P' corresponding to a mode component M, $d \neq \infty$, then

$$distance(P,A) > d$$
$$distance(P',A') > d$$

where d is the smallest deadline on any lower/upper bound transition from M.

AND

(d) If the reference points P and P' correspond to an event from the event set,

$$distance(P,A) > 0$$
$$distance(P',A') > 0$$

Lemma 3 : Given a point P, with potential successors $A_1, A_2, \ldots A_n$, A_i is an actual successor of P if and only if $\forall j$ $distance(A_j, A_i) \leq 0$. □

Lemma 4: Suppose P_m and P'_m are two points in a computation graph sharing the same components and the same event sets. Furthermore, suppose the *distance equivalence condition* holds for the corresponding reference points at P_m and P'_m.

(i) A potential successor A of P_m is an actual successor iff the corresponding potential successor A' of P'_m is an actual successor.

(ii) For each actual successor A of P_m and the corresponding actual successor A' of P'_m, the *deq* condition holds for each reference point at A and A'. □

---

* Proofs of lemmas omitted due to space limitations.

Theorem 5: Given the two points P_m and P'_m in the previous lemma, the subtree generated from point P_m is isomorphic to the subtree generated from point P'_m.

Proof: The theorem can be directly proved from the previous lemma by induction on the level of the subtree generated. □

Having shown the conditions under which two points are instances of each other, we now present an algorithm for constructing a computation graph equivalent to the computation tree.

ALGORITHM: (Computation Graph Construction)

1. Construct P_0, the initial point in the computation graph, denoting the modes that the system is in initially and the values of the state variables. Designate P_0 as an unexpanded point.

2. Choose an unexpanded point P_m, let P_l be the immediate predecessor of P_m if one exists.

3. If there is a potential successor P_r of the point P_l such that $distance(P_r, P_m) > 0$, discard P_m.

4. Otherwise, if there is another point P_n in the graph such that P_n and P_m have the same *components* and the same *event sets*, and the *distance equivalence condition* holds for the reference points at P_n and P_m,
   (a) then conclude P_m is another instance of P_n, add an edge from P_l to P_n, and discard P_m.
   (b) Otherwise, let $\tau_1, \tau_2, \cdots, \tau_k$ be the set of transitions that can be taken from P_m with potential successors $Q_1, Q_2, \cdots, Q_k$. Add each Q_i to the graph, mark it as an unexpanded point and add an edge from the point P_m to Q_i. Mark P_m as an expanded point.

5. Repeat steps 2 through 4 until there are no unexpanded points left.

Theorem 6: The computation graph is finite.

Proof : Partition the points of the computation graph by their mode components and event sets. For each mode, refine the partition as follows. If the smallest deadline on any transition from the mode is ∞, then the refinement partitions the points into classes 0 through r, where r is the largest delay on a transition from the mode, according to the value of $distance(P', P)$, where P' is the reference point for P for the given mode component, with any value over r in the class of r, because they all satisfy the deq condition for the component of P'. If the smallest deadline on any transition from P' is finite, say d, then for each potential successor A of P, choose the classes to be 0 through $d+1$, and allocate the points by the value of $distance(P', A)$. Further

refine the partition by the values of *distance*(A,P') for each A and P'. If the deadline on A is infinite, then *distance*(A,P') is $-\infty$ because that edge must appear on a path from A to P'. Alternatively, if the deadline on A is finite, then the *distance*(A,P') is bounded by the sum of the two deadlines. Therefore, this refinement also introduces only a finite number of equivalence classes.

The above partition divides the point space into a finite number of equivalence classes, and within each class, the points pairwise satisfy the requirements of Theorem 5. The computation graph will contain at most one copy of a point from each equivalence class. Therefore the graph is finite. □

7. Example: Railroad Crossing

In this section, a Modechart specification for a system which monitors and controls a railroad crossing is presented. A safety assertion for the system is proved using the approach presented here. This example is based on one that appears in [Leveson & Stolzy 85].

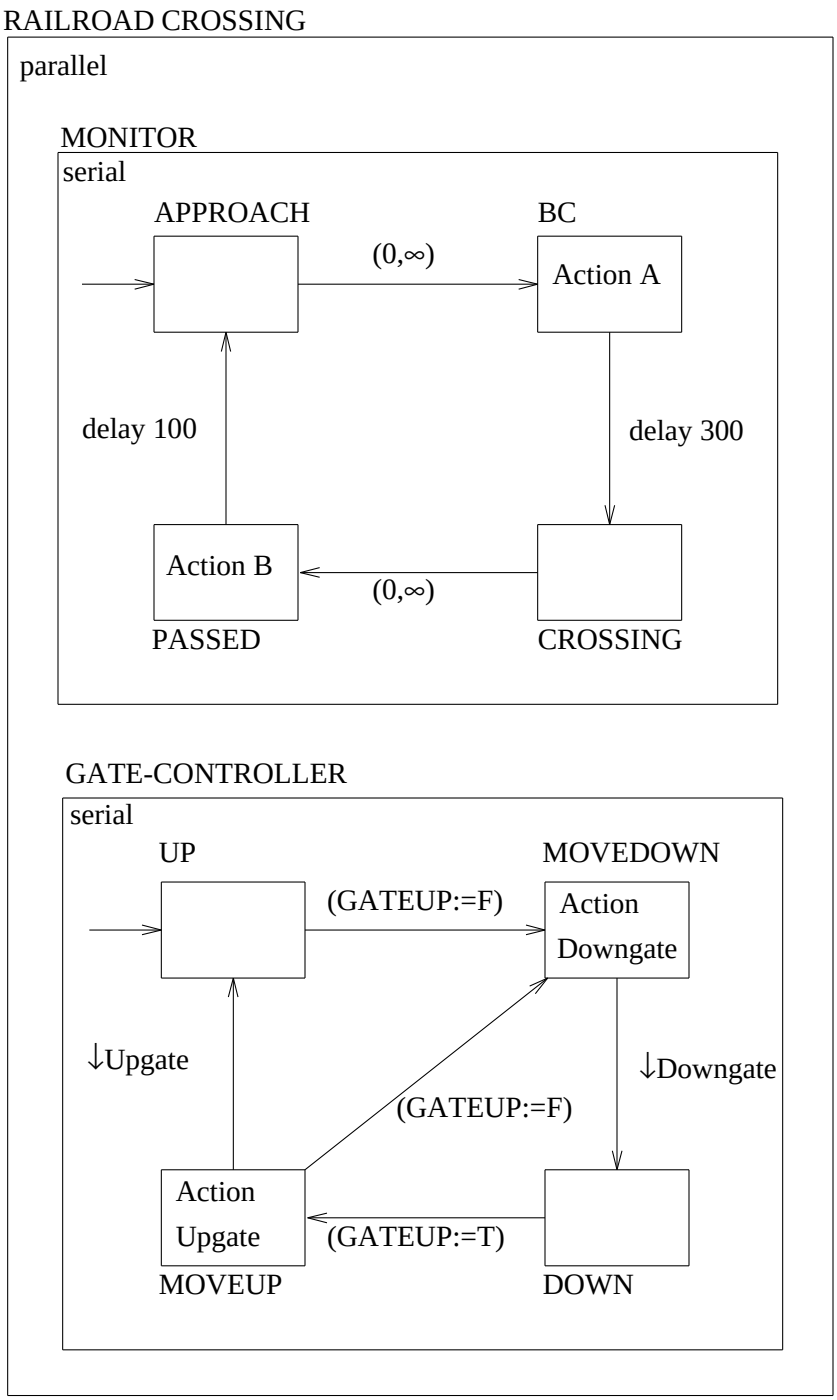
7.1. Specification

Figure 8 illustrates a Modechart specification for a railroad crossing. The system consists of two parallel components: a process monitoring the oncoming trains and a controller interface to the guard gate. The monitoring process is in one of four modes, depending on its view of the train:

APPROACH:	Train is approaching, but very far from, crossing.
BC:	Train is less than 1/2 mile from crossing. Monitor sends signal to controller to lower gate.
CROSSING:	Train is within the railroad crossing.
PASSED:	Train has cleared crossing, and monitor sends signal to controller to raise gate.

The controller process for the gate can be in one of the following modes:

UP:	Gate is up.
MOVEDOWN:	Signal is received to lower gate, and action is initiated to lower gate.
DOWN:	Gate is down.
MOVEUP:	Signal is received to raise gate, and action is initiated to raise gate.



Action A: sets (GATEUP:=F), deadline = 50

Action B: sets (GATEUP:=T), deadline = 50

Action DOWNGATE: set light to red move gate down
deadline = 50

Action UPGATE: move gate up; set light to green
deadline = ∞

Initially, (GATEUP:=T)

FIGURE 8.

Initially, the monitoring process is in APPROACH mode waiting for a train and the gate is UP. As a train reaches the 1/2 mile mark, the system moves from APPROACH to BC (before crossing) mode. Observe that there is no lower/upper bound on when this transition is taken. After entering BC mode, the monitoring process initiates an action to send a signal to the gate controller to lower the gate. The delay on the transition from BC to CROSSING denotes the minimum time that it will take for the train to reach the railroad crossing. There is no upper bound on the transition. After the train passes, the system moves from CROSSING to PASSED. Again, the train can take its time and there is no timing constraint imposed on that transition. After entering PASSED mode, the monitoring process initiates an action to send a signal to the gate controller to raise the gate. Finally, the system can move from PASSED to APPROACH after the delay of 100 to reflect that another train may be approaching. The timing constraint on this transition indicates that there is a minimum separation between a train that just passed and the next oncoming train.

The controller process for the gate is initially in UP mode. When a signal is received from the monitoring process to lower the gate (state variable GATEUP is set to F), a transition to mode MOVEDOWN is taken. While in MOVEDOWN, the action DOWNGATE is performed to lower the gate. Upon completion of the action, the transition to DOWN mode is taken. When a signal is received from the monitoring process to raise the gate (GATEUP is set to T), a transition to mode MOVEUP is taken. While in MOVEUP, the action UPGATE is done to move the gate up. Upon completion of the action, the transition to UP mode is taken. While executing the action Upgate in mode MOVEUP, if another signal is received to lower the gate, the action is aborted and a transition to mode MOVEDOWN is taken.

7.2. Safety assertion

In the above example, the system is unsafe if a train can cross while the gate is up. Hence, a safety assertion for the above system requires that the gate must be down while a train is crossing:

$$\forall i \exists j \;\; @((\text{DOWN}:=T),j) \le @((\text{CROSSING}:=T),i) \;\wedge$$
$$@((\text{CROSSING}:=F),i) \le @((\text{DOWN}:=F),j)$$

7.3. Verification

Figure 9 illustrates the computation graph for the railroad crossing system. The unreachable points (vertices) are shown to be unreachable by applying Lemma 3. One edge in the graph is of particular interest because it illustrates how the deq condition and Theorem 5 are used. Con-

sider the edge labelled α in the Figure 9. It is a backward edge to vertex P_i. Observe that it was not necessary to create the point P'_i because P'_i is an instance of P_i by Theorem 5. The mode components of P_i and P'_i are the same, and the event set for each point is {(GATEUP:=F)}. Furthermore, the corresponding reference points at P_i and P'_i satisfy the deq condition. Therefore, Theorem 5 guarantees that the subtree below P_i is isomorphic to the subtree below P'_i. A similar argument produces the edge labelled β.

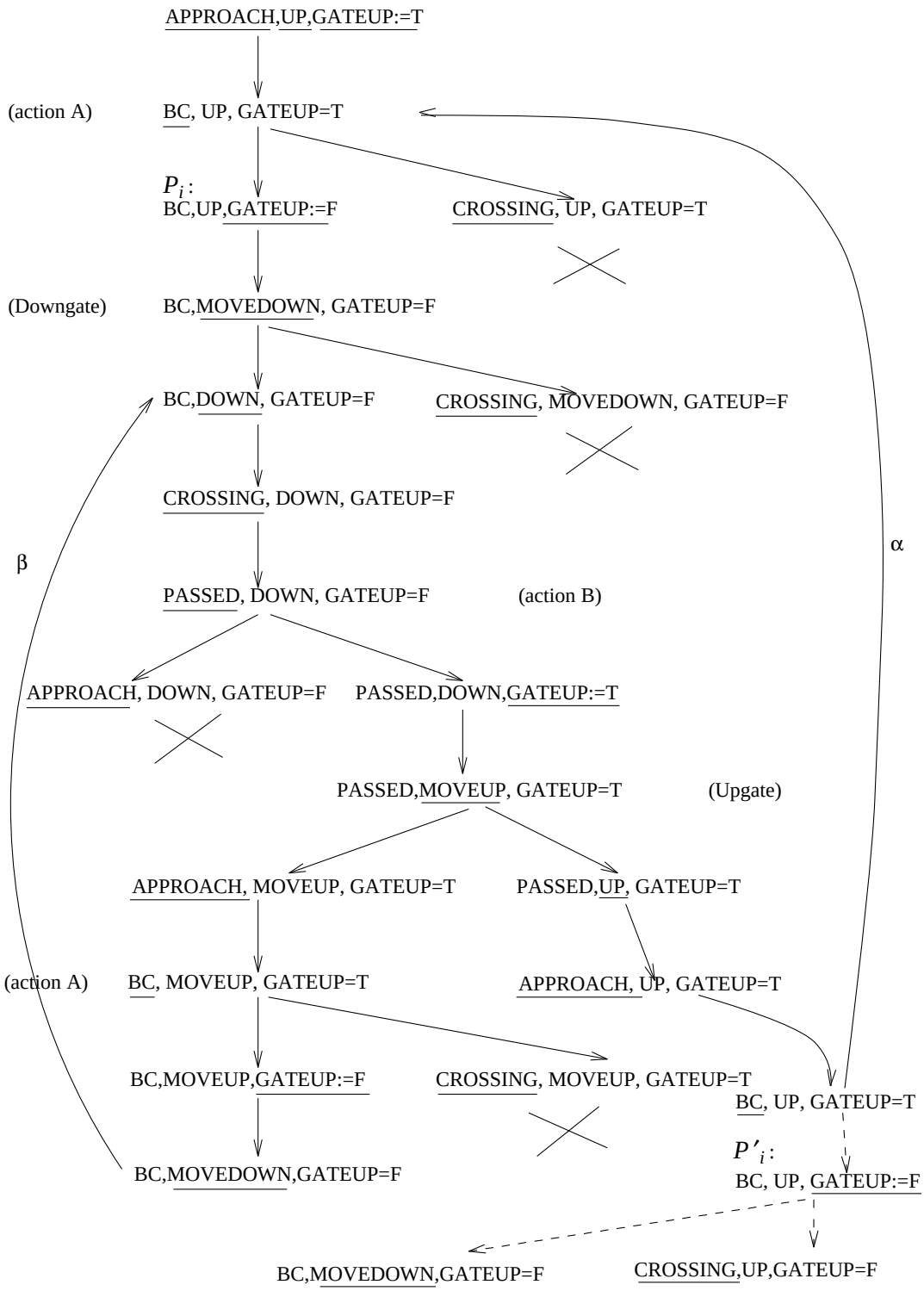


FIGURE 9.

An examination of the graph shows how the safety property holds for the system. The point at which the system enters the mode CROSSING is preceded by a point at which the system

enters the DOWN mode. Furthermore, it remains in DOWN mode even after the transition from the mode CROSSING to PASSED is taken.

8. Decidable Classes of Timing Properties

In the preceding section, we described the construction of a computation graph which represents the computations of a system specified in Modechart. The objective of this section is to introduce two classes of timing properties expressed in RTL. For each class, we shall present decision procedures for determining whether a given computation graph satisfies a property in the class.

8.1. Preliminary Definitions

The RTL subclasses to be described later in this section include properties which can be expressed by RTL formulas containing literals of the form:

$$e_1 \pm I \leq e_2$$

where e_1 and e_2 are occurrence function applications, and I is an integer constant. The classes are surprisingly expressive. Class 1 includes properties such as minimum and maximum separation of events, actions, and modes. Class 2 includes mutual exclusion of actions, modes, and intervals. The two subclasses and the corresponding verification procedures are described in terms of intervals and endpoints. Informally, an endpoint is an occurrence function application, i.e., the time of occurrence of an event. An interval is denoted by two related endpoints.

Definition: An *endpoint* is defined to be an application of an occurrence function of the form

$$@(E, i \pm c)$$

where E is an event, i is an integer variable, and c is a non-negative integer.

Definition: Two endpoints are *related* if both contain the same integer variable in the occurrence index of their respective '@' function applications.

For example, $@(E_1, j)$ and $@(E_2, j+2)$ are related endpoints, but $@(\downarrow A, i)$ and $@(\downarrow A, j)$ are not related.

Definition: An *interval* consists of two endpoints of the form

$$@(E_1, i \pm c_2), \text{ and } @(E_2, i \pm c_2)$$

where E_1 and E_2 are events, i is an integer variable, and c_1 and c_2 are non-negative integer constants. Two endpoints denoting an interval are by definition related.

RTL inequalities are employed to describe the relationship between the endpoints of an interval or the relationship between endpoints of distinct intervals. For instance,

@((M:=T),i) and @((M:=T),i+1)

are two endpoints defining an interval, and the inequality @((M:=T),i) + 100 ≤ @((M:=T),i+1) denotes that the successive occurrences of entering mode M must be at least 100 time units apart. Similarly, the RTL formula

$$\forall i \, \forall j \;\; @(\downarrow A,i) \le @(\uparrow B,j) \;\vee\; @(\downarrow B,j) \le @(\uparrow A,i)$$

expresses the mutual exclusion relationship between the interval denoted by @(↑A,i) and @(↓A,i) and the interval denoted by @(↑B,j) and @(↓B,j), i.e., mutual exclusion of actions A and B.

The two classes of timing properties to be described shortly are defined in terms of endpoints and intervals. Suppose we are given a computation graph and two endpoints @(E_1,i) and @(E_2,i) whose relationship is to be verified with respect to the graph. If each cycle in the graph is labeled with the same number of E_1 and E_2 events, then by designating a point in the graph containing E_1 as representing the i th occurrence of E_1, the point representing the i th occurrence of E_2 follows in (or precedes by) a constant number of cycles of the graph. The number of iterations must reflect both the number of additional occurrences of E_1 before the cycle is entered, as well as the difference in the constants. However, if a cycle is labeled with different number of E_1 and E_2 events, then the number of cycles separating the i th occurrences of E_1 and E_2 is a function of i , rather than a constant.

Definition: Suppose G is a computation graph, @(E_1,$i \pm c_1$) and @(E_2,$i \pm c_2$) are related endpoints. A cycle in G is said to *preserve* the endpoints if the number of vertices in the cycle labeled with E_1 is the same as the number of vertices labeled with E_2.

Definition: A computation graph is said to *preserve* an RTL formula if each cycle in the graph preserves each set of related endpoints in the formula.

In the remainder of this paper, we are interested in the RTL formulas which are preserved by a given computation graph. This does not appear to be a very restrictive condition. It is trivially true that when E_1 and E_2 are the same in the above definition, a computation graph *preserves* the endpoints. Furthermore, when E_1 and E_2 are the entry and exit events for the same mode, the related endpoints containing E_1 and E_2 are preserved by a computation graph of a Modechart specification. The same holds when E_1 and E_2 denote transition events for a state variable, or when they are the start and stop events for an action. Hence, for the above cases, it is trivial to check if each set of related endpoints in an RTL formula are preserved by a computation graph. However, when two related endpoints do not fit in the above cases, the check for the condition

can be performed in polynomial time. The remaining subsections introduce two subclasses of timing properties expressed in RTL for which one can decide if a computation graph satisfies the property.

8.2. Class 1: Min/Max Separation of Related Endpoints

In this subsection we are concerned with a class of timing properties which specify the relative and absolute ordering of related endpoints. The class consists of the RTL formulas of the form Q F where Q is either $\forall i$ or $\exists i$, and F is a quantifier free formula such that each inequality in F is of the form

$$e_1 \pm I \leq e_2$$

where e_1 and e_2 are related endpoints, and I is a natural number. Observe that the endpoints in F are related because the same variable (plus/minus an integer offset) appears as the occurrence index in each endpoint. For example, the following formula states that two successive entries to the mode M are either 10 time units apart or within 5 time units of each other:

$\forall$ i @((M:=T),i)+10 ≤ @((M:=T),i+1) $\vee$ @((M:=T),i+1) ≤ @((M:=T),i)+5

As a different example, consider a formula specifying that an instance of action A is performed once each time the system is in mode M:

$$\forall i \ @((M:=T),i) \leq @(\uparrow A,i) \ \wedge \ @(\downarrow A,i) \leq @((M:=F),i)$$

Each inequality in a formula of class 1 can be rewritten as one of two forms:

$$e_1 - e_2 \leq I \qquad (1)$$

$$I \leq e_2 - e_1 \qquad (2)$$

The first inequality specifies a maximum separation of I on the two endpoints, and the second inequality imposes a minimum separation of I on the endpoints.

Suppose G is a computation graph and F is a class 1 RTL formula which is preserved by G. The objective is to determine if every computation in the graph G satisfies the timing property expressed in the formula F. Since F has one quantifier, all the endpoints in the formula form one set S of related endpoints which are preserved by every cycle in G:

$$S = \{ @(E_1,i+c_1), @(E_2,i+c_2), \ldots , @(E_n,i+c_n) \}$$

where i is the occurrence index variable quantified by a $\forall$ or $\exists$, and n is the number of endpoints in F. As mentioned in subsection 8.1, if we take a point P in G labeled with an event E_j (for some 1≤j≤n) to mean the ith occurrence of E_j, then all the corresponding end points in F (if any exist) can be located by unrolling each cycle in G a constant number of times. For each inequal-

ity in F, if it is of form (1), we find the maximum distance between the two points in the unrolled graph and assign a truth value to the inequality. Otherwise, the inequality is of form (2). Hence, we find the minimum distance between the two points in the unrolled graph and assign a truth value to the inequality. In this manner, a truth value can be obtained for F. However, this truth value applies to all of the occurrences (perhaps infinitely many) corresponding to the point P in the graph.

If the formula F is quantified by a $\forall$, then the above procedure must be repeated for each point in the graph G which is labeled with the event E_j. A *true* assignment to the formula F in each case is necessary to conclude that every computation in G satisfies F. However, if F is quantified by a $\exists$, then it is necessary to show that F is assigned *true* for at least one application of the procedure to a point labeled with the event E_j.

8.3. Class 2: Exclusion/Inclusion of Interval and Endpoints

In this subsection we are concerned with a class of timing properties describing exclusion/inclusion of an endpoint or an interval with respect to another interval. The timing properties in this class can be expressed as RTL inequalities relating the endpoints of an interval to the endpoints of another interval. Hence, the RTL formulas specifying the properties in this class have two quantifiers, one for each interval. Recall that an endpoint was defined in subsection 8.1. to be of the form $@(E,i \pm c)$. We extend that definition to include an integer offset I: $@(E,i \pm c) \pm I$

The properties in this class can be in two general forms. Suppose e_1 and e_2 are the endpoints of an interval α and e_3 and e_4 are the endpoints of a second interval β. The first form specifies the inclusion of one interval inside the other:

$$Q \;\; e_1 \le e_3 \;\wedge\; e_4 \le e_2 \qquad (1)$$

where Q is a prefix of two quantifiers on the occurrence index variables for the intervals. For instance, the following RTL formula states that each stay in mode M contains some execution of action A which starts at least 100 time units after entering the mode:

$$\forall \text{i} \exists \text{j} \;\; @((\text{M:=T}),\text{i}) + 100 \le @(\uparrow\text{A},\text{j}) \;\wedge\; @(\downarrow\text{A},\text{j}) \le @((\text{M:=F}),\text{i})$$

The second form specifies the exclusion of one interval from the other:

$$Q \;\; e_4 \le e_1 \;\vee\; e_2 \le e_3 \qquad (2)$$

For instance, the following formula specifies the mutual exclusion of two actions A and B:

$$\forall i \forall j \;\; @(\downarrow B,j) \le @(\uparrow A,i) \;\vee\; @(\downarrow A,i) \le @(\uparrow B,j)$$

Given a computation graph G and a RTL formula F specifying a timing property in class 2, the objective is to determine if every computation in G satisfies F. We identify five cases for the inclusion property shown in form (1). In each of the cases below, i is the occurrence index variable for the endpoints e_1 and e_2 (interval α), and j is the variable for the endpoints e_3 and e_4 (interval β).

(1a) $\forall i \exists j \;\; e_3 \le e_1 \wedge e_2 \le e_4$

This case specifies that *every* instance of interval α is contained in *some* instance of interval β. The above formula is verified by first locating an appearance of interval α in G, then checking if an instance interval β surrounds it. Observe that only a finite number of instances of β needs to be checked because there can be only a finite number of pairs of points in the graph labeled with the events in e_3 and e_4 such that it encloses the points corresponding to each appearance of α. The above procedure must be repeated for each appearance of interval α in graph G.

(1b) $\forall i \exists j \;\; e_1 \le e_3 \wedge e_4 \le e_2$

(1c) $\exists i \exists j \;\; e_1 \le e_3 \wedge e_4 \le e_2$

(1d) $\exists i \forall j \;\; e_1 \le e_3 \wedge e_4 \le e_2$

(1e) $\exists i \forall j \;\; e_3 \le e_1 \wedge e_2 \le e_4$

Similarly, we identify four cases for the exclusion property shown in form (2). The occurrence index variable for the endpoints e_1 and e_2 (interval α) is i, and j for the endpoints e_3 and e_4 (interval β).

(2a) $\forall i \forall j \;\; e_4 \le e_1 \vee e_2 \le e_3$

This case specifies that *every* instance of interval α is mutually exclusive with *every* instance of β.

(2b) $\exists i \exists j \;\; e_4 \le e_1 \vee e_2 \le e_3$

(2c) $\forall i \exists j \;\; e_4 \le e_1 \vee e_2 \le e_3$

(2d) $\exists i \forall j \;\; e_4 \le e_1 \vee e_2 \le e_3$

The two classes of timing properties above obviously do not include all the properties which one may wish to specify about a Modechart system. For example, the following properties cannot be expressed in either of the above classes:

- Each instance of an interval α contains an instance of interval β or an instance of interval γ.

- The three intervals α, β, and γ each contain the next in the list.

For both properties, alternative decision procedures can be developed to verify the properties with respect to a graph. An area of continuing research is the discovery of other decidable classes of timing properties.

9. Concluding Remarks

Real Time Logic (RTL) is a logic for the specification and analysis of hard real-time systems that was developed as part of the SARTOR project, which is an attempt to create a design environment for the automated development of hard real time systems. As a consequence, it was desired to produce a decision procedure for RTL to aid in this process. Unfortunately, it was proved that RTL in full is undecidable [Jahanian et al 88a].

The primary difficulty in deciding the satisfiability (or unsatisfiability) of an RTL formula is the potentially infinite number of event occurrences that must be considered. This problem arises due to the use of both arbitrary quantification and the uninterpreted functions in RTL formulas. The occurrence function does not pose a problem in obtaining decidability, because an occurrence function cannot be an argument to an uninterpreted function. However, skolemizing an existentially quantified variable may yield an uninterpreted function which may be an argument to other uninterpreted functions.

Even after using serial and parallel modes in specifying a system, the corresponding RTL formulas involve arbitrary $\forall$ and
$\exists$ for variables denoting event occurrences. In order to take advantage of the hierarchical and orthogonal control information that is provided by a Modechart specification, we must avoid verifying a given property against the entire set of RTL formulas. This paper has shown how a system property specified in RTL can be verified with respect to a Modechart specification without explicitly examining all possible occurrences of events in the system.

The verification technique proposed in this paper requires the construction of a computation graph for a given Modechart specification. System properties are proved using the computation graph and the corresponding RTL formulas. Although timing constraints are used to discard unreachable vertices, constructing the entire computation graph suffers from the same drawbacks as similar techniques based on reachability analysis. In particular, it is impractical to generate the entire graph for a complex Modechart specification. A natural extension of the proposed approach is to investigate the verification of properties by constructing a partial graph.

continued ...

Bibliography

[Harel 86]. D. Harel, ‘‘Statecharts: A Visual Formalism for Complex Systems,’’ The Weizmann Institute of Sci. Tech. Report, Israel (July 1986). (also in Science of Programming 8, 1987)

[Jahanian & Mok 86]. F. Jahanian and A.K. Mok, ‘‘Safety Analysis of Timing Properties in Real-Time Systems,’’ *IEEE Transactions on Software Engineering* **SE-12**(9) pp. 890-904 (Sept. 1986).

[Jahanian & Mok 88]. F. Jahanian and A.K. Mok, ‘‘Modechart: A Specification Language for Real-Time Systems,’’ *to appear in IEEE Transactions on Software Engineering*, (1988).

[Jahanian et al 88a]. F. Jahanian, A. Mok, and D. Stuart, ‘‘Formal Specification of Real-Time Systems,’’ Department of Computer Sciences, TR-88-25, University of Texas at Austin (June 1988).

[Jahanian et al 88b]. F. Jahanian, R.S. Lee, and A.K. Mok, ‘‘Semantics of Modechart in Real Time Logic,’’ *Proc. 21st Hawaii International Conference on System Sciences*, (Jan. 1988.).

[Lamport 83]. L. Lamport, ‘‘What Good is Temporal Logic?,’’ pp. 657-668 in *Proceeding of IFIP, Information Processing*, ed. R.E.A. Mason,,North-Holland (1983).

[Leveson & Stolzy 85]. N. G. Leveson and J. L. Stolzy, ‘‘Analyzing Safety and Fault Tolerance Using Time Petri Nets,’’ *TAPSOFT: Joint Conference on Theory and Practice of Software Development*, Springer Verlag, (March 1985).

[Merlin & Farber 76]. P.M. Merlin and D.J. Farbar, ‘‘Recoverability of Communication Protocols-Implications of a Theoretical Study,’’ *IEEE Trans. on Communications*, pp. 1036-43 (Sept. 1976).

[Ostroff 87]. J. S. Ostroff, ‘‘Real-time Computer Control of Discrete Event Systems Modelled by Extended State Machines: a Temporal Logic Approach,’’ Ph.D. Dissertation, Department of Electrical Engineering, University of Toronto, Toronto, Canada (January, 1987).

[Ramchandani 74]. C. Ramchandani, ‘‘Analysis of Asynchronous Concurrent Systems by Timed Petri Nets,’’ TR 120, Project MAC, MIT (Feb. 1974).

[Razouk & Phelps 84]. R.R. Razouk and C.V. Phelps, ‘‘Performance Analysis of Timed Petri Nets,’’ *Proc. of 4th International Workshop on Protocol Spec, Verf and Testing*, (June 1984).

[Shankar & Lam 87]. A.U. Shankar and S.S. Lam, ‘‘Time-Dependent Distributed Systems: Proving Safety, Liveness and Real-Time Properties,’’ *Distributed Computing*, (2) pp. 61-79 Springer-Verlag, (1987).

[Zuberek 1980]. W.M. Zuberek, ‘‘Timed Petri Net and Preliminary Performance Evaluation,’’ *7th Annual Symposium on Computer Architecture*, pp. 88-96 (1980).

continued ...