A Formal Method for Verifying Real-Time Properties of Modechart Specifications‡

Douglas A. Stuart†

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

Farnam Jahanian

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

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

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. Three classes of properties are presented for which decision procedures are developed.

1. Introduction

With the increasing reliance on digital computers in embedded systems, the need for dependable systems that deliver correct results in a timely manner has become more crucial. Large-scale embedded systems are being built in diverse applications such as avionics, air traffic control, manufacturing, and patient monitoring. As real-time applications become more sophisticated, the software design and development process has become

---

† Supported in part by a research grant from the Office of Naval Research under ONR contract number N00014-89-J-1472

‡ This is a substantially revised version of an earlier work originally published as [Jahanian & Stuart 88] .

increasingly more complex. The increasing complexity of software control of time-critical functions requires a means for the precise specification of their behavior and formal methods for analyzing system requirements.

Modechart is a graphical specification language developed to provide a compact and structured way to represent real-time systems [Jahanian et al 88b]. 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 86a]. The goal of this research is to use the structuring of Modechart to produce a decision procedure for classes of properties that are 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* capturing the system behavior. The computation graph is used to validate properties of a system with respect to its Modechart specification.

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 computation does not include the notion of state, there is no concept of an invariant holding in all states, as in most of the real-time verification literature (e.g., [Lamport 83] , [Ostroff 87] , [Henzinger et al 90] , [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 86a]. 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 Modechart and RTL. Then in sections 4 through 7 computation graphs will be introduced. Section 8 presents an example illustrating these concepts. Lastly, in section 9 three classes of RTL formulas will be introduced and the corresponding decision procedures will be developed. The proofs of most of the theorems and lemmas are in the appendix.

2. 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. Each action also has a deadline on the completion of the action relative to the entry of the mode.

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.

Two classes of events are identified for specifying conditions on transitions that depend on the occurrence of other transitions. 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. The events (→M) and (M→) are the events corresponding to entering and leaving mode M.

The condition for a mode transition is of the form

$$c_1 \lor c_2 \lor \cdots \lor c_n$$

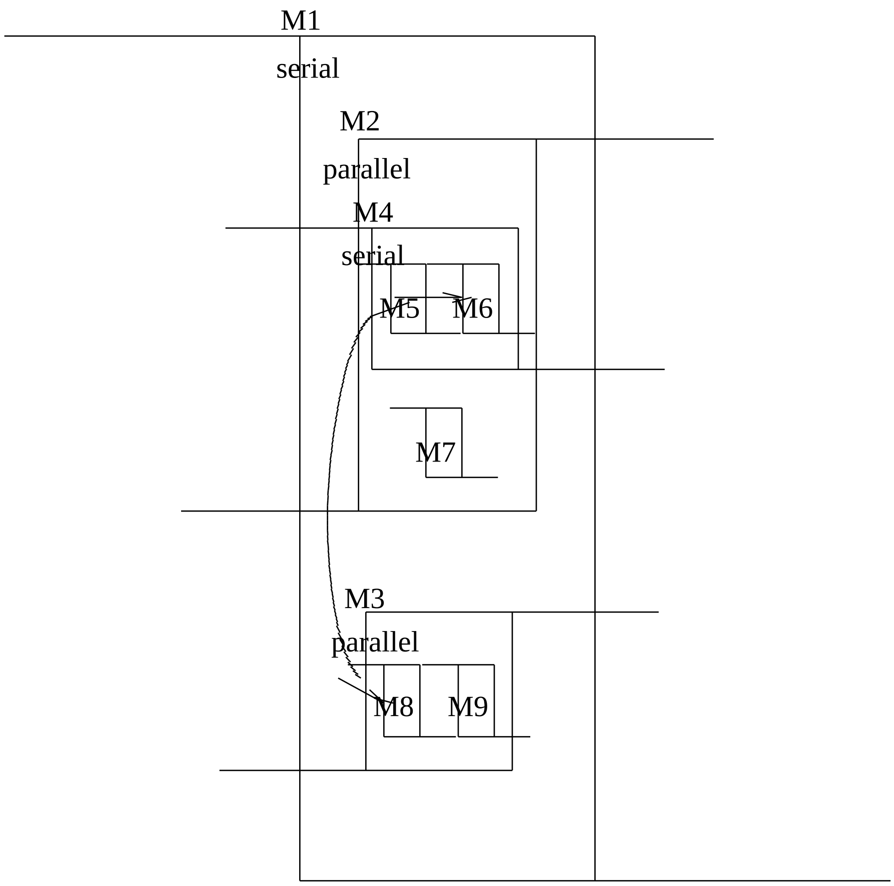

Figure 1

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 and t denotes the time at which the transition is taken.)

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

(b) 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 only if 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 non-negative 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 no sooner than r time units and no later than d time units after 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 the semantics of Modechart, as will be seen in section 5, 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 what follows, each disjunction of a transition condition will be considered as a separate transition.

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 (or the transition must be a self-loop and either the source or destination must be the child of a serial mode, and the other be contained in it). 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 appear in [Jahanian & Mok 88] and [Jahanian et al 88b] .

The examples in this paper are all flat modecharts (where the (top level) modechart is a parallel mode, with a number of serial children, each with primitive children), although the techniques apply to arbitrarty modecharts. Also, consideration of actions will be deferred because they pose special problems.

3. 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 a natural number 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. We also distinguish the mode transition and entry events introduced in the preceding section. 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 of the twelfth instance of event e 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 @.

$$\begin{aligned} i \ \ & @(\Omega\text{CTRL\_D},i) \leq @(\uparrow\text{LOGOUT},i) \ \wedge \\ & @(\downarrow\text{LOGOUT},i) \leq @(\Omega\text{CTRL\_D},i) + 20 \end{aligned}$$

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.

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. We will 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 the most recent occurrence is used for each event in the occurrence function applications. 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. 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.

5. Computations

The set of all computations, or models, of a modechart specification will be represented first as a tree and later as a finite graph. Before considering the set of all computations of a specification some preliminary definitions and theorems about individual computations will be introduced in this section. Each computation will be represented as a timed trace, where the elements of the trace, called *points*, will represent a configuration of the system specified by the modechart, and contain sufficient information to indicate which modes are active and which transitions can be taken, and later, the status of actions and the values of state variables.

Each point has a label, which consists of three parts. The *name* of a point is the list of the modes of the specification that are considered active in the point. The *entry event* is an indication of the event which caused entry of the point (in the trace). This is either the entry event of the root mode of the modechart (in the case of the initial point in a trace) or the transition event for some transition. The *event set* of a point is a subset of the set of *condition events*. The *condition events* of a specification are those events that appear in (or can affect) a triggering condition. The intent is that the events in the event

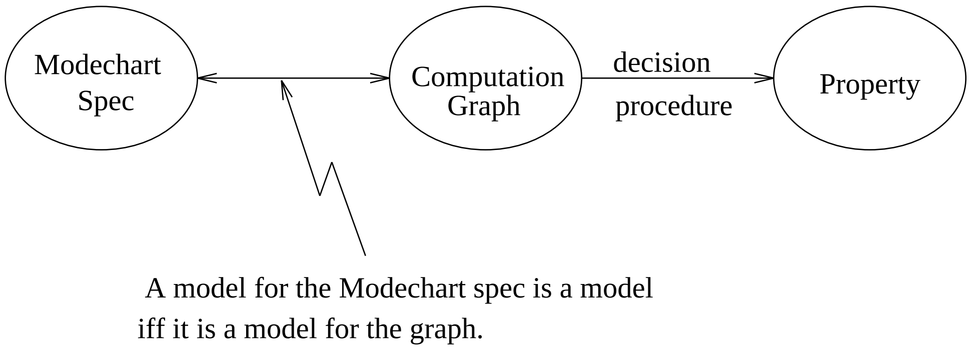

Figure 2

set will be used to determine which transitions governed by triggering conditions can be taken when the system is in the configuration represented by the given point. To that end, the semantics of traces will ensure that the events in the event set of a point happen at the same time as the point, and are the result either of the entry event of the point, or are in the event set of the predecessor point in the trace. Since only condition events can determine which transitions can be taken at any given time, it is sufficient to include in the event set of a point only the condition events.

Having defined the elements of the traces, it is now possible to define the (untimed) traces of a specification. A *trace* of a modechart specification is a sequence $\{\sigma_i\}_{i \geq 0}$ of points of the specification such that σ_0 is the initial point, and for all $i > 0$, the entry event for σ_i is the transition event for some transition which is enabled in point σ_{i-1}, and the event set of σ_i is either the set of condition events associated with the transition, or the union of the set of condition events associated with the transition and the event set of σ_{i-1}. In the second case, σ_i is said to have *inherited* the event set of σ_{i-1}. A transition is *enabled* in a point if its source mode is part of the name of the point, and either it is governed by a lower/upper bound condition or by a triggering condition which is satisfied by the events in the event set of the point. Also the name of σ_i must include all modes either entered by the transition, or in the name of σ_{i-1} and not exited by the transition, and must not include any modes not in the name of σ_{i-1} not entered by the transition, nor in the name of σ_{i-1} and exited and not re-entered by the transition. The initial point is the point that has as its name exactly the initial modes of the specification, as its entry event the entry event of the root mode of the specification, and as its event set the set of those entry events of initial modes that are condition events.

The definition of trace given above does not take into account the timing constraints imposed by lower/upper bound constraints and event set inheritance. These two factors give rise to the notion of a *constrained trace*. A *constrained trace* is a trace where each point is augmented by a *time*, together with a set of constraints on the times of the points. The timing constraints in force on a constrained trace are determined as follows. For all i:

(1)$\text{time}(\sigma_i) \geq 0$.

(2)$\text{time}(\sigma_i) \leq \text{time}(\sigma_{i+1})$.

(3)If σ_{i+1} inherits the event set of σ_i, then $\text{time}(\sigma_{i+1}) = \text{time}(\sigma_i)$.

(4)If σ_{i+1} does not inherit the event set of σ_i, then $\text{time}(\sigma_i) + 1 \leq \text{time}(\sigma_{i+1})$.

(5)If the entry event of σ_{i+1} is the transition event for a transition governed by a triggering condition, then $\text{time}(\sigma_{i+1}) = \text{time}(\sigma_i)$.

(6)If the entry event of σ_{i+1} is the transition event for a transition governed by a timing constraint (r, d), and σ_j is the reference point for the transition at σ_i, then $\text{time}(\sigma_j)+r \le \text{time}(\sigma_i)$ and $\text{time}(\sigma_i)-d \le \text{time}(\sigma_j)$.

(7)If there is a transition governed by a triggering condition enabled in σ_i then $\text{time}(\sigma_{i+1})=\text{time}(\sigma_i)$.

(8)If there is a transition governed by timing constraint with deadline d enabled in σ_i and σ_j is the reference point for the transition at σ_i, then $\text{time}(\sigma_i)-d \le \text{time}(\sigma_j)$.

The constraints that result from (6) and (8) use the concept of reference point. The *reference point* for a transition is the point with respect to which a timing constraint is measured. For transitions governed by triggering conditions, the reference point is the preceding point, since the transition is enabled by the event set in the preceding point, and therefore must happen at the same time as the preceding point. For transitions governed by timing constraints, since timing constraints are measured with respect to source mode entry, the reference point is the most recent point that has an entry event that causes the entry of the source mode of the transition.

The timing constraints that result from (1) and (2) ensure that time does not regress as the trace proceeds. (3) and (4) ensure that the event set of a point does in fact include exactly those condition events that occur at the same time as the point, and as a result of entry events that occur at or before the point in the trace. (5) ensures that transitions governed by triggering conditions are in fact taken at the same time as their triggering events. (6) enforces lower/upper bound conditions. (7) and (8) ensure that no transition can be taken that violates the deadline of some other pending transition. This is necessary because the semantics of Modechart require that deadlines be absolute. The form of the constraints has been chosen to simplify the definitions of separation graphs in the next sections.

One final definition is needed before a computation can be defined. A *completed constrained trace* is a constrained trace that is either infinite, has no enabled transitions in its final point, or has only transitions governed by lower/upper bound conditions with infinite deadlines enabled in its final point. A *computation* is an assignment of times to the points of a completed constrained trace such that no constraint is violated. Computations are only given for completed constrained traces to ensure progress. Otherwise, a computation could end with a pending enabled transition with a finite deadline, which would be violated once time progressed beyond the end of the computation.

Before turning to the set of all computations of a specification and representing it as a tree, two additional definitions and a theorem are necessary. A point is a *potential successor* of a finite trace if the trace obtained by appending the point to the end of the trace is also a trace. The set of potential successors of a trace therefore captures the set of events that could possibly "come next" independent of time. A point is an *actual successor* of a trace if it is a potential successor of the trace and the set of constraints on the constrained trace corresponding to the trace obtained by appending the point to the end of the trace is satisfiable. The set of actual successors of a trace is therefore the set of events that could "come next" without violating any currently pending timing constraints. The relationship between actual successors and computations is the main result of this section and is stated in the following theorem.

Theorem 1: A completed constrained trace has a computation if and only if each point in the trace (other than the initial point) is an actual successor of its predecessor.

6. Computation Trees

Having defined the individual computations of a specification, as a result of Theorem 1 the computations of a system specified in Modechart can be viewed as a directed tree in which the nodes are points, and the children of a point are its actual successors. A trace would then be a path in the tree.

Example 1:

Consider the modechart of Figure 3. Figure 4 is a segment of its computation tree. Each point (other than the initial point) is labelled with its entry event. 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, M3, M5 and M8. The point P_1 corresponds to the occurrence of the event (M1$\rightarrow$M2) which denotes that a mode transition from M1 to M2 is taken at that point. The following lower/upper bound separations are imposed on the points in the path segment P_0 to P_5 by the definition of constrained trace. The inequalities denote bounds on the instances of events corresponding to each point.

$P_0 + 10 \le P_1$	delay on transition M1$\rightarrow$M2
$P_2 \le P_1$	M8$\rightarrow$M9 happens at same time as M1$\rightarrow$M2
$P_2 + 20 \le P_3$	delay on transition M9$\rightarrow$M10
$P_1 + 150 \le P_5$	delay on transition from M2$\rightarrow$M4

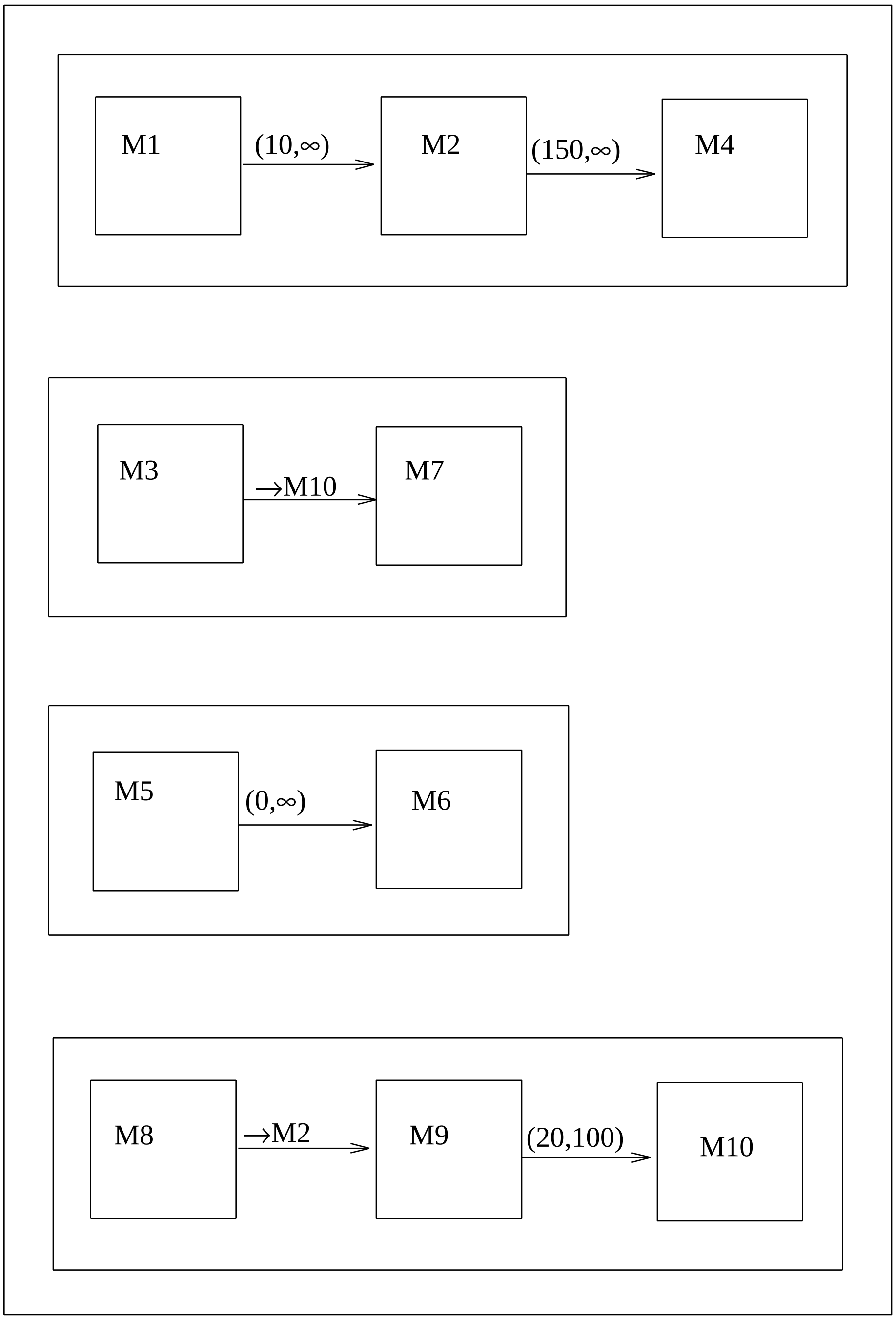

Figure 3

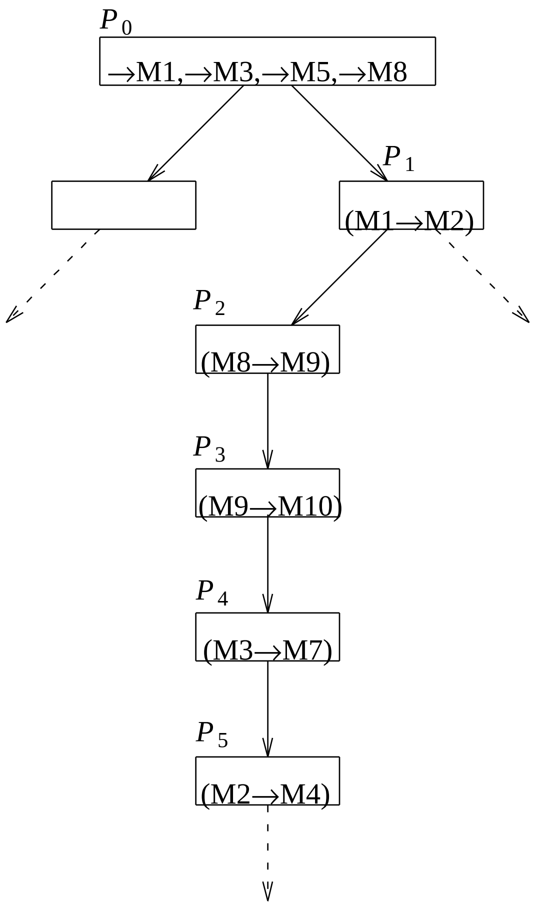

Figure 4

$P_3 \le P_1 + 100$	deadline on transition M9→M10
$P_4 \le P_3$	M3→M7 happens at same time as M9→M10

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. As stated in the previous section, 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 possible prefix of a computation of the system.

The set of timing constraints 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 hence the corresponding trace) and the weights on the edges denote the specified separation between the nodes (and are the constraints of the corresponding constrained trace). For simplicity and to reduce clutter, the constraints due to clauses (7) and (8), those that enforce deadlines of pending transitions, are omitted if the transitions are later taken (since the constraints then become redundant). 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 5 illustrates part of the corresponding separation graph for the path in Example 1. Notice that the edges corresponding to deadlines have non-positive weights.

Theorem 2: The weight of a cycle in a separation graph is non-positive.

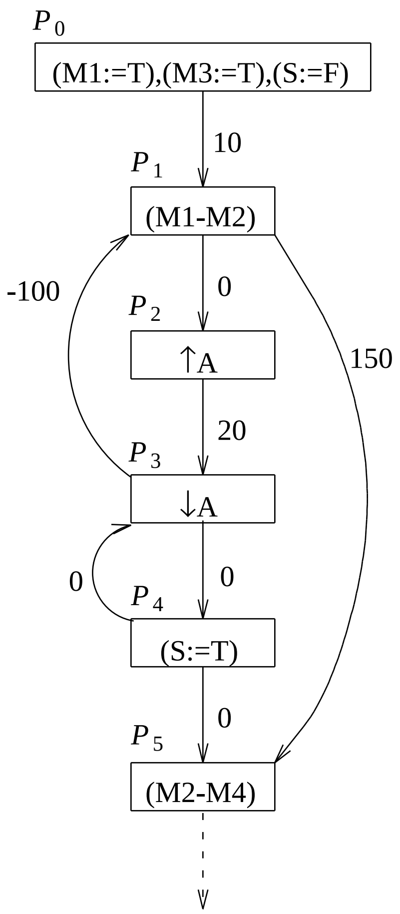

Figure 5

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

The distance from A to B is the earliest time B can happen relative to A. If the distance is negative it is can be viewed as a deadline, and if the distance is positive, it can be viewed as a delay. 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 3, 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 5, 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 entry of mode M10 and the entry of mode M4. The distance from P_3 to P_5 is the maximum weight of all paths from P_3 to P_5, a non-negative integer. As seen in Figure 5, the longest path between the two points is the path from P_3 to P_1 to P_5. Hence, $distance(P_3,P_5)=50$, so at least 50 time units must elapse between entry of M10 and entry of M4. The distance from P_4 to P_2 is the largest weight on a path from P_4 to P_2, a non-positive integer. Again, referring to Figure 5, the longest path travels from P_4 to P_3 to P_1 and then to P_2. Hence, $distance(P_4,P_2)=-100$, and so at most 100 time units may elapse between the entry of M9 and the entry of M7.

Theorem 3: 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) $distance(B,A) \leq 0$.

7. Computation Graphs

In Section 6, the notion of a computation tree for a Modechart specification was introduced. Since a system often has infinite traces, our goal is to construct a finite *computation graph* that represents the computations of the system. In a 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 6(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 6(b) shows a typical global state graph for this system, with modes entered at each point are underlined. 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.

Example 3:

Consider the Modechart specification in Figure 7(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 timing constraints. The transitions M1→M2, M2→M3, and M3→M1 have deadlines of 100, 200, and 40, respectively. The transition from mode M4 to M5 cannot be taken until the delay of 1000 time units after starting in M4. Figure 7(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 7(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 7(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

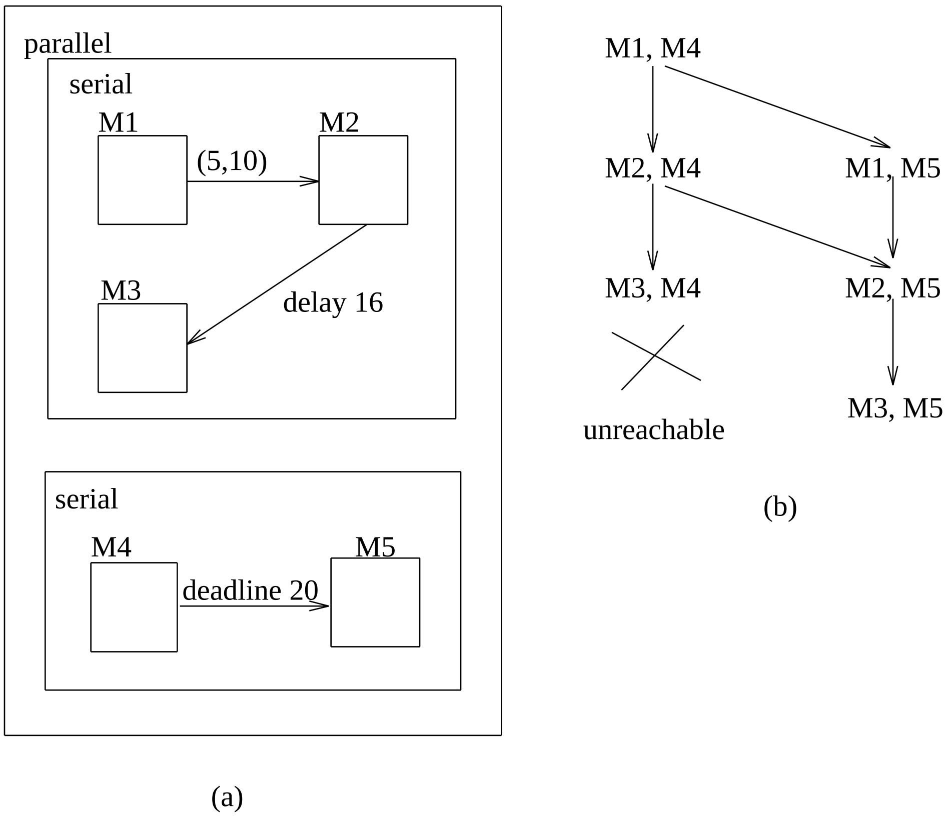

Figure 6.

while it remained in M4. In particular, the edge labelled α in Figure 7(b) should go from vertex 'M3,M4' to a new instance of vertex 'M1,M4'. Figure 7(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.

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, under certain conditions the new point P_n may be 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. If so, 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 some

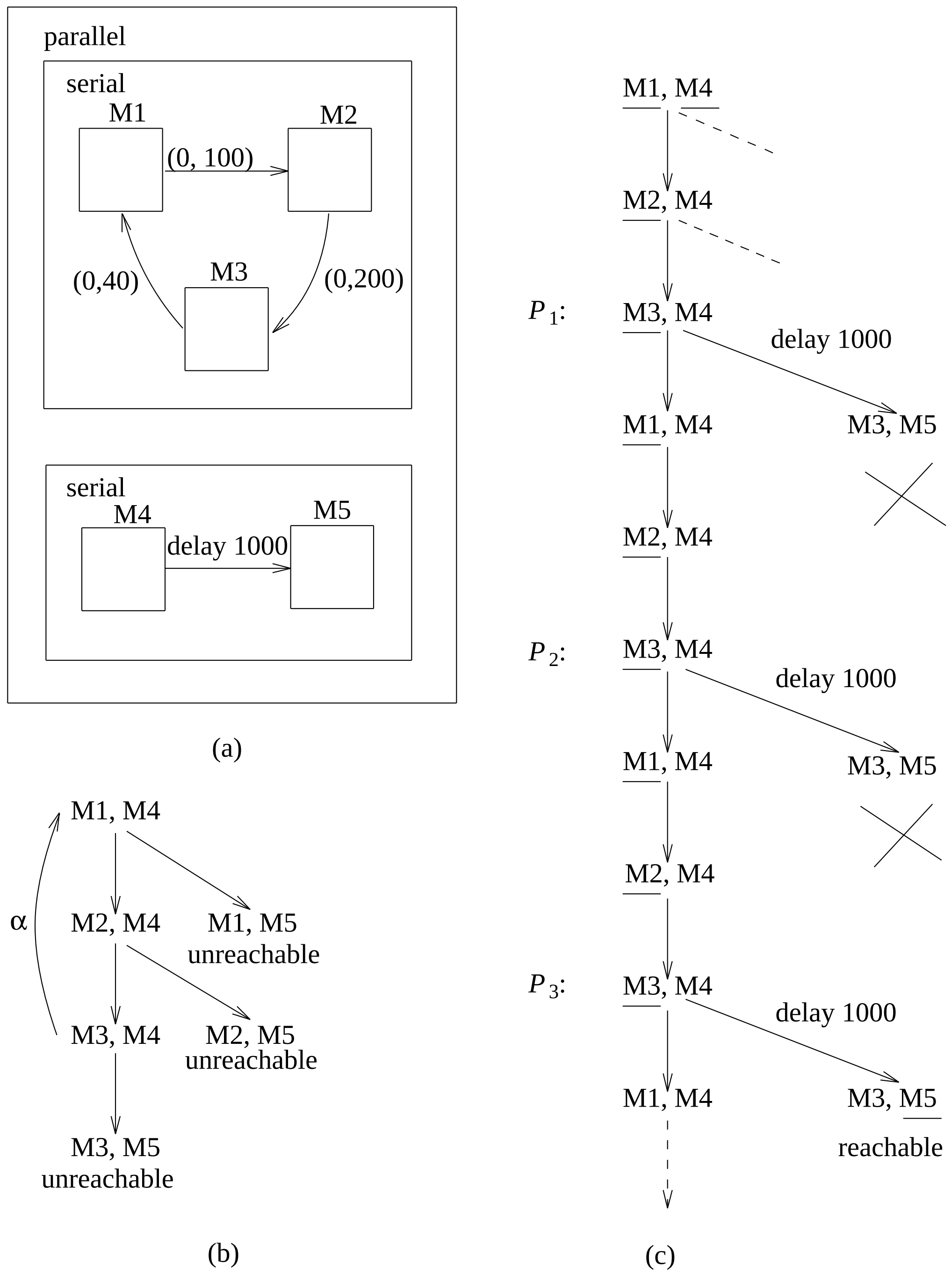

Figure 7.

points in the graph may correspond to infinitely many event occurrences.

As in a computation tree, the vertices in a computation graph are points, and an edge from one point to another indicates that the second is an actual successor of the first. For convenience, in the figures, each point is labelled with its name, with the modes entered by its entry event in boldface. After each point the system remains in the same modes until the next point. We shall refer to the modes active in a point (that appear in its name) as the *components* of that point.

Before we can present the conditions under which a point can be shown to be an instance of another point in the graph, we first need to be able to effectively determine the actual successors of a point. To do so we will define the separation graph for a finite path. Given a finite rooted path in a computation tree (or graph), its separation graph is a weighted directed graph with vertices the points on the path together with points representing the potential successors of the endpoint of the path. For brevity, we will refer to the vertices representing points as the points themseslves in what follows. Points representing potential successors differ from the potential successors themselves in that they have empty event sets, so that there is only one potential successor point for each name and entry event, rather than two if the final point on the path had a non-empty event set.

The edges of the separation graph are the same constraint edges as before. The exceptions being that constraints on points on the path due to enabled transitions (clauses (7) and (8) in the definition of a constrained trace) are omitted if the enabeled transition is actually taken since they are then redundant, and applied only to the point that results in the transition being disabled if they are not, which also enforces them on the intermediate points. Constraints due to transitions enabled at the endpoint are represented by edges incident on the potential successors rather than on the points on the path. Since the potential successors do not have an event set component, they are not subject to constraints due to clauses (3) and (4), which enforce event set inheritance. Also, since the potential successors will come after the points on the path, there is an edge with weight 0 from the endpoint of the path to each of the potential successor points, corresponding to clause (2). Note that this captures all of the constraints imposed on the corresponding constrained trace, since any constraints due to edges not in the separation graph are captured by paths with the same weight as the edge, since the missing edges are all due to clauses (7) and (8), and there is a path following edges due to (2) ("tree edges") to a point where the transition is either disabled, taken, or used to generate a potential successor, which has an edge with the appropriate deadline.

Figure 8 is an example of a separation graph for the path P_0-P_1-P_2 of example 1. In the figure, (and in the figures that follow) event sets are omitted for space reasons, and the entry event is indicated by printing in **bold** type the name(s) of the modes entered by the event. Since the modechart is flat, the root and serial mode names are also omitted. Also, in the figure, multiple edges are allowed to illustrate all of the constraints, as between P_2 and A_3, although the only edge actually present is the edge with maximum weight.

Lemma 4 : Given a path ending at point P, with potential successors $A_1, A_2, \ldots A_n$, There is an actual successor of P with name and entry event corresponding to A_i if and only if j $distance(A_j, A_i) \leq 0$.

Once the names and entry events of the actual successors are determined, it remains necessary to determine the event sets. If some actual successor, independent of event set, has a distance of 0 to the endpoint of the path, then all actual successors must inherit the event set of the endpoint of the path, since not doing so introduces a delay of 1 from the endpoint, and the point with a distance of 0 to the endpoint of the path will then have a zero distance to the proposed actual successor that did not inherit the event set. Similarly, any actual successor that has a positive distance to it from the endpoint of the path can not inherit the endpoint's event set. Accordingly, if no actual successor has a distance of 0 to the endpoint of the path, each actual successor that has a distance to it of 0 from the endpoint of the path is split into two actual successors, one that inherits the

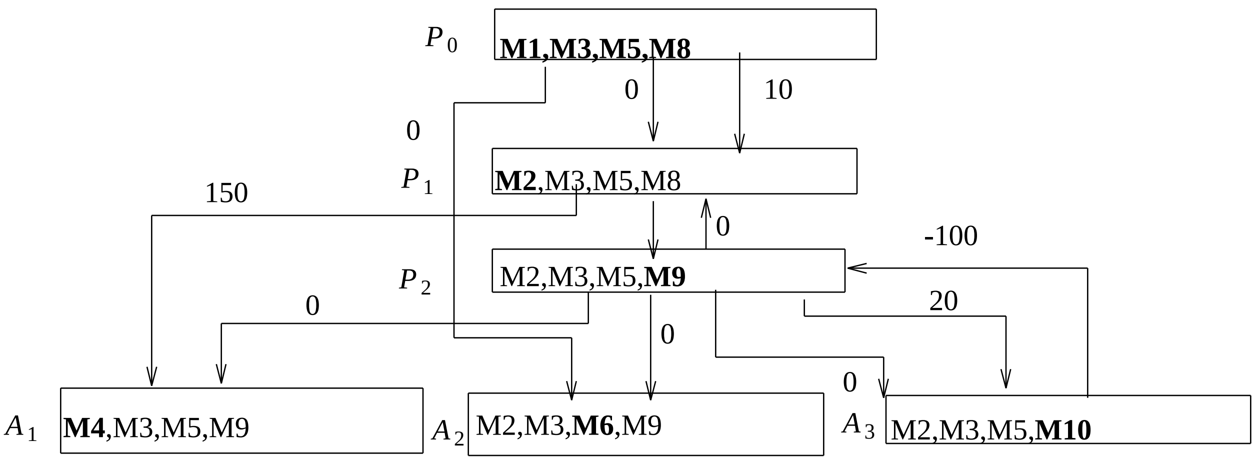

Figure 8

event set of the endpoint, and one that does not, and becomes subject to the appropriate clause (3) or (4) constraints.

As one last preliminary observation, for purposes of determining future behavior, two entry events that result in entry of the same modes are indistinguishable, since constraints are always enforced with respect to mode entry, and the only other way to distinguish two such entry events is if at least one is a condition event, in which case the points will be distinguished by their event sets. To that end, two entry events are *equivalent* if they cause entry of the same modes. We are now ready to define when two points in a computation tree (or graph) are equivalent so that one can be regarded as an instance of the other.

Definition: Suppose P_m and $P_{m'}$ are two points in a computation graph with the same name and event set, and equivalent entry events. Also, suppose Λ is the set of potential successors of P_m (as defined for separation graphs above) 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* holds for P_m and P'_m iff for each pair of corresponding points $A, B \in \Lambda \cup \{P_m\}$ and $A', B' \in \Lambda' \cup \{P'_m\}$ (where P_m and P'_m correspond, and $A \in \Lambda$ and $A' \in \Lambda'$ correspond if they result from taking the same transition):

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

Lemma 5: Suppose P_m and P'_m are two points in a computation graph with the same name and event set and with equivalent entry events, and P_m and P'_m satisfy the *distance equivalence condition*. Then:
(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 A and A'.

Proof:
(i) This follows from the definition of the deq condition and lemma 4.
(ii) This result follows from the way distances may be calculated, and the way successors are generated. Since A and A' are corresponding points, they must have the same components and event sets. Therefore they have isomorphic sets of potential successors λ and λ'. It remains to show that the distances between the corresponding successors of

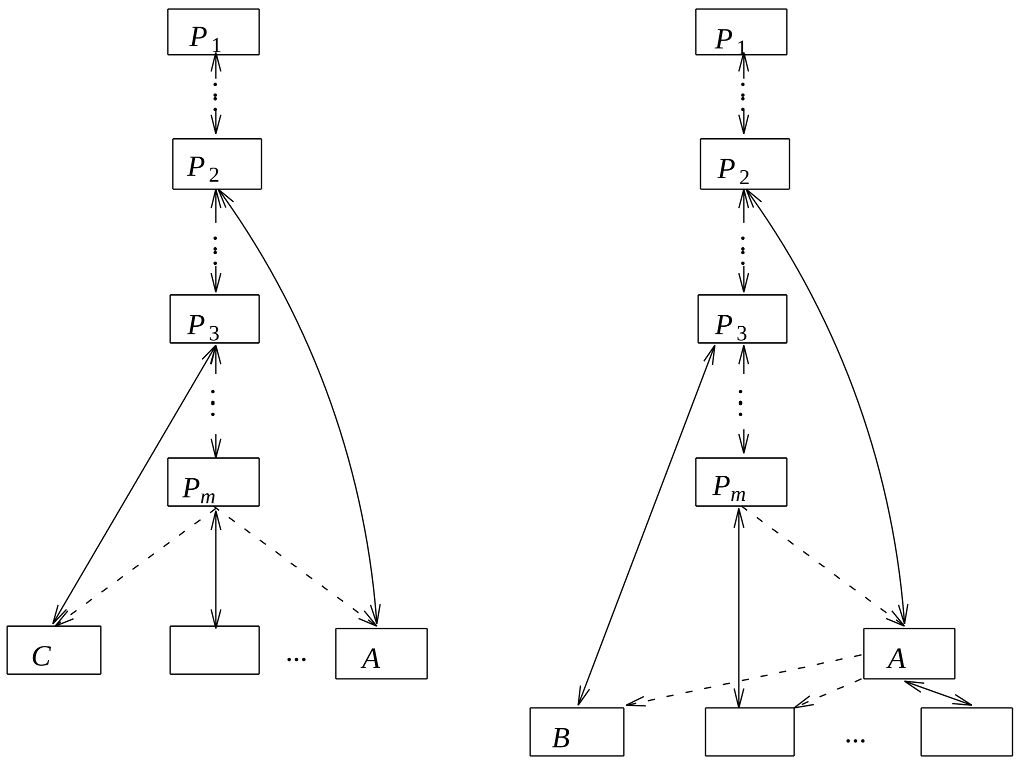

Figure 9

A and A' satisfy the deq condition. This follows from the definition of distance. The argument that follows is based on the Floyd-Warshall all-pairs shortest paths algorithm.

First consider the distances between A and A' and their corresponding successors (e.g. A to B in figure 9b). The successors will be either new successors (with A and A' as reference points), or carry-overs from the preceeding points (P_m and P'_m). If they are new, the distances will clearly be equal. If they are carry-overs, a path that determines the distance (longest paths are not unique) will either use one of the new edges, in which case the distance is the cost of the new edge (0), or it will not, in which case the distance is the old distance, which is still valid since all of the edges appearing in the path defining the distance are present in the new separation graph, because of clauses (7) and (8) in the definition of constrained trace. In both cases, the corresponding distances (from A and A') will be equal, since in the first case $0 = 0$, and in the second, because the deq condition holds for P and P'. Since the actual distance is the maximum of the two, then the distances are equal. A similar argument holds for the distances from the successors to A and A'.

Now consider the corresponding distances between successors. This is basically an iteration of the all-pairs algortihm. A path that may go through a new point either does so, in which case the cost of the path is the cost of the path from the source to the new point plus the cost of the path from the new point to the destination (without re-entering the new point), or the cost of the old path. Since in this case, all corresponding longest paths have the same cost, either from the discussion in the preceeding paragraph, or the deq condition on P and P', the distances between corresponding potential successors of A and A' are equal. □.

Theorem 6: 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.

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.

Theorem 7: Let A and B be potential successors of P. Then $distance(A,B) = \infty$ or $-d \le distance(A,B) \le r$, where d is the smallest deadline on the transition to point A, and r is the largest delay on the transitions corresponding to point B.

Theorem 8: The computation graph is finite.

We conclude this section by stating without proof two theorems about computation graphs that are necessary for the decision procedures of section 9.

Theorem 9: Given two rooted paths inluding in order the not necessarily distinct points P,Q,R and S and P′,Q′,R′,and S′ respectively, such that the first path ends at S and the second at S′, and P and P′ satisfy the disatance equivalence condition, and Q and Q′, R and R′, and S and S′ all correspond, then $distance(Q,R)=distance(Q',R')$ and $distance(R,Q)=distance(R',Q')$.

Theorem 10: Given a finite path containing two points P and Q in that order, if $-distance(Q,P) \le t \le distance(P,Q)$, then there is a computation that agrees with the trace of the path in which the difference in the times of the entry events of P and Q is t.

ALGORITHM: (Computation Graph Construction)

1. Construct P_0, the initial point in the computation graph,
   denoting the modes that the system is in initially.
   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. If not, split P_m if appropriate, and replace P_m with
   the resulting points.
   For each of the resulting points P'_m, if there is another point
   P_n in the graph such
   that P_n and P'_m have the same name and
   the same event set and equivalent entry events, and
   the *distance equivalence condition* holds for
   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 9 implies that the distance between two points on a path in a computation graph is independent of the path to the root, since a point can have two ancestors only if the distance equivalence condition is satisfied by the two instaces of the point on the two paths. Theorem 10 on the other hand shows that the definition of distance is exact, in the sense that any allowed distance can in fact occur in a computation.

8. 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 its computation graph. This example is based on one that appears in [Leveson & Stolzy 85].

8.1. Specification

Figure 10 illustrates a Modechart specification for a railroad crossing. The system consists of two parallel components: a process monitoring 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.
DOWN:	Gate is down.
MOVEUP:	Signal is received to raise gate.

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 only a trivial lower/upper bound on when this transition is taken. Entering BC mode signals the gate controller to begin moving the gate down. 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 the only timing constraint imposed on that transition is that the train spends 1 time units in the crossing. Entering PASSED mode signals the gate controller to begin moving the gate back up. 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.

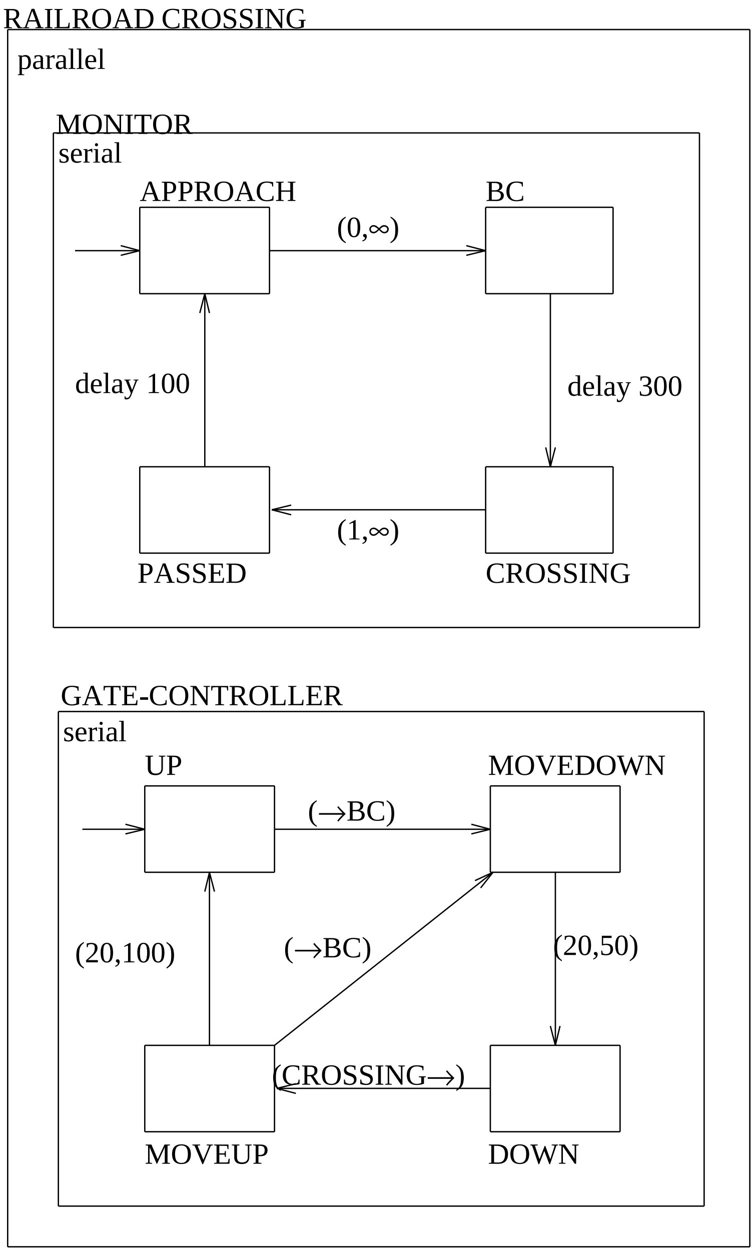

Figure 10

The controller process for the gate is initially in UP mode. When a signal is received from the monitoring process to lower the gate (mode BC is entered), a transition to mode MOVEDOWN is taken. While in MOVEDOWN, the gate is lowered. Taking the transition to DOWN represents completion of the process of lowering the gate. When a signal is received from the monitoring process to raise the gate (PASSED is entered), a transition to mode MOVEUP is taken. While in MOVEUP, the gate is moved up. Upon completion, the transition to UP mode is taken. While in mode MOVEUP, if another signal is received to lower the gate, the process of moving the gate up is aborted and a transition to mode MOVEDOWN is taken.

8.2. 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:

$$\forall i\ \exists j\ \ @(\rightarrow \mathrm{DOWN},j) \leq @(\rightarrow \mathrm{CROSSING},i)\ \wedge$$
$$@(\mathrm{CROSSING}\rightarrow,i) \leq @(\mathrm{DOWN}\rightarrow,j)$$

8.3. Verification

Figure 11 illustrates the computation graph for the railroad crossing system during its construction. The unreachable points (those marked out) are shown to be unreachable by applying Lemma 4. One edge in the graph is of particular interest because it illustrates how the deq condition and Theorem 6 are used. Consider the edge from point 16 up to point 3 in the Figure 11. It is a backward edge to point 3. Observe that, by Theorem 6, it was not necessary to retain point 3′ at the bottom of the figure because the points 3 and 3′ satisfy the distance equivalence condition. The names of 3 and 3′ are the same, the event set for each point is $\{\rightarrow BC\}$, and each entry event enters only MOVEDOWN. Furthermore, the distances among the corresponding points satisfy the deq condition. Therefore, Theorem 6 guarantees that the subtree below 3 is isomorphic to the subtree below 3′. A similar argument produces the other up edges. Figure 12 shows the computation graph without all of the unreachable and replaced points.

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. In this case, it is not necessary to actually calculate the distances bewteen the points in the graph because the order of the points in the graph is sufficient to ensure that the formula is satisfied, although this is not true in general.

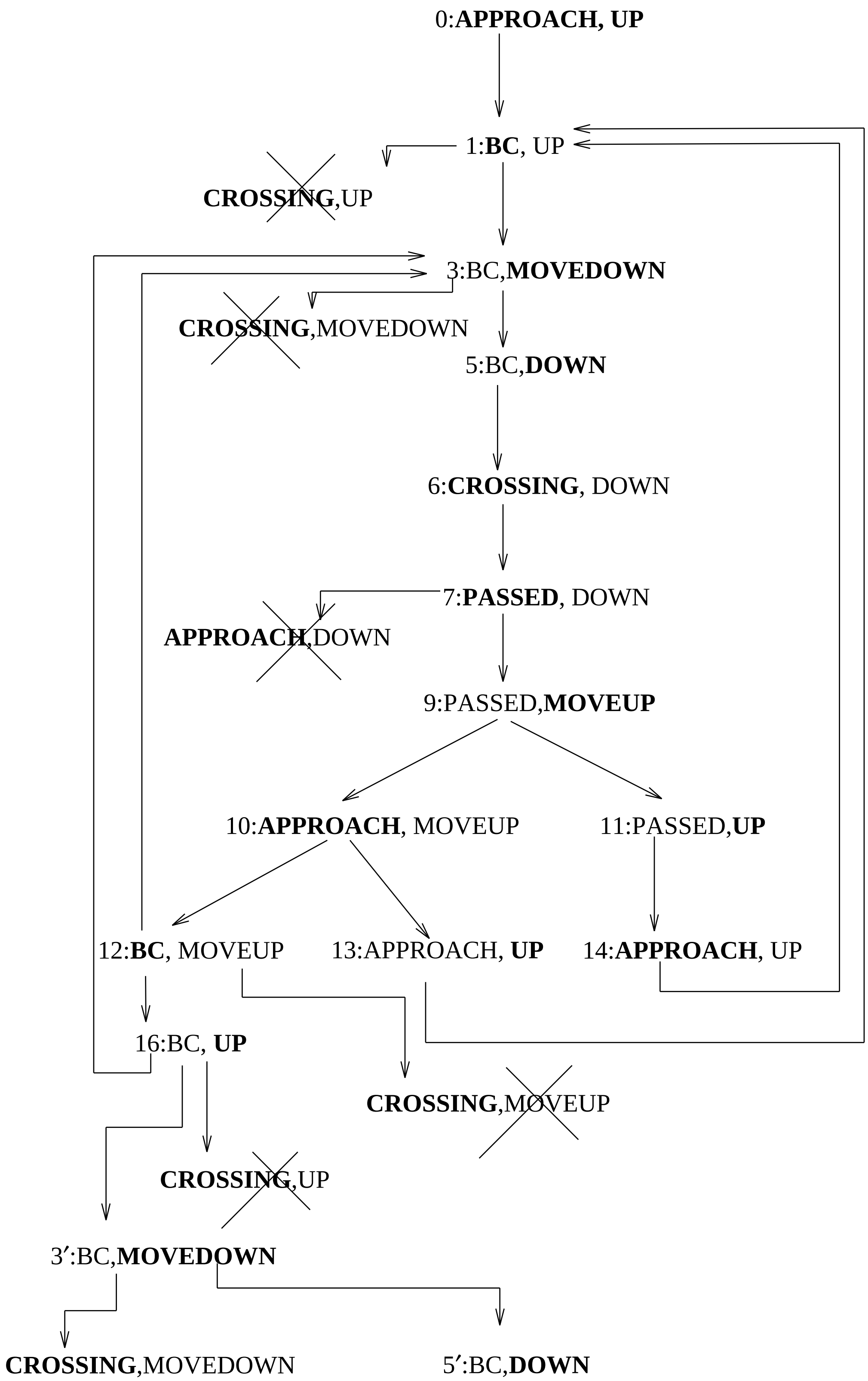
0:**APPROACH, UP**
1:**BC**, UP
CROSSING,UP
3:BC,**MOVEDOWN**
CROSSING,MOVEDOWN
5:BC,**DOWN**
6:**CROSSING**, DOWN
7:**PASSED**, DOWN
APPROACH,DOWN
9:PASSED,**MOVEUP**
10:**APPROACH**, MOVEUP
11:PASSED,**UP**
12:**BC**, MOVEUP
13:APPROACH, **UP**
14:**APPROACH**, UP
16:BC, **UP**
CROSSING,MOVEUP
CROSSING,UP
3′:BC,**MOVEDOWN**
CROSSING,MOVEDOWN
5′:BC,**DOWN**

Figure 11

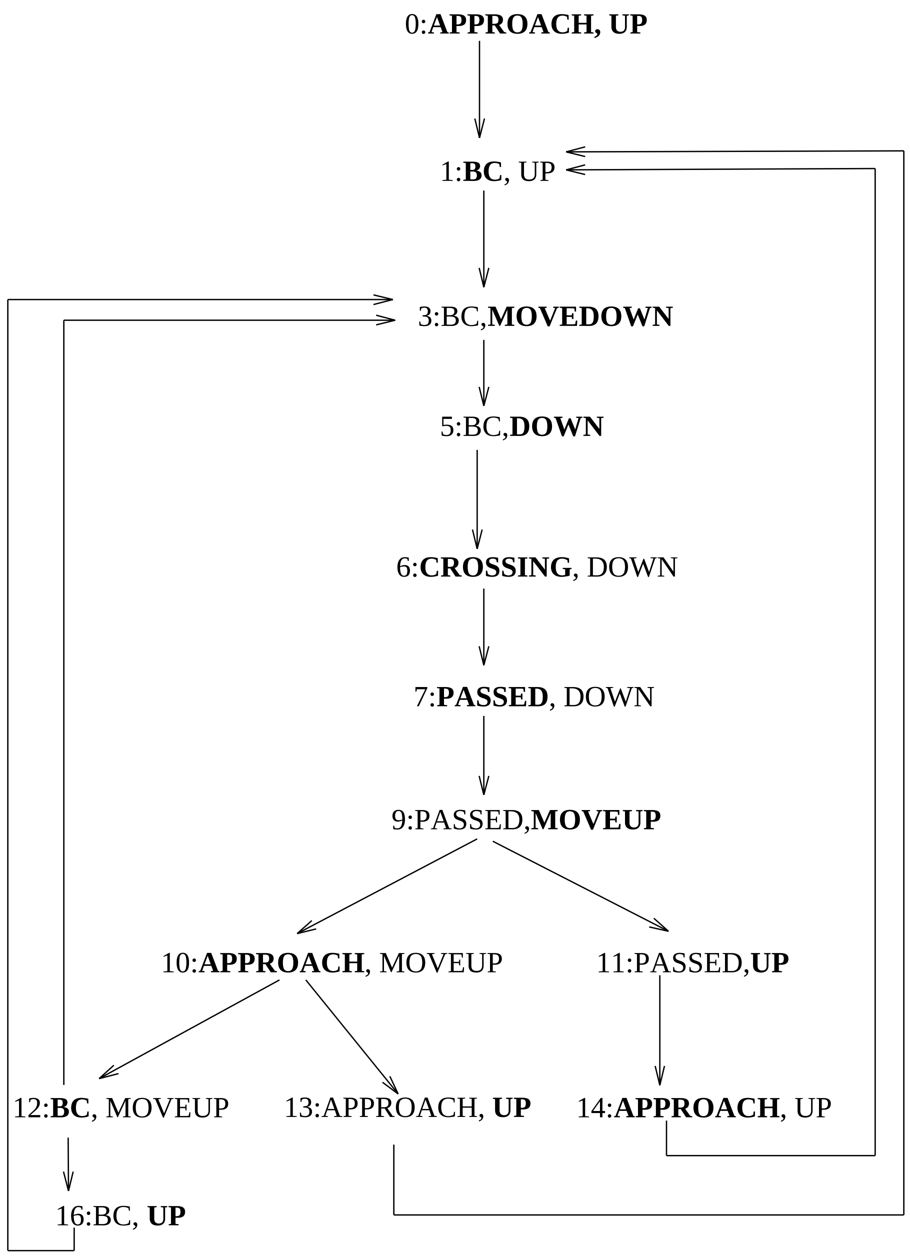

Figure 12

9. 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 three classes of timing properties expressed in RTL. For each class, we shall present computation graph based decision procedures for determining whether a given formula in the class is satisfied by or valid for a a computation graph.

Since RTL is undecidable ([Jahanian et al 88a]), no general procedure for checking decidability of RTL formulas exists. Also, in it was shown that the question of satisfiability of RTL formulas with respect to modechart specifications is also undecidable, so decision procedures for subsets of RTL formulas is the best that can be done.

9.1. Preliminary Definitions

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

$$e_1 \pm I \le e_2$$

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. Class 3 is a generalization of class 2. The three 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 $@(\downarrow \mathrm{A}, i)$ and $@(\downarrow \mathrm{A}, j)$ 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,

$$@(\rightarrow\mathrm{M},\mathrm{i}) \text{ and } @(\rightarrow\mathrm{M},\mathrm{i}+1)$$

are two endpoints defining an interval, and the inequality $@(\rightarrow\mathrm{M},\mathrm{i}) + 100 \le @(\rightarrow\mathrm{M},\mathrm{i}+1)$ denotes that the successive occurrences of entering mode M must be at least 100 time units apart. Similarly, the RTL formula

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

expresses the mutual exclusion relationship between the interval denoted by $@(\uparrow A,i)$ and $@(\downarrow A,i)$ and the interval denoted by $@(\uparrow B,j)$ and $@(\downarrow B,j)$, i.e., mutual exclusion of actions A and B.

The three 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 $@(E_1,i + c_1)$ and $@(E_2,i + c_2)$ 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 labeled with E_1 as representing the $(i + c_1)$ occurrence of E_1, the point representing the $(i + c_2)$ 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 corresponding occurrences of E_1 and E_2 is a function of i, rather than a constant, and therefore the length (in number of edges) of the path between the two corresponding points could be arbitrarily long.

Definition: Suppose G is a computation graph, and $@(E_1,i \pm c_1)$ 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.

As an example of preservation of a formula, note that the computation graph of the railroad crossing does indeed preserve the formula describing the railroad crossing safety property. Note however that if →UP were substituted for DOWN→, the formula would not be preserved, because the cycle 3-5-6-7-9-10-12-3 contains one instance of →DOWN and no instances of →UP.

In the remainder of this paper, we are interested only in RTL formulas which are preserved by a given computation graph. This does not appear to be a very restrictive condition, and roughly corresponds to the notion of bounded overtaking. 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. Also, the preserving requirement is not satisfied by the reduction of The remaining subsections introduce three subclasses of timing properties expressed in RTL for which one can decide if a computation graph satisfies the property given that the formula is preserved.

9.2. 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 RTL formulas of the form Q F where Q is either $\forall i$ or $\exists i$, and F is a quantifier free formula with i as its only free variable, such that each inequality in F is of the form

$$e_1 \pm I \leq e_2$$

where e_1 and e_2 are endpoints, and I is a natural number. Observe that the endpoints in F are related because the same variable appears in the occurrence index in each endpoint. The following class 1 formula states that two successive entries to the mode M are either 10 time units apart or within 5 time units of each other:

i @(→M,i)+10 ≤ @(→M,i+1) \/ @(→M,i+1) ≤ @(→M,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:

i @(→M,i) ≤ @(↑A,i) /\ @(↓A,i) ≤ @(M→,i)

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

$e_1 - e_2 \leq I$ (1)

$I \leq e_2 - e_1$ (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 ∀ or ∃, and n is the number of endpoints in F. As mentioned in subsection 9.1, if we take a point P in G labeled with an event E_j (for some $1 \le j \le n$) 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 inequality 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. Note that the correctness of this approach folows from theorems 9 and 10.

If the formula F is quantified by a , 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 , 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. This assures that a formula with a universal quantifier is valid with respect to the specification, and that a formula with an existential quantifier is satisfied by the specification.

$$i \ \ @(\rightarrow\text{CROSSING},i) + 400 \le @(\rightarrow\text{CROSSING},i+1)$$

is a class 1 formula which is a minimum separation property satisfied by the computation graph of the railroad crossing example.

9.3. Class 2: Exclusion/Inclusion of Intervals 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 9.1. 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:

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

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:

$$\mathrm{i\ j}\ \ @(\rightarrow\mathrm{M,i}) + 100 \leq @(\uparrow\mathrm{A,j})\ \wedge\ @(\downarrow\mathrm{A,j}) \leq @(\mathrm{M}\rightarrow\mathrm{,i})$$

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

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

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

$$\mathrm{i\ j}\ \ @(\downarrow\mathrm{B,j}) \leq @(\uparrow\mathrm{A,i})\ \vee\ @(\downarrow\mathrm{A,i}) \leq @(\uparrow\mathrm{B,j})$$

Given a computation graph G and 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) i j $e_3 \leq e_1\ \wedge\ e_2 \leq e_4$

(1b) i j $e_1 \leq e_3\ \wedge\ e_4 \leq e_2$

(1c) i j $e_1 \leq e_3\ \wedge\ e_4 \leq e_2$

(1d) i j $e_1 \leq e_3\ \wedge\ e_4 \leq e_2$

(1e) i j $e_3 \leq e_1\ \wedge\ e_2 \leq e_4$

Case (1a) 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. Case (1b) is similar and the other cases are all degenerate variations. The railroad crossing safety assertion is an example of a case (1a) formula.

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) i j $e_4 \le e_1 \lor e_2 \le e_3$

This case specifies that *every* instance of interval α is mutually exclusive with *every* instance of β. As before, the following cases are variations of case(2a).

(2b) i j $e_4 \le e_1 \lor e_2 \le e_3$

(2c) i j $e_4 \le e_1 \lor e_2 \le e_3$

(2d) i j $e_4 \le e_1 \lor e_2 \le e_3$

9.4. Class 3:Generalized Inclusion of Intervals

In this subsection we will introduce a class of properties that generalizes the notion of inclusion of intervals of class 2. Recall that in order to check for the existence of an included interval it was necessary to know which region of the computation tree (or graph) had to be examined for each interval governed by the leading quantifier. In this class, we allow multiple intervals as long as there are only two sets of related endpoints in the formula (only two distinct variables appear as index variables in occurrence function applications). In order to assure that we know which region of the graph to examine for each set of endpoints bound to the outer quantifier to find points corresponding to endpoints bound to the other quantifier, we will further make an inclusion requirement that each disjunct of the formula contains one relation of the form $e_1 \le e_3$ and one of the form $e_2 \ge e_4$, where e_1 and e_2 are bound to the outer quantifier, and e_3 and e_4 are bound to the inner quantifier. Doing so forces the endpoints bound to the inner quantifier to be found in a region bounded by e_1 and e_2.

Accordingly, the formulas of class 3 are of the form QF where QF has no free variables, Q is a quantifier prefix, F is a quantifier free formula in disjunctive normal form with two free variables, and each disjunct satisfies the inclusion requirement of the preceding paragraph. An example of a class 3 property is the following variation of the safety property satisfied by the railroad crossing example.

$$\begin{aligned}&\forall i \exists j\ @(\rightarrow \mathrm{BC},i) \le @(\rightarrow \mathrm{DOWN},j) \land @(\rightarrow \mathrm{DOWN},j) \le @(\rightarrow \mathrm{CROSSING},i) \land \\ &\quad @(\rightarrow \mathrm{CROSSING},i) \le @(\rightarrow \mathrm{PASSED},i) \land @(\rightarrow \mathrm{DOWN},j) \le @(\rightarrow \mathrm{PASSED},i) \land \\ &\quad @(\rightarrow \mathrm{DOWN},j) \le @(\rightarrow \mathrm{MOVEUP},j) \land @(\rightarrow \mathrm{CROSSING},i)+100 \le @(\rightarrow \mathrm{DOWN},j+1)\end{aligned}$$

9.5. Other Properties

The three classes of timing properties above obviously do not include all the properties which one may wish to verify about a Modechart specification. For example, the following properties cannot be expressed in any of the above classes:

- Each instance of an interval α contains an instance of interval β and 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.

10. Concluding Remarks

Earlier it was stated that the consideration of actions and state variables would be deferred. State variables in and of themselves do not pose a problem, however their values can only be changed by actions, and actions do pose problems. The reasons actions cause problems is that their atomicity can give rise to unsatisfiable timing constraints, so that a point could have potential successors and yet no actual successors. It is to avoid this problem, among others, that theorem 1 refers to completed constrained traces. Ignoring this problem for the moment, actions can be introduced by adding an action status to the name of a point for each active mode that hosts an action. The status of the action is either unstarted, started, finished, or aborted. The action status is unstarted when the mode is entered, and in addition to transitions, entry events can now include the change of status of an action. Each is a potential successor of a point with action status unstarted, but traces can only contain actual successors with a completed status if they first contain a point with started status. Likewise, a point with aborted status can only be an actual successor if its predecessor had action status unstarted. Finally, a transition that exits a mode can only result in an actual successor if the predecessors action status is completed or aborted. It is this requirement that can lead to a lack of actual successors, because a transition's deadline may prevent a point due to an action completion from being an actual successor, yet the transition event can not result in an actual successor either.

Another problem arises because of the restriction of event sets to condition events that occur earlier in the trace. The actual semantics of RTL do not have this restriction, and its absence can lead to computations that violate causality if a transition can cause the event that enables it. Both this problem, and issues surrounding the unsatisfiable timing constraints associated with actions, are discussed in more detail, although in a somewhat more general setting, in As a final technical observation, the semantics of Modechart in [Jahanian et al 88b] are actually given in terms of RTL formulas, rather than in terms of computations of constrained traces as in section 5. The proof that the causal models of the RTL formula that provides the semantics for a modechart specification correspond to the semantics given in section 5 is cumbersome but fairly straightforward, and is omitted.

At this point, it is possible to make a few more observations on the position of the method presented in this paper in comparison to other approaches in the literature. The other approaches all fall into three broad categories. The various Petri net approaches all have mechanical proof techniques but lack a specification language, so do not provide the possibility of general techniques such as the one described here. The various algebraic approaches on the other hand, such as Timed CSP, have sophisticated manual proof techniques, but as yet do not have any mechanical procedures (Lastly, various temporal logic based approaches, such as [Ostroff 87] , , [Henzinger et al 90] , and exist, and have mechanical procedures for checking satisfiability and doing model checking. However, the current work is distinguished from these approaches in that the specification language Modechart is richer in its features than many of the temporal logic based approaches, with its actions and triggering conditions. The approach here uses neither an interleaved nor maximal parallelism model of computation, which is more flexible and may in some sense better model real-time computation, although at the cost of the added complexity of the method presented here. Also, most of the decision procedures for the temporal logic approaches represents in each state the actual relative time of each event, rather than collapsing a range of relative times into a single point, as is done in this approach, so that this approach can result in a potentially smaller structure (computation graph).

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.

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].

Modechart is a specification language developed to mitigate the undecidability of RTL and provide a more structured way of writing specifications that are more accessible to users. This paper presented a method for analysis of requirements specification of systems that are described in Modechart. The method presented in this paper has been implemented as part of the SARTOR project. describes the experimental prototype and also covers some of the more practical issues in implementing a verifier.

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 may be 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.

Appendix -- Proofs of Theorems

Proof of Theorem 1:

That each point in the trace is an actual successor of its predecessor if the completed constrained trace has a computation follows from the fact that the set of constraints in force at the point is a subset of the constraints in force on the trace as a whole, and if the trace has a computation all of the constraints are satisfiable, and therefore each subset is satisfiable as well, so that the point is an actual successor. Suppose instead that the trace does not have a computation. Then the set of constraints on the trace is unsatisfiable, and by compactness has a finite unsatisfiable subset. Let σ_i be the point in the trace with highest index that appears in the finite unsatisfiable subset. This finite subset is itself a subset of the constraints in force at σ_i, and if it is unsatisfiable, then so is the set of all constraints in force at σ_i, and so σ_i is not an actual successor of its predecessor.□

Proof of Theorem 2:

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:

$$A + I \le A$$

where $I = I_1+I_2+\cdots+I_n$. Clearly, I must be non-positive since each point on the path, including A, is an actual successor of its predecessor, so the set of constraints is satisfiable. Hence, the sum of weights on the transitions forming the cycle is non-positive. □

Proof of Theorem 3:

(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, there is a path with weight zero from A to B in the separation graph using edges due to clause (2) of the definition of constrained trace. Hence, $distance(A,B) \ge 0$.

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

Proof of Lemma 4:

If for some j the $distance(A_j, A_i) > 0$, and A_i was an actual successor of P, then there are two cases. In the first, the transition resulting in A_j is enabled at A_i, so that the separation graph of the path extended to include A_i would have an edge of weight 0 from A_i to A_j, resulting in a positive cycle, so that the set of constraints on the path is unsatisfiable, a contradiction with A_i representing an actual successor. In the second case, the transition to A_i disables the transition to A_j, in which case an actual successor corresponding to A_i would be subject to the deadline on A_j, and again a positive cycle would result, again resulting in a contradiction.

Conversely, if every potential successor has a non-positive distance to A_i, then the set of constraints obtained by making A_i an actual successor must be satisfiable, and so A_i represents an actual successor. □.

Proof of Theorem 6:

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

Proof of Theorem 7:

Any path from A to B must start at A, go to its reference point, go to a point having an edge to B, and follow that edge to B. If the smallest deadline on the transition to point A is ∞, then, because the only edges from A in the separation graph have weight $-\infty$, $distance(A,B) = -\infty$. Assume d is finite. Then since A is a potential succsessor of P, either A's reference point preceedes P, or is P. In either case, the distance from the reference point to P (and hence to B) is non-negative. Therefore, $-d \leq distance(A,B)$.

To show $distance(A,B) \leq r$, there are two cases. First, assume P is A's reference point. Then, the edge to B must come from a point at or before P, so the distance to it from P is non-positive, as is the distance from A to P, so the largest $distance(A,B)$ can be is r. Alternatively, suppose P is not A's reference point. Then any path from A, to its reference point, to the point that is the source of an edge to B, and then to P must be non-positive (since P is not the reference point of A the edge P-A has weight 0) to avoid a positive cycle (prohibited by Theorem 2). Since P cannot preceede the source of the edge to B in the tree, the length of the path from A to that point is no longer than the length of the path from A to P. Accordingly, the length of the path from A to the source of the edge to B is non-positive, so the length of the path from A to B is bounded by the weight of the edge to B, which is bounded by r. Therefore $distance(A,B) \leq r$. □.

Proof of Theorem 8:

Partition the points of the computation graph by their name and event sets. Refine the partition by equivalent entry events. Further refine the partition by the distances between potential successors. By theorem 7, 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 6. The preceding algorithm ensures that the computation graph will contain at most one copy of a point from each equivalence class. Therefore the graph is finite. □.

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)

[Henzinger et al 90]. T. Henzinger, Z. Manna, and A. Pnueli, ‘‘An Interleaving Model for Real Time,’’ Department of Computer Science, STAN-CS-90-1329, Stanford University (September 1990).

[Jahanian & Mok 86a]. 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 & Stuart 88]. F. Jahanian and D. Stuart, ‘‘A Method For Verifying Properties of Modechart Specifications,’’ *Proceedings of the Ninth Real-Time Systems Symposium*, (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 & 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 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.).

Bibliography

[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).