A Methodology and Support Tools for Analysis of Real-Time Specifications *

Douglas A. Stuart
Aloysius K. Mok
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
{dastuart,mok}@cs.utexas.edu

Farnam Jahanian
Real-Time Computing Laboratory
Electrical Engineering and Computer Science Department
University of Michigan
Ann Arbor, MI 48109-2122
farnam@eecs.umich.edu

Keywords: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 and formal methods for analyzing system requirements becomes increasingly important. Modechart is a graphical specification language introduced to meet this need. The main focus of paper is on methods and supporting tools for representing and reasoning about properties of time-critical systems specified in Modechart. The paper describes a verification methodology which takes advantage of the structuring inherent in a Modechart specification to determine whether a system specification satifies the required properties. The paper also describes the implementation of a *mechanical verifier*, based on the proposed approach, which has been recently integrated as part of the Modechart Toolset prototype development environment from the Naval Research Lab [7].

*Supported in part by a research grant from the Office of Naval Research under ONR contract number N0014-94-1-0582

1 Introduction

With the increasing reliance on computer control of embedded real-time systems in diverse civilian and military applications such as avionics, air-traffic control, patient monitoring, and automated manufacturing, the need for dependable systems that deliver services in a timely manner has become crucial. Ensuring that an embedded real-time system meets its prescribed specification is a growing challenge that confronts software engineers and system designers. Several studies have demonstrated that the cost of detecting and removing software errors increases significantly as the development process moves from requirements specification toward implementation and system integration [11]. It has been estimated that the cost of removing an error from a system specification is an order of magnitude smaller than the cost of removing it from a system during integration testing. Other studies have demonstrated that errors in the specification are the most frequent types of software errors and the most expensive to correct [4, 28]. It has also been observed that safety-critical failures can often be traced to requirements specification or design errors very early in the software development process [22].

In spite of the complexity of real-time systems, they are being used to control and monitor important, even safety-critical, applications. During the last decade, software researchers have devoted much attention to formal methods for representing and reasoning about behavioral requirements of real-time systems. Researchers have argued that formal specification methods can reduce the number of errors in a specification by removing imprecision and ambiguity from a specification. Unlike informal approaches to software specification, such as natural language descriptions, the expression of software requirements or designs should produce a precise and clear statement of the desired behavior. A formal approach can organize the specification so that inconsistency and incompleteness are easier to detect. Furthermore, formal specifications are amenable to formal analysis including testing, validation, and verification techniques. A recent independent study of 12 cases in which formal methods were applied to the construction of industrial systems concluded that, "while still immature in certain respects, [formal methods] are beginning to be used seriously and successfully by industry to design and develop computer or computerized systems" [8, 9]. The same study also pointed out the lack of robust tools for formal methods, and in particular, support for specification and analysis of timing requirements.

The Modechart Toolset (MT), developed at the Naval Research Laboratory in collaboration with researchers at the University of Texas, is a collection of integrated tools for specifying and analyzing real-time systems [7, 32]. MT supports the formal specification of real-time behavior in a language called Modechart and formal analysis via formal verification, static completeness and consistency checking, and specification simulation. A recent report presented an overview of the Modechart Toolset and its application to an actual Navy real-time system, namely, the HARM Low-Cost Seeker Missile (HLCS) software [7]. The graphical interface to the toolset developed for the UNIX and X-windows environment, as shown in Figure 1, is described in the MT User's Guide from NRL [32].

This paper focuses on the *mechanical verifier* of the Modechart Toolset. The purpose of the verifier is to determine whether the specification of a system in Modechart satisfies a desired safety assertion. The paper describes the formal

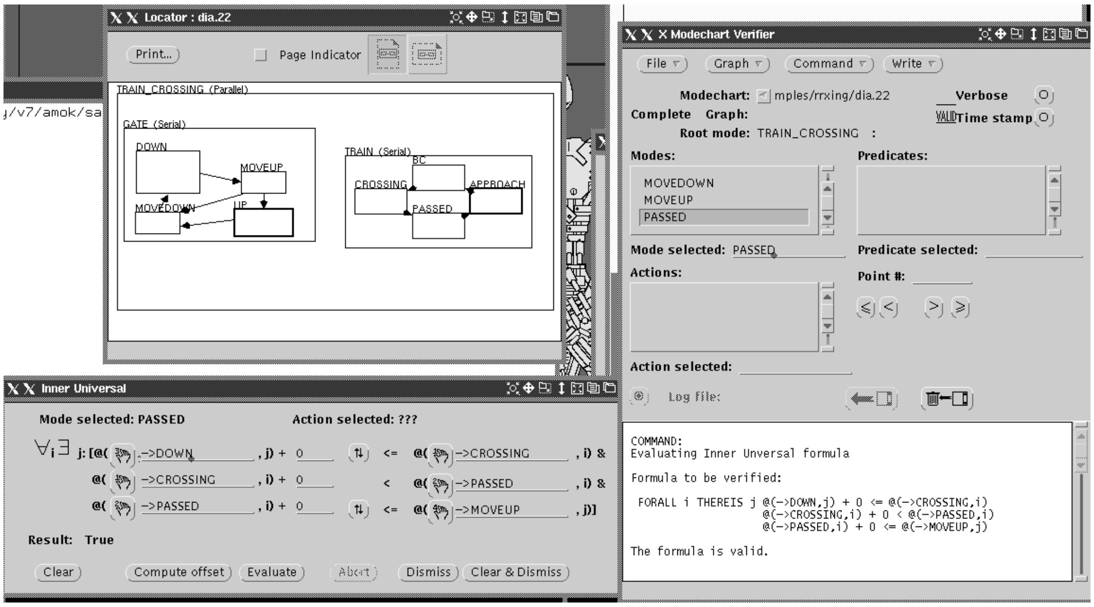


Figure 1: Specification and Verification Using the Modechart Toolset.

technique underlying the mechanical verifier. It also discusses the usage of the verification technique and classes of properties for which decision procedures have been developed. The Modechart specification language and its verification technique are illustrated via an example throughout the paper.

The organization of this paper is as follows: Section 2 briefly describes the verification approach. Section 3 presents an overview of Modechart and RTL. Section 4 illustrate how a finite computation graph can be generated for a modechart specification. This section also describe the usage of computation graphs in demonstrating the satisfiability of safety properties. Section 5 discusses the related work.

2 Approach

In our approach, a real-time system is formally specified in the Modechart graphical language. Desired properties about the system are represented as assertions in a formal language, called Real-Time Logic (RTL). The purpose of the verifier is to determine whether a timing or behavioral property (expressed in RTL) holds for a system specification (described in Modechart),

The objective is to use the Modechart specification to consider only a finite number of event occurrences to verify a system property. 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. If a procedure can decide a class of properties in RTL for this graph, then 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

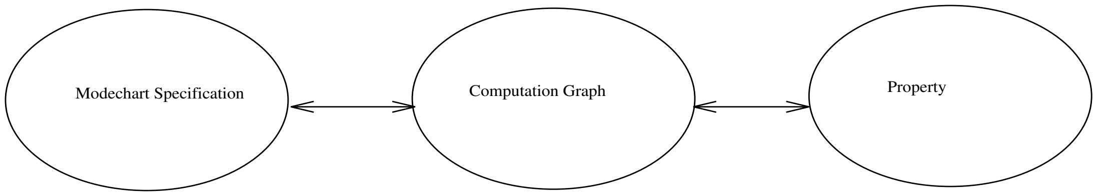


Figure 2: Methodology for Verifying Modechart Specifications.

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.

3 Specification of Real-Time Systems

3.1 Specifying Systems: Modechart

Modechart is a graphical specification language developed to provide a compact and structured way to represent real-time systems [16, 17]. Although similar in some ways to Harel's Statecharts [13], Modechart is specifically tailored to representing time-critical systems. Modechart can be viewed as a language for constructing *modes*, which are control information that impose structure on the operation of a system. The concept of modes was first introduced by Parnas et al. to simplify describing the requirements specification of the A-7E aircraft [14, 1]. The semantics of Modechart is given in Real Time Logic (RTL), a first-order logic for the specification and analysis of such systems [19].

Modechart is extended from Statechart [13] with appropriate constructs for expressing timing constraints. Its visual hierarchical structure together with a small set of well-defined constructs positions it as an attractive programming language for event-driven real-time control systems. Fundamental constructs of a Modechart specification are *modes*, *mode transitions*, *events*, and *timing constraints*. *Modes* describe 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*

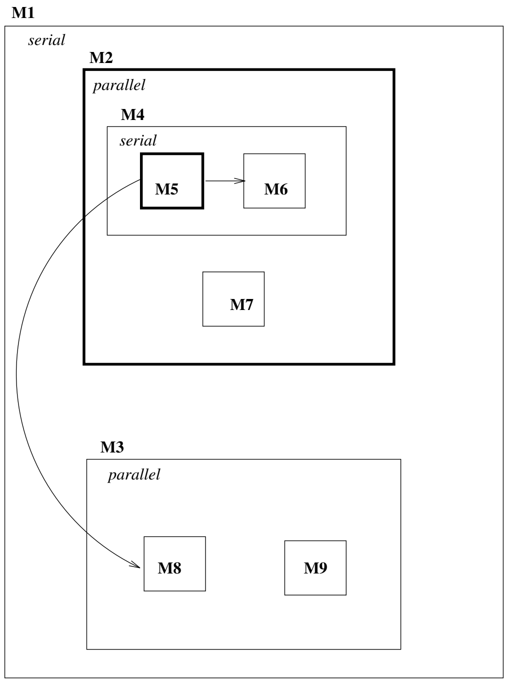


Figure 3: A Modechart Specification

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.

Figure 3 illustrates a modechart specification. The system consists of a 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. 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.

We distinguish between two classes of events for specifying conditions on transitions that depend on the occurrence of other transitions: (a) A *mode transition event* is of the form (M1→M2), where M1 is the source mode of the transition, and M2 is the destination mode, and (b) *Entry and exit events* correspond to entering and leaving a mode M, and are of the form (→M) and (M→), respectively.

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 a triggering condition for taking the transition, or a lower/upper bound restriction on when the transition may be taken, as defined below:

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 each p_j specifies a condition for taking the transition depending on the occurrence of an event and/or the truth values of certain predicates. In particular, each sub-condition p_j is of one of the following forms: (a) The sub-condition $\{(M_1, M_2, \cdots, M_m)\}$ which requires the system to be in one of the specified modes at the time t when a transition is taken, or (b) The sub-condition *E* which requires the occurrence of an event E at time t for a transition to be taken, where E is an event in the event set of the specification. 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 a pair (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. In each such condition $r \leq d$. 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 4, by associating timing constraints with mode entry. In particular, it allows the reference point for the timing condition to be the entry of the source mode when the semantics of such conditions are defined. 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. 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 presentation. More formal treatment of the syntax and semantics of Modechart appear in [16, 17].

3.2 Specifying Properties: 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.

When using RTL to describe or reason about Modechart specifiations, 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 entry and exit events for each mode in the specification. For each mode M, its corresponding entry and exit events are represented by →M and M→, respectively. Another subclass consists of the transition events for each mode transition in the specification. For a transition from mode M1 to mode M2, the corresponding transition event is represented by M1→M2.

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

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

(where $\mathcal{E}$ is a set of event constants, $\mathcal{Z}^{+}$ is the set of positive integers, and $\mathcal{N}$ 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 $@$.

$$\forall i @(\rightarrow EXIT, i) \leq @(\rightarrow LOGOUT, i) \wedge @(LOGOUT \rightarrow, i) \leq @(\rightarrow EXIT, i) + 20$$

is an example of an RTL formula, which states that each time the $EXIT$ mode is entered, the mode $LOGOUT$ is entered, and the mode must be exited twenty time units later.

3.3 Example: Railroad Crossing Control System

We choose a well-known example from the literature to illustrate the constructs of Modechart. This example is selected because its specification is concise and it is a suitable representative of the types of properties that we wish to specify in a real-time system. The example describes a monitoring and control system for a railroad crossing, mentioned in [23] and [18]:

"The gate at a guarded railroad crossing is to be software controlled, and since the gate cannot control the train, a real-time monitor is used to detect a train that is approaching. The monitor is an early warning signal at a distance of 1/2 mile from the crossing that gives notice to the gate controller that a train is approaching. It is know that the train takes at least 300 time units to reach the crossing from the signal. It is also known that the time required to lower a gate is between 20 and 50 time units. The controller itself can detect the departure of the train and it requires between 20 and 100 time units to raise the gate. It is also known that trains are scheduled so that it takes at least 100 time units from the time a train leaves the crossing until the next train reaches the early warning monitor."

Railroad Crossing Specification in Modechart:

The Modechart specification of the railroad crossing system is shown in Figure 4. The system consists of two parallel components: a process monitoring oncoming trains and a controller interface to the guard gate. The system is specified in Modechart by a parallel mode, RAILROAD-CROSSING, that consists of two serial modes, MONITOR and GATE-CONTROLLER, representing the concurrent components in the system. The innermost boxes in each serial mode are primitive modes representing the internal states of each serial mode.

The monitoring process can be in one of four modes, depending on its view of the train:

APPROACH:	Train is approaching, but very far from, crossing.
BEFORE-CROSSING:	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 mmde 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

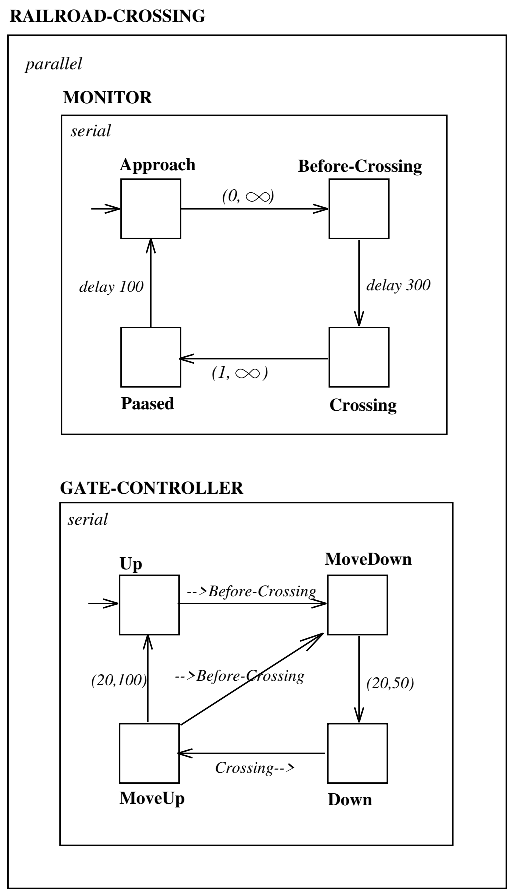


Figure 4: A Modechart Specification

transition is that the train spends at least 1 time unit 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 has 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 (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.

In the Modechart specification of the railroad crossing system, as shown in Figure 4, the conditions on all mode transitions in the serial mode MONITOR are lower/upper bound conditions. Similarly, the conditions on mode transitions from UP to MOVEDOWN and from DOWN to MOVEUP are triggering conditions.

Safety Property in RTL:

$$\forall i \forall j \; @(\rightarrow DOWN, j) \leq @(\rightarrow CROSSING, i) \wedge @(CROSSING \rightarrow, i) \leq @(DOWN \rightarrow, j).$$

4 Analysis of Real-Time Systems

Since a goal of this work is the mechanical analysis of real-time specifications, an analysis method is needed that is suitable for mechanization. While deduction and theorem proving, and process algebraic manipulation can be mechanized, the model checking approach is particularly suited to mechanization, especially when the task at hand is checking that Modechart specifications are models of Real Time Logic formulas. The basic idea of analysis based on model checking is to form a finite representation of the behaviors of a specification, and then determine whether the behaviors thus modeled satisfy the properties of interest. This section will first introduce the particular finite representation for Modechart behaviors, the *computation graph*, and then discuss how computation graphs can be used to determine if a modechart has a particular property stated as an RTL formula. The details of the theorems and algorithms presented in this section can be found in [34].

4.1 Computations

Since a system often has infinite traces, and as indicated our goal is to construct a finite *computation graph* that represents the computations of the system, it will be necessary to replace a potentially infinite set of potentially infinite objects with a single finite object, or at least a finite set of finite objects. For non-real-time systems, it is generally sufficient to construct a global state reachability graph, where each global state is represented as a vertex in a graph,

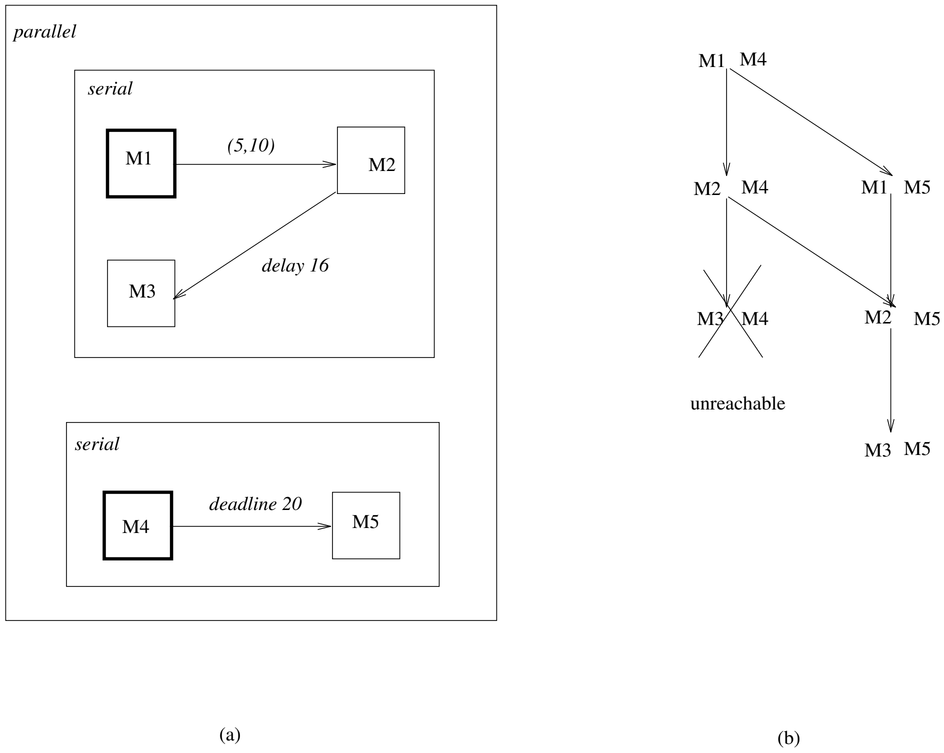


Figure 5: Deadlines May Result in Unreachable Points.

and directed edges join a global state with its successors, and each global state appears at most once. Since a mode is control information and may be *viewed* as a state, it is helpful to discuss why a global state reachability graph is not sufficient when dealing with real-time systems specified in Modechart. We use two examples to illustrate ways in which the timing constraints on transitions invalidate the usual global reachability graph.

Example 1:

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 entering 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 of entering 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.

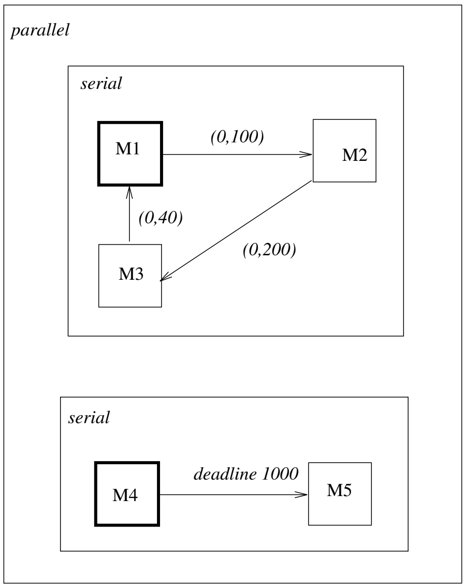
Example 2:

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 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 6(b) shows a reachability graph for this system in which delays and deadlines are considered to discard 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, second, and subsequent times that the system entered M1 while it remained in M4. In particular, the edge labeled α in Figure 6(b) should go from vertex 'M3,M4' to a new 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.

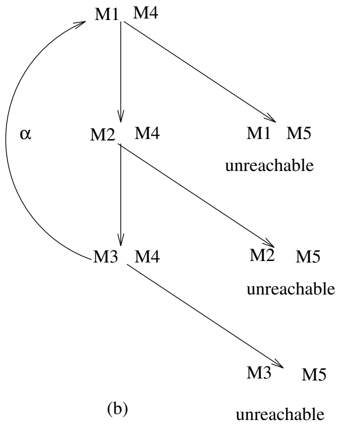
Having looked at some of the shortcomings of ordinary reachability graphs, we can turn to modifying the basic idea to one that is suitable for a real-time context. Although set of all computations of a modechart specification will be represented by a finite computation graph, we will first represent them by a set of traces.

Each individual computation will be represented by a timed trace, where the elements of the trace, called *points*, will represent configurations of the system specified by the modechart, and contain sufficient information to indicate which modes are active and which transitions can be taken. 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 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.

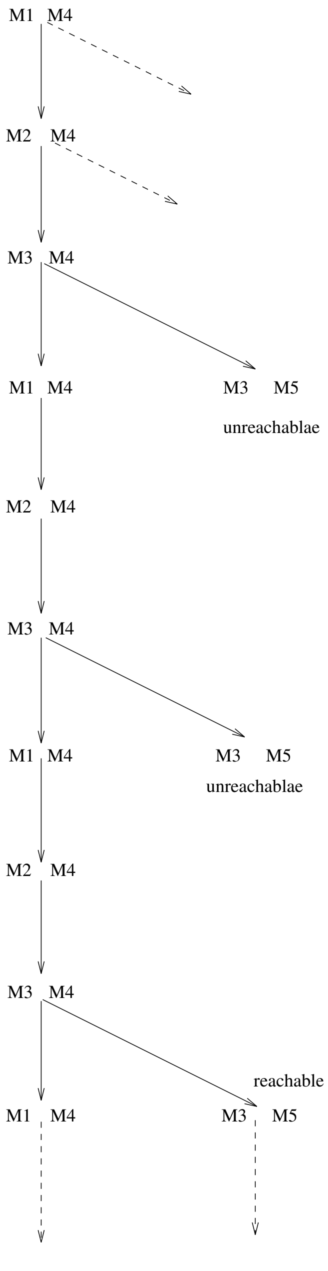
Having defined the elements of the traces, it is now possible to define the (untimed) traces of a specification. A *trace* of a modechart specification is a sequence $\{\sigma_i\}_{i\geq 0}$ of points of the specification such that σ_0 is the initial point, and for all $i > 0$, the entry event for σ_i 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



(a)



(b)



(c)

Figure 6: Delays May Result in Repeated Point Labels.

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. time(σ_i) ≥ 0.
2. time(σ_i)$\leq$time(σ_{i+1}).
3. If σ_{i+1} inherits the event set of σ_i, then time(σ_{i+1})=time(σ_i).
4. If σ_{i+1} does not inherit the event set of σ_i, then time(σ_i)+1$\leq$time(σ_{i+1}).
5. If the entry event of σ_{i+1} is the transition event for a transition governed by a triggering condition, then time(σ_{i+1})=time(σ_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+1}, then time(σ_j)+r $\leq$time(σ_{i+1}) and time(σ_{i+1})-$d$$\leq$time($\sigma_j$).
7. If there is a transition governed by a triggering condition enabled in σ_i then time(σ_{i+1})=time(σ_i).
8. If there is a transition governed by timing constraint with deadline d enabled in σ_i and σ_j is the reference point for the transition at σ_{i+1}, then time(σ_{i+1})-$d$$\leq$time($\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.

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.

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 following definition is needed in order to state Theorem 1 which provides a test for distinguishing the actual and potential successors of a trace.

Definition:

The *distance* from point A to point B in a constrained trace is defined to be the earliest time B can happen relative to A.

If the distance from A to B is positive, it serves as a delay, and if the distance is negative it serves as a deadline. Intuitively, if A precedes B in a trace, then $distance(A,B)$ in the trace is the earliest relative time B can happen after the occurrence of A; $distance(B, A)$ then is the earliest relative time A can happen before the occurrence of B (the longest time B can happen after the occurrence of A). The distance between two points can be evaluated in a number of ways. One approach is to solve the set of inequalities making up the constraints on the trace. Another approach, covered in detail in [34], represents these constraints as a graph, and the distance between points is then the weight of a path in the graph.

We can now state the following theorem which allows us to effectively determine the actual successors of a point.

Theorem 1 : Given a trace ending at point P, with potential successors A_1,A_2,... A_n, There is an actual successor of P with name and entry event corresponding to A_i if and only if

$$\forall j \; distance(A_j, A_i) \leq 0.$$

4.2 Computation Graphs

Having defined the individual computations of a specification in terms of traces, we can represent all of the computations of a system specified in Modechart 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 such a tree. Such a tree would be infinite. A computation graph can be constructed in an analogous fashion. Each new point (actual successor) generated, P_n, may be an *instance* of a point P_m already in the graph; that is, the computation subtree below P_n may be identical to the subtree below P_m. If so, the point P_n need not be added to the graph. Instead, an edge to P_m from the immediate predecessor of P_n can be added to denote the occurrence of another instance of P_m in the computations of the system. Such a computation graph may be finite, even though some points in the graph may correspond to infinitely many event occurrences.

The main result of this section will be an equivalence condition which can be used to carry out construction of computation graphs as outlined above, and such that the resulting computation graphs will be finite. As with traces, 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 labeled with a unique id number and its name, with the modes entered by its entry event in **bold**. 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.

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 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):

$$\begin{aligned} distance(A, B) &= distance(A', B') \\ distance(B, A) &= distance(B', A') \end{aligned}$$

Theorem 2: Given the two points P_m and P'_m that satisfy the distance equivalence condition, the set of traces generated from point P_m is isomorphic to the set of traces generated from point P'_m.

The following algorithm can therefore be used to generate a computation tree for a modechart specification that captures all of the computations of the specification. This section concludes with the following theorem that ensures that the algorithm will always terminate.

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 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 $\tau_1, \tau_2, \ldots, \tau_k$ be the set of transitions that can be taken from P'_m with potential successors $Q_1, Q_2, \ldots, Q_k$.
   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 3: A computation graph constructed according to the graph construction algorithm above is finite.

4.3 Computation Graph Based Analysis

Computation graphs were developed not as an end of themselves, but as a step in the development of a mechanical process to verify properties of Modechart specifications. Unfortunately, it was shown in [15] that the problem of determining whether a modechart satisfies an arbitrary RTL formula is undecidable. However, while this is the case for arbitrary formulas, it is possible to construct subsets of RTL for which the problem is decidable. This section will introduce several classes of RTL formulas for which decision procedures do exist. Also, there is additional information about the behavior of a Modechart specification that can be obtained from the computation graph that may not be best expressed in terms of RTL formulas. This section will also illustrate how this type of information can be extracted and used.

Although there is a large subset of RTL formulas that can be checked for satisfiability with respect to Modechart specifications, this section will focus on two classes of such formulas. The first class, known as *single interval formulas*, since all instances involve the same instance variables, contains formulas of the following form:

$$\forall i \bigwedge_j (@(e_{j,1}, i + c_{j,1}) + c_{j,3} \leq @(e_{j,2}, i + c_{j,2})),$$

for events $e_{j,k}$ and constants $c_{j,l}$ for $k \in \{1,2\}$ and $l \in \{1,2,3\}$ and some positive j. Such formulas concern the timing relationships among instances of events quantified with the same instance variable. Formulas in this class can be used to establish timing relationships among events that have synchronized instances. For example, referring to the Railroad Crossing Example, the formula

$$\forall i @(MOVEDOWN \rightarrow, i) = @(\rightarrow DOWN, i) \wedge @(DOWN \rightarrow, i) = @(\rightarrow MOVEUP, i)$$

establishes that when the gate has finished moving down, it is down, and when it begins moving up, it is no longer down. Since the gate may begin moving down while moving up without waiting for the gate to be completely up, the mode UP could not appear in such a formula and have it remain true, since the instances of UP are not synchronized with the other three modes of the gate.

A second class of formulas, known as *two interval formulas*, since there are two distinct instance variables, consists of formulas of the form

$$\text{Prefix(x,y)} \bigwedge_j (@(e_{j,1}, v_{j,1} + c_{j,1}) + c_{j,3} \leq @(e_{j,2}, v_{j,2} + c_{j,2})),$$

where Prefix(x,y) is a prefix of quantifiers for free variables x and y, $c_{j,l} \geq 0$ for $k \in \{1,2\}$ and $l \in \{1,2,3\}$ and some positive j, and $v_{j,k}$ is either x or y. Additionally, there is at least one predicate each of the forms $@(e_1, x+c) < @(e_2, y+c)$ and $@(e_3, y+c) < @(e_4, x+c)$. Such formulas can be used to establish the inclusion or exclusion of intervals of time bounded by events. For example, the safety property for the railroad crossing example, that cars not be run over, is the following formula of this class:

$$\forall i \forall j\ @(\rightarrow DOWN, j) \leq @(\rightarrow CROSSING, i) \wedge @(CROSSING \rightarrow, i) \leq @(DOWN \rightarrow, j).$$

Formulas of both classes are checked in the same basic manner. Each path segment in the computation graph that contains events that correspond to events in the formula being checked is identified, and a correspondence between the events on the path segment and events in the formula is established. The distances between the events on the path segment are then examined to determine if the timing among the events satisfies the timing requirements in the formula. The existence of one or more satisfying path segments is then used to establish the satisfiability of the formula, depending on the exact structure of the formula. In order for this to result in a decision procedure, the number and size of such segments must be finite. This is the reason for the additional restriction on the *two interval* formulas, and also necessitates the following additional requirement, that at any time the difference in the number of instances of each event associated with the same instance variable is bounded by a constant. In other words, the events associated with the same quantifier can overtake each other by only a bounded amount. This restriction is necessary for decidability.

There are other uses for computation graphs. The size of the computation graph provides some information about the behavior of the system. Also, examining the computation graph can indicate which modes are reachable, and what combinations of modes can appear together in the label of a point, and which such combinations result in sink points or points where the computation of the system may pause indefinitely. While the first can easily be expressed as a formula, the second can not be expressed as a formula at all, since formulas can only refer to computations, not traces. The third can be expressed as a formula, but only with difficulty. All three are easy to describe informally, and the answer to each is readily available by inspecting the computation graph, and may provide useful insight into the behavior of the system in question.

4.4 Analysis Example

This section will conclude with an illustration of the techniques developed in this section by applying them to the Railroad Crossing example. This will also illustrate how the tool implements some of the decision procedures discussed.

Figure 7 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 Theorem 1. One edge in the graph is of particular interest because it illustrates how the deq condition and Theorem 2 are used. Consider the edge from point 16 up to point 3 in the Figure 7. It is a backward edge to point 3. Observe that, by Theorem 2, 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 (omitted in the figures) is $\{\rightarrow BC\}$, and each entry event enters only MOVEDOWN. Furthermore, the distances among the corresponding points satisfy the deq condition. Therefore, Theorem 2 guarantees that the subtree below 3 is isomorphic to the subtree below $3'$. A similar argument produces the other up edges. Figure 8 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 between 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, because all of the constants are 0, and a distance of at least 0 is assured by constraints of type (2).

A prototype tool has been developed that implements some of the computation graph construction and query algorithms discussed in this paper. The tool was written in *C* under *X* using *Devguide*. Figure 9 shows the main tool window after the construction of the computation graph for the railroad crossing modechart. The window is divided into three areas. The top area consists of four menus. The `File` menu contains operations for reading and writing modechart specifications and computation graphs. The `Graph` menu contains commands pertaining to computation graph construction. The `Command` menu contains commands for evaluating formula and formula-like queries with respect to the computation graph. The `Write` menu contains commands for evaluating non-formula-like queries, typically queries which provide static information about the computation graph. The second area of the window displays information about the specification and the status of the tool, such as the name of the modes in the specification, and the file in which the specification resides. The third area is an output area for printing output from the various commands. For example, in the figure, the results of a `Write` command, which prints statistical information about the computation graph, appear in the output area.

Figure 1 includes a `Command` window for the `Inner Universal` command, which has been used to evaluate the safety property for the railroad crossing example. The formula has also been echoed to the output area of the main window. The tool takes the approach of allowing the user to fill in formula templates of instances of the various classes of formulas, rather than typing the formula in directly, to direct the user to formulas that can be verified. Figure 11 is

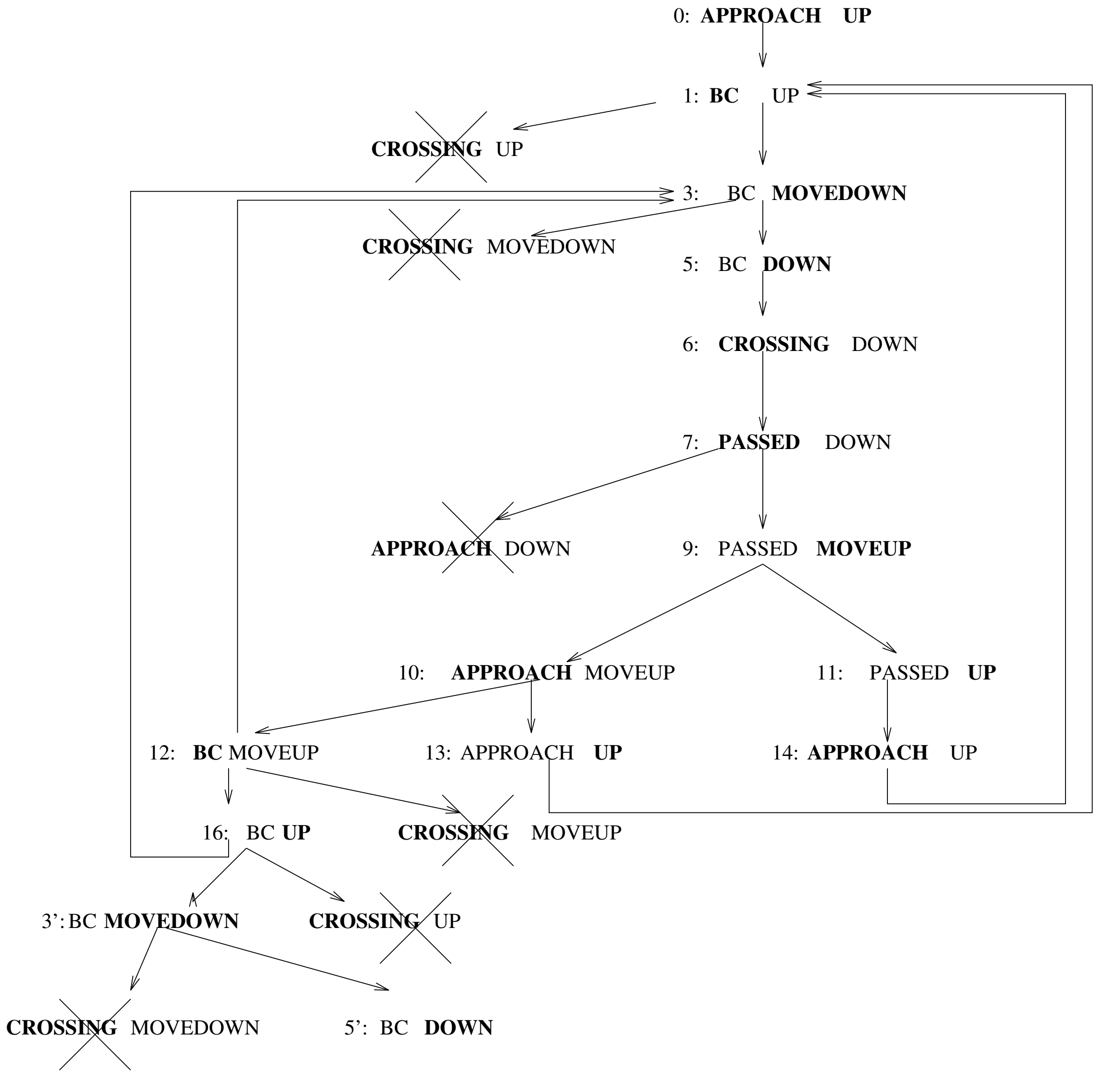


Figure 7: Incomplete Computation Graph

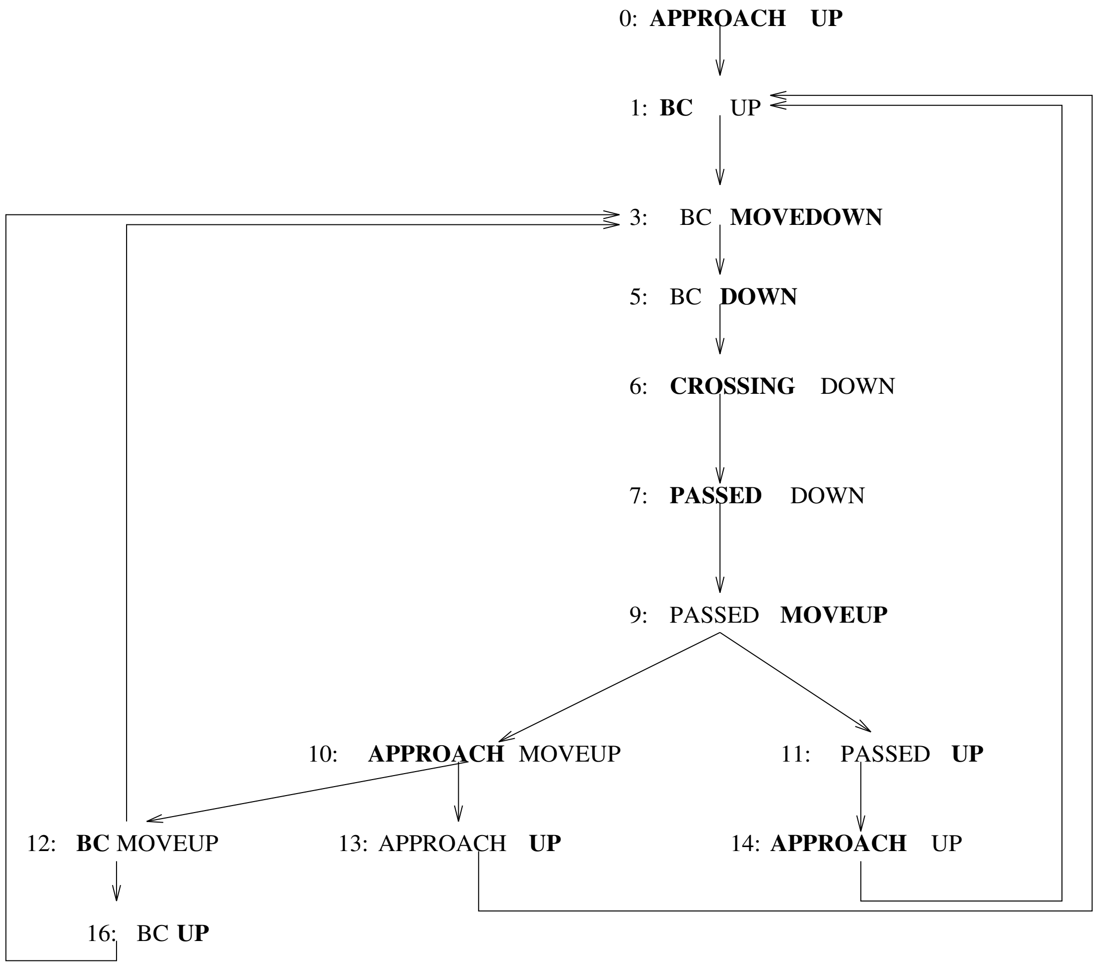


Figure 8: Computation Graph for the Railroad Crossing Example

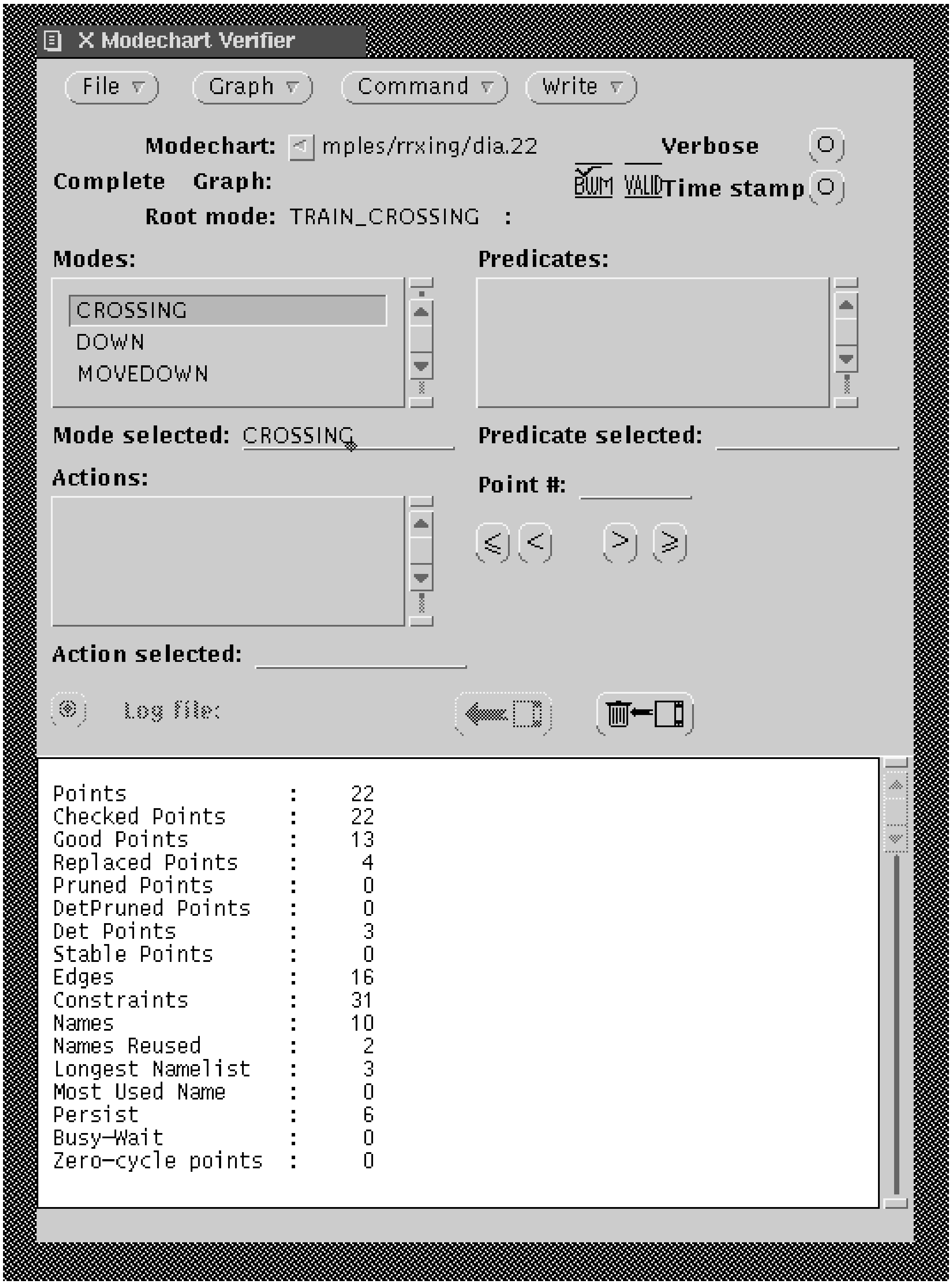


Figure 9: The Verifier Window

Figure 10: The Elapsed Time Window

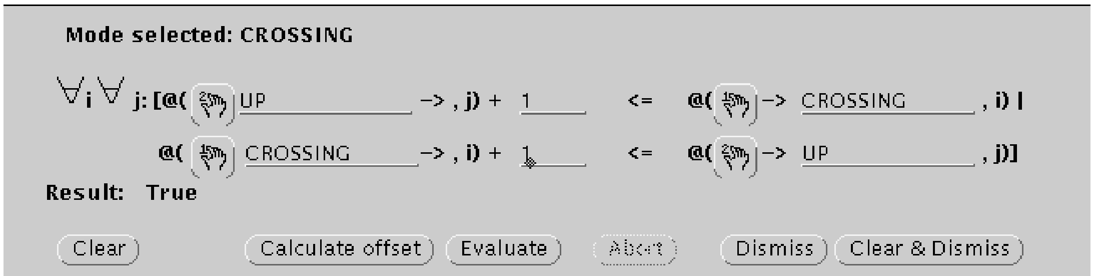


Figure 11: The Exclusive Modes Window

another `Command` window, this time for the `Exclusive Modes` command, which allows the user to specify two modes, and then checks to ensure that their activation intervals are mutually exclusive. These are both formula queries, although the `Exclusive Modes` query is perhaps more user friendly, in that it speaks to the mutual exclusion of modes, rather than timing relationships among events. In a similar vein, figure 10 is a third `Command` window, this time for the `Elapsed Time` command, which determines the minimum and maximum time a given mode may be active. In this case, rather than even express the query as a formula, the query is directly expressed in terms of an intuitive notion of mode activation duration. Further information about the tool can be found in [32].

5 Concluding Remarks

The verification technique proposed in this paper is based on model checking. It requires the construction of a computation graph for a given Modechart specification. The computation graph, which is a finite representation of possible system behaviors, can be used to determine whether the system satisfies the properties of interest. 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 or model checking. In particular, it may be impractical to generate the entire graph for a complex Modechart specification. The integration of the verifier described in this paper into the Modechart Toolset (MT) during the last year is making a number of practical options possible. MT supports the formal specification of real-time behavior in Modechart and provides software engineers with a number of analysis techniques including formal verification, static completeness and consistency checking, and specification simulation. Furthermore, a software engineer now has the necessary support tools to combine various analysis techniques while designing a system. For example, it is possible to guide the toolset to generate execution simulations for a particular scenario and then invoke the verifier to demonstrate a property from a particular generated by the simulator. A natural extension of the proposed approach is to investigate the verification of properties by constructing a partial graph.

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 last few years have seen considerable progress in the development of formal methods for representing and reasoning about time-critical systems. two principal types of languages, *automata-theoretic* and *process-algebraic*, have been developed for describing systems and their behavior. Each of these two types of language has augmented existing models of untimed systems with temporal constructs to capture the timing behavior of systems. Some representative examples of automata-theoretic languages for real-time systems are those of [20, 25, 2, 3, 24, 19, 10]. Some representative examples of process-algebraic languages for real-time systems are those of [12, 21, 31]. Also, several researchers [18, 27], have devised dual language approaches. Such an approach provides a front-end language, sometimes graphical, that is intuitive and easily understood by developers. Specifications written in the front-end language are translated to another language (usually state-machine-based) that is more amenable to formal analysis. The approach described in this paper is a dual language approach.

Considerable progress has also been made in the past few years in the development of formal methods for verifying real-time system correctness. In *model checking* methods, all of the *reachable* system states (i.e., those that can arise in an execution) are checked for violations of specified safety properties. Since real-time systems typically have infinitely many states, clever ways of aggregating state information are needed to enable exploration of all possible execution paths. Extension of model checking methods to real-time systems have been proposed in [3, 33] and an earlier version of this paper [18]. The advantage of model checking methods, such as the one described in this paper, is that they are amenable to mechanical procedures. Proof methods based on *process algebras* rely on descriptions of systems and requirements in a process algebraic language. Both the system model and the requirements are represented as algebraic expressions. The proof method relies heavily on algebraic identities to prove equivalence between algebraic expressions, combined with some mapping proofs. Development of mechanical proof methods for for process algebras is still preliminary. To evaluate the effectiveness of these reasoning techniques, prototype tools have been developed, such as those in [6, 5].

Although the approach proposed in this paper is similar to reachability analysis of timed-petri nets, e.g., [30, 35, 29, 26], there are several major differences. 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. Furthermore, 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 [30], 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 truth values to predicates.

Acknowledgement

We wish to thank Paul Clements, Connie Heitmeyer, Bruce Labaw, Ann Rose and others at the Naval Research Lab for their effort on the Modechart Toolset. We also wish to acknowledge Alex Ho, at the University of Texas, for his work on the graphical interface for the mechanical verifier.

References

[1] Thomas A. Alspaugh, Stuart R. Faulk, Kathryn Heninger Britton, R. Alan Parker, David L. Parnas, and John E. Shore. Software requirements for the A7-E aircraft. Technical Report NRL/FR/5530-92-9194, Naval Research Laboratory, August 1992.

[2] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Prodeedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, June 1990.

[3] R. Alur and D. Dill. The theory of timed automata. In *Proceedings of REX Workshop "Real-Time: Theory in Practice"*, pages 45–73, Mook, The Netherlands, June 1991. Springer-Verlag. volume 600 of Lecture Notes in Computer Science.

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

[5] D. Clarke, I. Lee, and H. Xie. Versa: A tool for the specification and analysis of resource-bound real-time systems. Technical Report MS-CIS-93-77, University of Pennsylvania, Department of Computer and Information Science, September 1993.

[6] R. Cleaveland, J. Parrow, and B Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *TOPLAS*, 15:36–72, 1993.

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

[8] D. Craigen, S. Gerhart, and T. Ralston. An international survey of industrial applications of formal methods, volume 1 study methodology. Technical Report PB93-178556/AS, Tech. Report 5546-93-9581, US Naval Research Laboratory, Washington, DC; Tech. Report Info-0474-1, Atomic Energy Control Board of Canada, Ontario, National Technical Information Service, Springfield, VA, 1993.

[9] D. Craigen, S. Gerhart, and T. Ralston. An international survey of industrial applications of formal methods, volume 2 case studies. Technical Report PB93-178564/AS, Tech. Report 5546-93-9582, US Naval Research Laboratory, Washington, DC; Tech. Report Info-0474-2, Atomic Energy Control Board of Canada, Ontario, National Technical Information Service, Springfield, VA, 1993.

[10] E. Emerson, A. Mok, A. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In *Workshop on Automatic Verification of Finite-state Systems*, Grenoble, 1989.

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

[12] R. Gerber and I. Lee. A resource-based prioritized bisimulation for real-time systems. Technical Report MS-CIS-90-69, University of Pennsylvania, Department of Computer and Information Science, September 1990.

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

[14] K. L. Heninger. Specifying software requirements for complex systems. *IEEE Trans. Software Engineering*, SE-6(1):2–13, January 1980.

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

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

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

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

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

[20] Leslie Lamport. The temporal logic of actions. Technical Report 79, Digital Systems Research Center, December 25 1991.

[21] I. Lee, P. Brémond-Grégoire, and R. Gerber. A process algebraic approach to the specification and analysis of resource-bound real-time systems. Technical Report MS-CIS-93-08, January 1993. To appear in Proc. of IEEE, Jan. 1994.

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

[23] N. G. Leveson and J. L. Stolzy. Analyzing safety and fault tolerance using time petri nets. In *TAPSOFT: Joint Conference on Theory and Practice of Software Development*, 1985.

[24] N. Lynch and H. Attiya. Using mappings to prove timing properties. *Distrib. Computing*, 6(2):121–139, 1992.

[25] Z. Manna, O. Maler, and A. Pnueli. The theory of timed automata. In *Proceedings of REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lecture Notes in Computer Science*, pages 447–482, Mook, The Netherlands, June 1991. Springer-Verlag.

[26] P.M. Merlin and D.J. Farbar. Recoverability of communication protocols-implications of a theoretical study. *IEEE Trans. on Communications*, pages 1036–43, Sept. 1976.

[27] J. Ostroff. *Temporal Logic for Real-Time Systems*. John Wiley and Sons Inc., 1989.

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

[29] C. Ramchandani. Analysis of asynchronous concurrent systems by timed petri nets. Technical Report TR 120, Project MAC, MIT, Feb. 1974.

[30] R.R. Razouk and C.V. Phelps. Performance analysis of timed petri nets. In *Proc. of 4th International Workshop on Protocol Spec, Verf and Testing*, June 1984.

[31] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theor. Comp. Sci.*, pages 249–261, 1988.

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

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

[34] D.A. Stuart, A.K. Mok, and F. Jahanian. A formal method for verifying real-time properties of modechart specifications. Technical report, Technical Report, University of Michigan, November 1994.

[35] W.M. Zuberek. Timed petri net and preliminary performance evaluation. In *7th Annual Symposium on Computer Architecture*, pages 88–96, 1980.