Implementing a Verifier For Real-Time Systems†

Douglas A. Stuart

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

Abstract

The SARTOR project has as one of its goals the development of an environment for the development of correct real-time systems. Modechart is a specification language for real-time systems developed as part of this project. Verify4 is an implementation of a verifier for certain classes of properties of systems specified using modechart. This paper describes the program verify4 and addresses implementation issues surrounding three of the key algorithms used in the program.

1. Introduction

The SARTOR project,[Mok 85], has as its ultimate goal an environment for the development of correct real-time software. There are many research threads under SARTOR. Software has already been developed to do timing analysis of assembly language and c source code. Other projects are covering real-time rule based decision systems, data flow computation, and scheduling. There is also an attempt to provide a framework for the specification and verification of real-time systems.

The verification effort has as its foundation two elements: a logic for real-time systems (RTL), and a specification language (Modechart). Real Time Logic first appeared in [Jahanian & Mok 86a] , and is an extension of Presburger arithmetic. Modechart first appeared in [Jahanian et al 88b] , and was inspired by Harel's statechart ([Harel 86]). A method for verifying properties of systems specified in modechart was described in [Jahanian & Stuart 88] , and its implementation is the subject of this paper.

The rest of this paper is organized as follows. Section 2 describes an example that will be used throughout the paper. Section 3 gives a brief introduction to RTL and Modechart. Section 4 applies the tool to the example of section 2, and section 5 provides additional information about the tool. Section 6 is a review of the material in [Jahanian & Stuart 88] . Sections 7, 8, and 9 each concentrate on one aspect of the problem and how the present implementation addresses it. Section 10 presents several conclusions and areas for future development.

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

2. A Real-Time Railroad Crossing

The following real-time railroad crossing example will be used to illustrate and motivate the remainder of the paper. This example is based on one appearing in [Leveson & Stolzy 85] , and used in[Jahanian & Stuart 88] .

The gate at a guarded railroad crossing is to be software controlled, and since the gate cannot control the train, a real-time solution is needed. There is an early warning signal at a distance from the crossing that gives notice to the gate controller that a train is approaching, and it is known that it takes the train at least 300 time units to reach the crossing from the signal. It is also known that the time required to lower the 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 signal. These are the parameters of the system specification. The problem is to ensure that the system will prevent cars from being run over, by always having the gate down when a train is in the crossing.

3. Modechart and RTL

A specification language, a logic for specifying properties of specifications, and a method for verifying such properties are needed to exhibit a solution to the problem posed by the railroad crossing example. The specification language and logic used in SARTOR are Modechart and RTL. **Verify4** is an implementation of a method for verifying certain types of RTL formulas with respect to Modechart specifications . This section provides a brief introduction to Modechart and RTL. More details and additional features can be found in [Jahanian et al 88b] and [Jahanian & Mok 86a] .

Modechart is a graph based specification language, with its semantics given in RTL, which was developed to allow easier representation of the structure inherent in most systems. RTL provides a logic for reasoning about real-time systems.

3.1. Modechart

Modechart is a graph based specification language, with its semantics given in RTL, that attempts to capture the structure of the system being specified. Modechart is based on statechart ([Harel 86]), and can be viewed as representing the system as a hierarchical finite state machine. The key construct of Modechart is the mode, which is a representation of the control information of the system. There are three types of modes. Primitive modes have no internal structure. Serial modes contain a number of child modes, and are intended to capture the notion of sequential execution. Parallel modes also contain a number of child modes, and represent parallel execution.

Modechart also allows for transitions between modes. Each transition also has a condition associated with it that indicates when it may or must be taken. Transition conditions come in two flavors. A triggering condition is an expression of the form $\bigvee/\bigwedge p_i$, where each p_i is either an event name or a set of mode names. The second type of condition is the lower/upper bound condition, and is denoted by a pair (r,d), where r denotes a delay, and d a deadline. A transition may be taken from a mode when one of its conditions is satisfied, and some transition exiting its source mode must be taken before its deadline expires. Figure 1 shows the modechart specification of a solution to the railroad crossing problem. The system is divided into two parallel modes, one to monitor the location of the next train, and one to control the movement of the gate. Entry of the mode BC (Before Crossing) indicates that the early warning signal has detected the train.

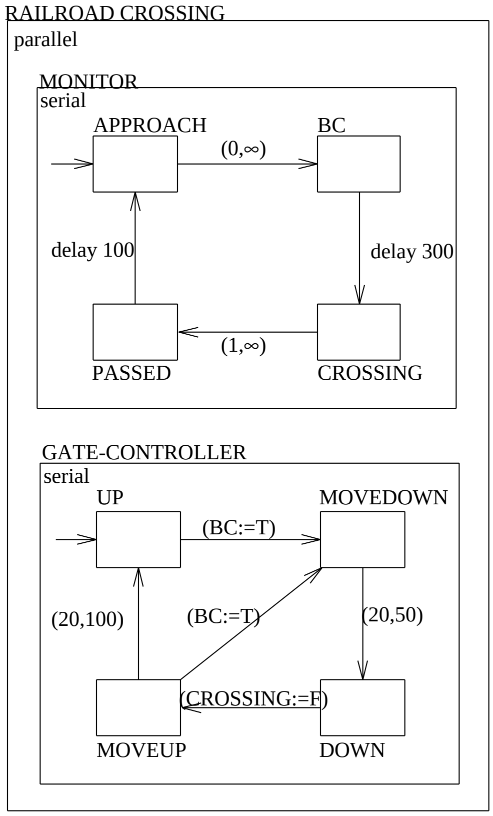

FIGURE 1.

3.2. RTL

RTL views a system as a set of events. An event is something that distinguishes a point in the computation of a system, and is instantaneous, since it is a temporal marker. A computation is an assignment of times (from the natural numbers) to event occurrences. An event occurrence is an instance of one of the events of the system.

RTL is a first order logic that is an extension of Presburger (natural numbers under addition) arithmetic. The notion of event occurrence is brought into RTL by the two argument occurrence function, denoted by '@'. The value of @(E,i) is the time of the ith occurrence of event E. Occurrence function applications, natural number variables, and constants form the atoms of RTL. RTL is completed by adding the usual relations, connectives and quantifiers of first order arithmetic.

Continuing the example, the safety property that the solution to the railroad crossing problem given in figure 1 must satisfy, is that whenever the train is in the mode CROSSING, the gate must be in the mode DOWN. In RTL, this may be written as

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

where (DOWN:=T) is the RTL notation for the event corresponding to entering the mode DOWN.

4. Verifying the Railroad Crossing Solution

Now that the problem and a candidate solution have been specified, it is necessary to show that the proposed solution actually satisfies the requirement.

Verify4 is a tool which implements the verification methodology presented in [Jahanian & Stuart 88] for verifying certain types of RTL formulas with respect to Modechart specifications. The tool will be discussed in section 5, and the method will be explained in detail in section 6. This section will illustrate the use of **verify4** to show that the modechart of figure 1 does in fact satisfy the desired safety property.

The tool takes a modechart specification as input and then generates a computation graph for the modechart. The tool then allows the user to choose among a set of queries it can answer about the modechart. The query *iu* will cause the tool to attempt to verify a certain type of formula with respect to the modechart by examining the computation graph. For the moment, a computation graph can be regarded as similar to a reachability or invariance graph.

To use **verify4** to show that the proposed solution satisfies the safety condition, the formula of section 3.2 must be modified slightly to fit one of the formula schemes that **verify4** recognizes. Since **verify4** can only recognize mode-entry events, and since the minimum time the train spends in CROSSING is one time unit, an equivalent formula that **verify4** can recognize is:

$$\forall i \exists j\ @((DOWN{:=}T),j)<@((CROSSING{:=}T),i) \wedge @((PASSED{:=}T),i)\leq@((MOVEUP{:=}T),j) \wedge @((CROSSING{:=}T),i)<@((PASSED{:=}T),i).$$

What follows is an abridged version of the output of the verifier on this input modechart and verifying this formula. The tool first declares that the modechart is valid, indicating a consistent specification. After printing some statistics about the graph, the actual graph is printed. Figure 2 is a graphic representation of this computation graph. In the output below, the new column indicates the mode entry event to which the point corresponds. In figure 2, this is indicated by italicizing the appropriate mode name. The next command, *iu*, verifies the formula above.

```
The modechart is valid.

 verification option ? print

Points           :    22
Good Points      :    13
Replaced Points  :     4
Edges            :    31

Clean or Long graph output? clean

Root Mode :  TRAIN-CROSSING
pt# status new ptname
             TRAIN     GATE
  0   G    0   APPROACH  UP
  1   G    1   BC        UP
  3   G    2   BC        MOVEDOWN
               .
               .
               .
 16   G    2   BC        UP

 verification option ? iu

enter the intervals
TRAIN, CROSSING
TRAIN, PASSED
GATE, DOWN
GATE, MOVEUP

The formula is valid.
```

5. The Tool Verify4

Verify4 is one of several first generation programs and tools related to modechart that have been implemented as part of the SARTOR project. The tool

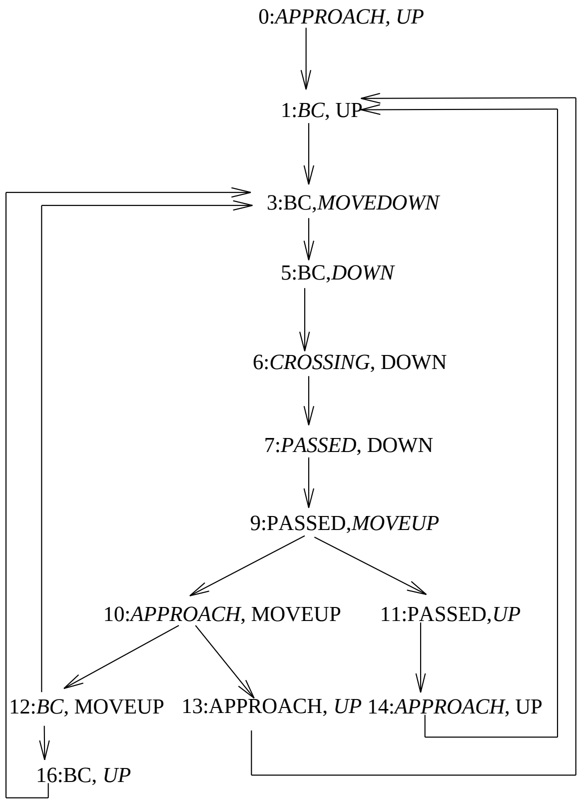

FIGURE 2

chart provides a graphical interface for creating, displaying, and revising modecharts. There are also programs for generating text output and for generating the RTL formulas that provide the semantics for a modechart.

Verify4 takes as input a file produced by **chart** containing a modechart. It then generates a computation graph from the modechart using the algorithm described in the next section. Next it checks to ensure that the modechart is consistent. The user is then presented with a list of seven alternative functions that the verifier can perform. These fall into three categories. The first category consists of verifying formulas as described in the next section. The second category includes other queries about the computation graph. The last category deals with output options.

The first category allows the user to check the validity of formulas that match one of four formula schemes. The first two,

$$\forall i\ @((M{:=}T),i)+cR@((M{:=}T),i+1),$$

where R is either < or >, are instances of formulas that represent the minimum and maximum separation of related events. The second two,

$$Q\ @((M_3{:=}T),j)<@((M_1{:=}T),i) \wedge$$
$$@((M_1{:=}T),i)<@((M_2{:=}T),i) \wedge$$
$$@((M_2{:=}T),i)\le@((M_4{:=}T),j),$$

where Q is either $\forall i\exists j$ or $\forall j\exists i$, are instances of formulas that represent inclusion of intervals. In each case, after selecting the type of formula, the user supplies the parameters (mode names and constants), and the program returns the validity (or invalidity) of the formula, or an error message. The user also has the option of asking for supplementary information, such as the points that correspond to the events in the formula, the paths they are on, and the relevant distances.

The second category includes three different types of query about the computation graph. The first type is a variation of the separation formula above. The user is allowed to supply two mode entry events, and then the program evaluates the distances between each point associated with the first event and the next point associated with the second event to evaluate the inequality. The second query in this category allows the user to describe a point name, and the verifier identifies all reachable points in the graph that match that name. The user describes the point name by supplying mode names that must appear in the name of the point, as well as mode names that may not appear in the name of the point. The last query determines the minimum and maximum amount of time the system may spend in a mode.

The last category provides two fairly simple output operations. The first allows the user to select a file to which the output will be appended, although results, diagnostics, and prompts will be duplicated on the standard output. The second operation prints statistics about the computation graph and its construction, as well as offering two formats for printing the computation graph itself. The statistics include the number of points explored during graph construction, the number of points actually in the graph, the number of replaced points, and the number of edges in the data structure (which roughly corresponds to the number of timing constraints in force). The first graph output format lists only the points and edges in the computation graph. The second output format also prints the unreachable potential successors of the points in the graph, and those points that were replaced with equivalent points.

Fairly simple data structures were chosen to allow concentration on the new computation graph and verification algorithms, rather than using more sophisticated data structures. This was made possible by the imposition of restrictions on the input modechart, which, like the restriction of the verifier to the formula schemes to those discussed above, was done to allow concentration on the central algorithms in this first implementation.

There are two major restrictions on the the input modechart. It must have a very simple hierarchical structure. The root mode must be a parallel mode, its children serial modes, and its grandchildren must be primitive modes. The second restriction is that triggering conditions are restricted to being the occurrence of a single mode-entry event.

As a further comment on the restrictions on the input modechart, the hierarchical prohibition is the most severe. The modifications necessary to accommodate more general hierarchical structuring are mainly related to storing and accessing the points in the graph, since the name of the point can now be variable length. This makes it necessary to use a data structure for the point table that allows for variable length keys as well. The effects of hierarchy on the graph are handled like those of disabling transitions, as described in section 7, except that in this case disabling is not symmetric.

6. The Verification Methodology

As indicated in section 4, the heart of the verification method implemented by **verify4** is the computation graph. For non-real-time distributed systems reachability graphs are used to capture the behavior of a system, and reasoning is done about the graph, rather than the system.[Jahanian & Stuart 88] introduces computation graphs by applying the reachability graph idea to real-time systems specified using modechart. The differences between computation graphs and traditional reachability graphs are the result of the differences in models of computation necessary to reason about time, and the differences in the systems specified, that use time. This section will provide a more detailed introduction to computation graphs, and enlarge on the verification method implemented by **verify4** that makes use of them.

6.1. Computation Trees and Graphs

Computation trees are an intermediate step in the development of computation graphs. The edges of a computation tree represent mode transitions, and each vertex, (called a point), represents the occurrence of the events associated with the transition denoted by its in-edge. The root point represents the entry of all of the initial modes of the system. Each point is labeled by the modes the system is in at that point (the name of the point), the events that occur at the same time as those represented by the point (called the event set), and which modes are entered by the transition represented by the point's in-edge.

The successors of a point in the computation tree are determined in two stages. In the first stage, the potential successors of the point are generated. For each transition with source mode as part of the name of the point a potential successor is generated by that transition by applying it to the point, if the transition's condition is either a lower/upper bound condition, or a triggering condition enabled by the label of the point. Once the potential successors of the point are determined, the distances, defined below, among the potential successors are examined, to determine which potential successors are actual successors.

The actual successors of a point in the computation tree are those of its potential successors that are reachable. A potential successor is not an actual successor if there is some other potential successor that has a positive distance to it. Intuitively, in order to be an actual (immediate) successor to the point, no other potential successor must be subject to timing constraints that require its event to occur strictly before the event corresponding to the candidate actual successor. Note that this condition is exactly what is necessary to ensure that there are no positive cycles in any separation graph (since non-actual potential successors are not in the computation tree). The distances needed can be determined from the separation graph generated by the rooted path ending at the point.

Associated with each finite rooted path in the computation tree is a weighted separation graph, which reflects the timing constraints in force along the path. Figure 3 is the separation graph for the path 0-1-3-5-6-7-9-10-12-3 in the computation graph of figure 2. The points in the separation graph are the points on the path, together with the potential successors of the end point. The potential successors are necessary to represent the effects of timing constraints imposed by transitions that are pending at the end point of the path. The edges in the graph record the timing constraints in effect on that path, with positive weights indicating delays, and negative weights deadlines. The effect is that the heaviest path between points A and B, called the distance between A and B, is the earliest time B can happen with respect to A. A property of separation graphs of well formed modecharts is that there are no positive weight cycles, so this notion of distance is well defined.

There are several sources of edges in a separation graph. To begin with, corresponding to each edge in the path that generated the graph is an edge with weight 0, to reflect that descendants can not happen earlier than ancestors. There are also edges associated

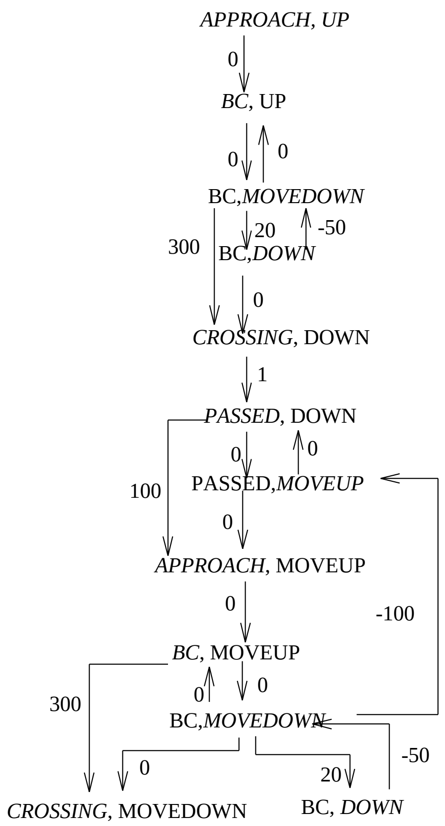

FIGURE 3

with transitions. If the transition associated with a point on the path has a lower/upper bound condition (r,d), then there is an edge in the separation graph with weight r from the point corresponding to the entry of the source mode of the transition (called the reference point for the transition) to the point, and a reverse edge with weight $-d$. If the transition has a triggering condition, then there is an edge in the separation graph with weight 0 from the point to its immediate ancestor (in the computation tree). The intent of this is that the triggering condition must have been satisfied by some previous transition, so the transition must have been taken at the same time as that transition, and therefore at the same time as the transition that resulted in its parent. This is independent of whether that transition was the one that enabled the transition in question. Effectively, a triggering condition is enabled by the events in the event set of the parent point, so the transition must be taken at the same time as the events in the parent's event set, so there is a 0 deadline to the parent (not directly to the point or points corresponding to the events that enable the transition).

It can be shown that if two points in a computation graph P and P′ have the same label, and the corresponding distances between each point and its potential successors, and among the successors, are equal, then the subtree rooted at P and the subtree rooted at P′ are isomorphic, and corresponding distances within the subtrees are equal. This condition, called the distance equivalence (deq) condition, allows all edges to P′ to be replaced by edges to P, and the subtree rooted at P′ to be deleted, without affecting the paths through the resulting graph, or the distances along them. It can also be shown that a finite graph can result after a bounded number of such transformations. Such a resulting finite graph is called a computation graph of the system.

These two results provide the basis for the computation graph construction algorithm. Each point is represented by its label, and an indication of its status, either explored, or unexplored. Initially, the graph consists of a single point corresponding to the root and marked as unexplored. The algorithm then proceeds to explore the points in the graph. An unexplored point is chosen for exploration, and its potential successors are generated. If an equivalent point is already in the graph, edges to the current point are replaced by edges to the equivalent point, and the current point is removed from the graph. If no equivalent point is found, the current point is marked as explored, and its actual successors are added to the graph as unexplored points.

6.2. Using the Computation Graph

Two basic classes of formulas that can be validated with respect to a modechart specification by analyzing computation graphs were identified in [Jahanian & Stuart 88]. Before they can be introduced, however, three preliminary definitions are necessary. An endpoint is an occurrence function application of the form @(E,i), where E is an event, i a natural number valued variable, and c positive integer. Two endpoints are related if they contain the same variable. An interval consists of several related endpoints. As an example, @(E,i+8) and @(E′,i-6) form an interval, while @(E,i+1) and @(E,j+1) do not.

The first class of formulas includes minimum and maximum separation of related endpoints. The formulas are of the form ∀v F or ∃v F, where F is a quantifier free formula such that each relation is of the form $e_1 + I \leq e_2$, where each e_i is an endpoint, all of the endpoints are related (and have variable v), and the I's are integers.

$$\forall j\ @(E,j)<@(E',j) \wedge @(E,j+1)-2>@(E',j)$$

is a formula of this type.

The second type of formula concerns the exclusion and inclusion of intervals. The formulas of this type are of the form

$$Q\ (e_1 + \mathrm{I} \leq e_3) \wedge (e_4 + \mathrm{J} \leq e_2) \text{ or}$$
$$Q\ (e_4 + \mathrm{I} \leq e_1) \vee (e_2 + \mathrm{J} \leq e_3)$$

where e_1 and e_2, and e_3 and e_4 are two distinct intervals, and Q is a prefix of two quantifiers that binds the variables in the two intervals. Examples include

$$\forall i \exists j\ @(A,j)<@(B,i) \wedge @(B',i)<@(A,j+1) \text{ and}$$
$$\forall i \forall j\ @(B',i)<@(A,j) \vee @(A',j)<@(B,i).$$

The first of these says that each pair of B and B′ events occurs between two consecutive A events, and the second says that A and A′ and B and B′ event pairs do not overlap.

The algorithms for the two classes of formulas are based on the same idea. Locate the points in the graph that can correspond to the endpoints in the formula in question, calculate the relevant distances, and then determine the truth or falsity of the instance of the formula. Based on the quantifiers involved, the truth of the whole formula is determined by the number of true (or false) instances.

This approach can only work if the number of correspondences is finite. In general this is not the case. For example, if there is a cycle in the computation graph that contains, for some interval, two of one point, and only one of another, then the number of pairs of corresponding points to be considered would be infinite, and the number of edges on the path between corresponding points could be infinite as well. The solution is to require that each cycle in the computation graph must contain the same number of points associated with each event in each interval in the formula. If this is true, the graph is said to preserve the formula. This resolves the problem, because given a point that may correspond to one of the endpoints in an interval, any point that is to correspond to one of the other endpoints must be a number of edges away from the given point that is bounded only by the size of the graph and the occurrence offsets involved. Since this results in a finite number of potential corresponding points for each point in the graph and a finite number of edges beteen corresponding points, the algorithm will terminate.

The problem of determining when a formula is preserved by a graph is the subject of section 8. The problem of actually finding the various sets of corresponding points is the subject of section 9. Before these two problems are addressed, section 7 finishes the discussion of computation graph generation.

7. Graph Construction Issues

The next three sections will focus on specific issues that arise in the implementation of the algorithms of section 6, and their realization in **verify4**. This section deals with three problems that must be resolved as part of the graph construction process. The first two problems, which have related solutions, deal with separation graph edges. The last problem is that of calculating distances.

In order to ensure that separation graphs accurately represent all timing constraints that are in force, some additional timing constraints may need to be applied to the actual successors of each point (which will have their effect in the separation graphs of any paths including them). If taking the transition that results in an actual successor would prevent another transition from being taken, by causing the source mode of the other transition to be exited, then that actual successor becomes subject to the same deadline as the disabled transition, in addition to any deadline of its own. As an example, consider the two transitions leaving mode MOVEUP in figure 1, one with condition (BC:=T), and another with condition (20,100), at point 12 in figure 2. Since both result in actual successors, the successor generated by the first transition would be subject to an additional deadline of 100 with respect to its reference point, as indicated in figure 3, and the point generated by the second transition would be subject to a deadline of 0 to to its parent. This reflects the fact that the first transition still must be taken within 100 time units of entering mode MOVEUP, and the first must be taken immediately upon the entry of BC (assuming its delay of 20 had been satisfied).

Secondly, to insure a proper inheritance of event sets, some actual successors may need to be split into two actual successors. If one of the actual successors of a point has a distance of 0 to that point, then each of the actual successors must happen at the same time as that point. Accordingly, they all inherit all of the events in the event set of their parent, to be added to the events associated with the individual actual successors. To ensure that they do occur at the same time as the point, all of the actual successors are subject to an additional deadline of 0 with respect to the point (Note that this is not strictly necessary, as the constraints imposed by the preceding paragraph will have the same effect.).

If all of the actual successors have a negative distance to the current point, then those that have a positive distance from the current point (happen strictly after), do not inherit any events from the event set of the current point. Those that have a distance of 0 from the current point must be split into two actual successors. One happens at the same time as the current point and inherits its event set and becomes subject to an additional 0 deadline as in the preceding paragraph. The other does not inherit the event set of its parent, and becomes subject to an additional delay of 1 from that point, to indicate that it happens strictly later than the current point.

The distances of interest in computation graph construction are the distances between the current point (or a candidate equivalent point) and its potential successors, and among those successors. These distances can be calculated incrementally from the same distances of the current point's parent. The technique is essentially the same as that of the iterative step in the normal all-pairs shortest-path algorithm. For points corresponding to transitions that were enabled at the current point's predecessor, the treatment of disabling transitions ensures that the path establishing the old distance still exists, so the only changes must arise from edges added in the process of making the current point an actual successor.

The effect of these can be calculated by first calculating new distances to and from the current point, and then comparing the old distance to the new distance to the current point plus the new distance from the current point to the destination point. The new distance to the current point is the maximum of the old distance and the old distance to the predecessor point plus any delay added due to splitting. The new distance from the current point is the maximum of the old distance, the new tree edge of delay 0 from the current point to the given point, and, if a deadline was added to the current point due to a disabled transition, the old distance from the disabled point to the the point of interest, and if a deadline edge was added due to splitting, the distance from the parent point to the point of interest. The distance between potential successors is now either the old distance, or the new distance from the point of interest to the current point plus the new distance from the current point to the destination point. For points which are the result of transitions not enabled at the predecessor point, the only edges to or from them in the separation graph go through the current point, so the distances to or from these points are the new distance to or from the current point plus the weight of the edge to or from the new point.

This algorithm is roughly linear in the number of distances to be calculated. Unfortunately, all of the distances must be stored, and can only be removed after the graph has been completed. The current program instead recalculates all distances from scratch when they are needed. It does this using the dual of the standard all-pairs short-paths algorithm, substituting MAX for MIN in the distance comparison. Since the all-pairs short-paths is correct for graphs with both positive and negative edge weights in the absence of negative cycles, the dual algorithm is correct for both positive and negative edge weights in the absence of positive cycles.

8. Preserving Formulas

This section, and the one that follows, address the two key tasks facing the verifier. The first task, determining whether or not the graph preserves the formula, is the subject of this section. One observation is that it is sufficient to check the simple cycles in the graph. It is easy to show that a computation graph preserves a formula if and only if all of the simple cycles of the computation graph preserve the formula. If the computation graph were an arbitrary graph, the problem would be exponential, since the number of cycles in the graph is also potentially exponential.

The hope for a more efficient algorithm rests on the structure of the graph. Since the graph is really a transformed tree, any cycle must contain at least one edge that resulted from an application of the deq condition. A further observation is that this edge must be an 'up' edge, in that it goes to a point at least as close to the root as the point it comes from. Again, since the graph began as a tree, this is well defined. Combining these three observations results in the following algorithm. For each edge which can potentially cause a cycle, traverse the edges in the reverse direction, doing a depth first search, until a cycle is found, or the root is reached, branching as necessary to cover all paths. Each up-edge can then be marked after it has been considered, and ignored when considering subsequent up-edges, since any simple cycles in which it appears have already been considered. If the 'bad' edges are comparatively rare, which seems likely to be the case in practice, the behavior of this algorithm should be comparatively good.

Although this algorithm seems likely to perform relatively well in practice, its complexity is still exponential. The answer seems to lie in the realm of heuristics. A simple heuristic that applies to the separation formulas that **verify4** supports is that every cycle preserves a single event. There are also other similarly simple heuristics that can be fairly useful. If all of the related events in an interval are associated with the children of a single serial mode, and the transitions within and from the mode are such that either all of the events must occur exactly once in a single cycle through the serial mode, or none of them occur, then the interval will also be preserved by any cycle. Likewise, given an event E and a transition that can not be disabled that has E as its triggering condition, E and any event associated with the transition will also be preserved.

The preceding heuristics have all focused on the structure of the modechart. There are also heuristics based on the structure of the computation graph. A simple example is the following. If all of the events in an interval are associated with points closer to the root than the destination of any up edge, then none of the events can appear in any cycle, and so the interval is preserved by the graph. A more useful but more complicated heuristic, is that if no destination or source point of any non-tree edge is associated with one of the events of the interval, and if the interval is preserved by each simple path between such points, then the formula is preserved. This last can be checked in time linear in the size of the graph, since the number of each event that has been seen since the previous replacement edge can be calculated by a preorder traversal of the subtree. This idea can also serve as the basis for a transformation to obtain a smaller graph to which the full preservation algorithm can be applied.

As an added complication, the question of preserving formulas is closely related to the problem of locating corresponding points, which is discussed in the next section. As an example of this, some of the algorithms for finding corresponding points require that all simple paths be examined. If this is the case, then checking whether the formula is preserved can be done at the same time, with little additional cost. This aspect of the problem will be discussed more in the following section.

9. Finding Corresponding Points

The validity of a formula with respect to a computation graph is determined by the collection of all (not necessarily simple) paths through the graph, points on these paths that can be instantiations of the endpoints of the formula, and the distances among the selected points on these paths. A consequence of the use of the deq condition in constructing the graph is that the distances among the points of interest are independent of the path from the root to the first of these points, so it is sufficient to examine a single instance of each subpath beginning at the first of the designated points and ending at the last of the designated points.

If the graph preserves a set of related endpoints, then each of the subpaths involved is finite and has length bounded by the size of the graph and the integer offsets of the endpoints. Since the subpath is finite, the distances can be calculated and the quantifier free part of the formula can be evaluated. Since there are only a finite number of correspondences between points and endpoints, and since the bound on subpath length is independent of the subpath itself, there are only a finite number of these (subpath, point-correspondence) pairs. Since the quantification is now effectively over a finite domain, the formula itself can be resolved as either a conjunction or disjunction over all of the subpaths, as appropriate. Formulas which involve two distinct sets of related endpoints present an added complication, and will be covered later.

The task remains of identifying the subpaths, and making the proper correspondences between the points on them and the endpoints in the formula. Notice that the same finite subpath could have more than one correspondence between points and endpoints, depending on the number of events corresponding to each endpoint that appears in the portion of the path before the subpath. The solution is the result of the following observation. If there is a path containing a given subpath and correspondence, then there is another path containing the same subpath and correspondence where the path from the root to the first of the corresponding points is a simple path. The reason for this is that since each cycle contains equal numbers of each of the related endpoints, deleting cycles before the subpath will not affect the correspondence.

This provides the basic algorithm. Consider all rooted simple paths in the graph, and then on each such path, for each occurrence of the first event of a set of related endpoints, generate enough of each path from that point to reach all of the other corresponding points. For each of the first points, the number of points with that label preceding it on the simple path determines the correspondence. The other points are then located based on that correspondence adjusted for any offsets.

A simple example may illustrate this. Suppose the formula uses two endpoints, @(E,i+3), and @(F,i). Also suppose there is a rooted simple path from the root to a point corresponding to the event F, and to that point, there have been three other F points, and one E point. The subpaths and correspondences associated with this point and path can be generated by following each path from the point until the sixth E point is reached. Then on each of these paths, the correspondence will be between the fourth F point, and the seventh E point. The correspondence essentially uses the value of i of the simple path. This example also highlights a second issue. If all of the endpoints have positive offsets, then it may be that the point used to establish the correspondence will not be one of the points for which the distances are calculated.

As indicated before, the situation is more complicated if the formula involves two distinct intervals. These formulas can be divided into two groups. The first group, formulas of the forms ∃i∀j P, ∃i∃j R, and ∀i∀j F, where P is either interval inclusion or interval containment, R is interval exclusion, and F is interval containment, are basically degenerate cases, can usually be decided by inspection, and will not be discussed further. The second group consists of formulas of the forms ∀i∃j P and ∃i∃j F, and ∀i∀j R, where where P, F, and R are as before. These are equivalent for the purpose of finding corresponding points.

The method for solving the corresponding point problem for the second group is similar to the problem for a single interval. The subpath is located for the related endpoints of the outer quantifier. Once this is done it is necessary to find the endpoints for the interval controlled by the inner quantifier. For endpoints farther along the path than the first point of the first interval, this is done in the same way as for the first interval. The difference is that for some formulas, a correspondence that used points closer to the root could satisfy the distance relationships. The problem is that the cycles that could be ignored because they did not affect the correspondence beyond the first point of the first interval can affect the correspondence before that first point. The solution is that if in finding one of the corresponding points for the second interval it is necessary to use points before the first point of the first interval, it is necessary to find the corresponding points on all of the cycles that can reach the first point of the first interval. This is basically the same thing that was done before, just in the reverse direction. Once the corresponding points are located, the distances are checked as before.

Another question is how many second intervals need to be checked. Since any two points corresponding to the same event must be separated by a positive distance, eventually either the root mode will be reached, so on that path no satisfying inner interval can be found, or the separations will be such that if the choices so far examined for the inner interval do not satisfy the distance condition, then no other choices will either. For example, if the formula is

$$\forall i \exists j\ @(E1,j)<@(E3,i) \wedge @(E4,i)<@(E2,j),$$

then no second intervals where the E1 point is after the E3 point, or the E2 point is before the E4 point, could possibly satisfy the distance formulas.

As was mentioned in the preceding section, since these algorithms require checking all simple paths in the graph, it may be simplest to check for the preservation of the formula at the same time. The only disadvantage is that in that case, the verifier must continue to use the simple paths algorithm for checking preservation even if enough paths have been checked to decide validity if preservation were guaranteed.

As is the case with the preservation problem, heuristics can also play a role here, and are used in **verify4** to simplify the problem, and reduce the number of path correspondence pairs to check. The simplest of the heuristics is that if an interval contains only events that are associated with a single mode, then the correspondence problem is trivial. For any point in the graph, on any path, the corresponding points are just the adjacent points associated with the same event, that are separated by the appropriate number of points. **Verify4** uses this to check the validity of the separation formula by finding each point in the graph associated to the mode entry event, and then for each one, the corresponding point is just the next such point.

A slightly more complicated heuristic is used to simplify the corresponding point problem for the containment formulas. If the computation graph is required to preserve formulas on simple paths as well as on cycles, then since the offsets on all of the endpoints in the formula are zero, the correspondence is also simple. The first correspondence on the path is between the first occurrence of the relevant points, the second correspondence is between the second points, and so on. Since both cycles and simple paths preserve the formula, and the offsets are all zero, all of the corresponding points can be found on the simple path. The addition of the predicate requiring that the contained interval have a positive duration also forces the points of any containing interval be on the same simple path as the interval being contained, otherwise the other inequalities could not be satisfied. This means that all that is necessary to do is check the corresponding points on the simple paths, and no other paths need be generated.

10. Conclusions

The implementation of **verify4** served two main purposes. It demonstrated that the algorithms described in [Jahanian & Stuart 88] could be implemented. Secondly, it highlighted and resolved some of the ambiguities of those algorithms. It also had several unlooked for benefits. In the process of using and developing the program, it became clear that there were other uses for the computation graphs than just verifying the types of formulas described in the previous paper, foremost among these are reachability and elapsed time queries supported by **verify4**.

Another result of the implementation of **verify4** has been a better understanding of what can and should be required of its successor. In addition to incorporating most of the algorithms described in this paper, and being expanded to cover all modecharts, the next generation of verifier should also be integrated more fully with the whole suite of next generation modechart tools. Among those currently under development are a standardized interface and storage structure, replacements for **chart** and the formula generator, and a new non-graphics interface.

Bibliography

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

[Jahanian & Mok 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 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.).

[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).

[Mok 85]. A.K. Mok, ‘‘SARTOR-a Design Environment for Real-Time Systems,’’ *Proc. 9th IEEE COMPSAC*, pp. 174-181 (Oct. 1985).