# Simulation-Verification: Biting at the State Explosion Problem

Douglas A. Stuart, Monica Brockmeyer, *Member*, *IEEE*,
Aloysius K. Mok, *Member*, *IEEE*, and Farnam Jahanian, *Member*, *IEEE*

**Abstract**—Simulation and verification are the two conventional techniques for the analysis of specifications of real-time systems. While simulation is relatively inexpensive in terms of execution time, it only validates the behavior of a system for one particular computation path. On the other hand, verification provides guarantees over the entire set of computation paths of a system, but is, in general, very expensive due to the state-space explosion problem. In this paper, we introduce a new technique: Simulation-verification combines the best of both worlds by synthesizing an intermediate analysis method. This method uses simulation to limit the generation of a computation graph to that set of computations consistent with the simulation. This limited computation graph, called a simulation-verification graph, can be one or more orders of magnitude smaller than the full computation graph. A tool, XSVT, is described which implements simulation-verification graphs. Three paradigms for using the new technique are proposed. The paper illustrates the application of the proposed technique via an example of a robot controller for a manufacturing assembly line.

**Index Terms**—Real-time systems, formal methods, specification, verification, timing constraints, Modechart, requirements analysis, simulation.

---

## 1 Introduction

Simulation and verification are two conventional techniques for analysis of specifications of real-time systems. While simulation is relatively inexpensive in terms of execution time, it only validates the behavior of a system for one particular computation path. On the other hand, verification provides guarantees over the entire set of computation paths of a system, but is, in general, very expensive due to the state space explosion problem. These two techniques, then can be viewed as opposite ends of a continuum, trading guarantees for cost in moving from verification to simulation. In this sense, there may be techniques which lie between the two which provide more guarantees than simulation, but at lower cost than verification. In this paper, we introduce a new technique: *Simulation-verification* combines the best of both worlds by synthesizing an intermediate analysis method.

Simulation-verification is a technique that combines simulation with verification and can be used as a basis for new paradigms for the development of correct specifications. This paper presents several techniques for using simulation-verification. Simulation-verification can be used as an aid to specification understanding by allowing a user or designer to navigate through behaviors of a specification. Simulation-verification can also be used to verify properties which can be cast in CTL-like forms such as:

$$E\ path\ U\ (A\ property\ U\ frontier).$$

(This property states that there is some computation that follows *path* until some point where all computations agree with *property* until *frontier* is satisfied.) This can be particularly valuable if full verification is impractical due to state-space explosion, and *path* can be used to restrict the portion of the computation graph being generated. Simulation-verification has been developed in the context of the Modechart [24] specification language and uses simulation to limit generation of a *simulation-verification graph* to that set of computations starting with the simulation path. It has been implemented by augmenting the existing specification and analysis tools of the MT toolset for Modechart.

Modechart is a graphical specification language and represents a system as a collection of *modes*, representing the state of the system, and *transitions*, representing the control flow of the system. The MT toolset for Modechart consists of a user interface tool, a simulator, and a model-checking-based verification tool which represents all of the behaviors of a modechart specification by a computation graph. The approach taken by the simulation-verification technique is to use the simulator to generate a prefix of a computation of a specification and, then use the verifier to extend the prefix into a fragment of a *computation graph* with a fixed *frontier*. This combination of techniques uses simulation to narrow the focus of the verification effort to a subset of the computations of the specification, while using verification to examine all of the possible behaviors that could follow. Fixing a frontier limiting the generation of the resulting computation graph fragment further

- D.A. Stuart is with the Boeing Company, St. Louis, MO 63166-0516. E-mail: douglas.a.stuart@boeing.com.
- M. Brockmeyer is with the Department of Computer Science, Wayne State University, Detroit, MI 48202. E-mail: mab@cs.wayne.edu.
- A.K. Mok is with the Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712. E-mail: mok@cs.utexas.edu.
- F. Jahanian is with the Real-Time Computing Laboratory, Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI 48109-2122. E-mail: farnam@eecs.umich.edu.

*Manuscript received 15 June 1998; accepted 6 Jan. 1999.*
*Recommended for acceptance by L. Dillon.*
*For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 107007.*

0098-5589/01/$10.00 © 2001 IEEE

reduces the cost of the technique by limiting the generated fragment to that portion that is of interest in the context of the particular computation prefix.

The remainder of this paper will present this technique in more detail. Section 2 provides an overview of the Modechart language and introduces an example that will be used to illustrate the simulation-verification approach. Section 3 provides the details of the technique, while Section 4 introduces the *XSVT* tool which implements the technique as part of the MT toolset. Section 5 provides several contexts in which simulation-verification might be used. The applications of this technique are illustrated via a robot controller for a manufacturing assembly line. Section 6 describes related work, while Section 7 indicates some areas of current and future work.

## 2 Modechart Overview

This section provides an overview of the Modechart computation model and describes the Modechart language.

### 2.1 The Modechart Computation Model

A specification language such as Modechart provides a means of describing real-time systems in a precise way, and for formally showing that a specification in the language satisfies certain properties. Although some properties may be purely syntactic, such as the size of the specification, or the number of processes or agents, most properties of interest are semantic properties, that is, requirements on the behaviors of the specification. Before proceeding to a more detailed introduction to Modechart, the computation model that will be used to represent the computations of a Modechart specification must first be introduced.

Modechart regards a specification as a collection of events. The building block of a Modechart computation is the event occurrence. An event occurrence is a time-stamped event, representing the occurrence of the event at the indicated time. A computation of a Modechart specification is a sequence of sets of time stamped event occurrences, with all event occurrences in each set happening simultaneously. Since each specification is finite, there are only a finite number of events in each specification, so an infinite computation will contain an infinite number of occurrences of at least one event. The formal definition of what it means for a computation to satisfy a modechart can be found in [23].

Intuitively, a computation can be regarded as a sequence of sets of event occurrences, where all of the event occurrences in the same set have the same time-stamp. Since Modechart is primarily concerned with the timing properties of specifications, in the sense of properties that can be expressed in terms of the times of various event occurrences, the only ordering on the events of a computation considered by Modechart is the ordering introduced by the time-stamps on the individual event occurrences. As a consequence, event occurrences with the same time-stamp are truly considered to be simultaneous, neither is considered to precede the other. This feature of Modechart computations must be borne in mind when writing Modechart specifications, and some of the consequences of this will be seen in subsequent sections.

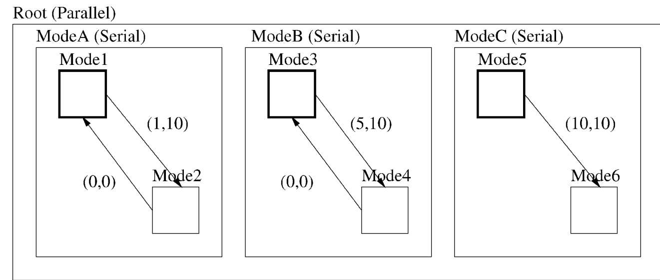


Fig. 1. Simple Modechart specification.

As a final note on computations of Modechart specifications, Modechart uses a discrete time model. This means that the events in a computation occur at integer times and, more importantly, with a fixed minimum separation between nonsimultaneous events. This is in contrast to continuous time, where event occurrences could have real time-stamps and, so, nonsimultaneous events could be arbitrarily close.

### 2.2 The Modechart Specification Language

As indicated in Section 1, Modechart is a hierarchical graphical specification language for real-time systems. The two basic elements from which Modechart specifications are constructed are *modes* and *transitions*. Modes represent the control state of the system being specified. Transitions represent the flow of control of the system being specified.

#### 2.2.1 Modes

Each mode in a specification, or modechart, must be one of three types. *Atomic* modes have no internal structure and are the basic building blocks of a specification. Referring to Fig. 1, the atomic modes are $Mode1$, $Mode2$, $Mode3$, $Mode4$, $Mode5$, and $Mode6$. Each represents a primitive control state of the system. Intuitively, a mode is *active* at a time when it has been entered but not yet exited.

*Serial* modes have internal structure. Each serial mode has at least one child mode and has a designated child mode which serves as an *initial mode*. A serial mode represents the sequential composition of its children, and the children of a serial mode are considered to be in series. Accordingly, at any time a serial mode is active, exactly one of its child modes must also be active. The initial mode of the serial mode is the mode that is considered to be entered when the serial mode itself is entered, unless entering the serial mode explicitly causes some other child of the serial mode to be entered. Referring again to Fig. 1, the serial modes in the specification are $ModeA$ with initial mode $Mode1$, $ModeB$ with initial mode $Mode3$, and $ModeC$ with initial mode $Mode5$, In the figures, the initial mode of a serial mode will be designated with a **bold** outline. In any computation of the specification of Fig. 1, whenever $ModeA$ is active, exactly one of $Mode1$ and $Mode2$ will be active.

*Parallel* modes also have internal structure. Each parallel mode has a number (possibly zero) of children. Although a parallel mode with zero children is syntactically legal, it is, in practice, equivalent to an atomic mode. A parallel mode represents the parallel composition of its children, and the children of a parallel mode are considered to be in parallel. Accordingly, at any time a parallel mode is active, all of its

child modes must also be active. Turning again to Fig. 1, there is only a single parallel mode, *Root*, which represents the entire specification. The children of *Root* are $ModeA$, $ModeB$, and $ModeC$.

### 2.2.2 Transitions

While modes represent the control state of a system, the control flow among control states is represented by *transitions*. Graphically, a transition is a directed edge between modes. Therefore, each transition has a source mode and a destination mode. Taking a transition from one mode to another represents control leaving the state represented by the source mode and being transferred to the destination mode and is considered to be instantaneous. Since all of the children of a parallel mode must be active when the parallel mode is itself active, transitions are only possible between children of a serial mode, that is, modes that are in series. However, since Modechart is a hierarchical language, a transition could have its source mode inside one child mode of a serial mode and its destination inside another child of the same serial mode. Accordingly, this requirement is stated more properly that the first common ancestor of the source and destination modes of a transition must be a serial mode, or the transition must be a self loop, and the parent of the source and destination mode must be a serial mode. Recalling the modechart specification in Fig. 1, there are two transitions among the children of $ModeA$ and $ModeB$, and one transition among the children of $ModeC$. There is a transition in $ModeA$ with source mode $Mode1$ and destination $Mode2$.

### 2.2.3 Events

Before turning to the question of when transitions can be taken, which is a semantic rather than syntactic issue, it is necessary to provide the link between the syntax of Modechart being described here and the computation model described above. Since the computation model views a system as a collection of events, it is necessary to identify the events associated with modes and transitions. Since an event is a point in time at which something occurs, a mode being active is not an event. The events associated with modes are the entry and exit of the mode. Syntactically, the entry event for mode $M$ is denoted by $\rightarrow M$, and the exit event for mode $M$ is denoted by $M \rightarrow$ . The event associated with a transition is the event of the transition being taken, known as the transition event. The transition event for a transition from mode $M1$ to mode $M2$ is represented as $M1 \rightarrow M2$. Every computation of a modechart will consist solely of mode entry and exit events and transition events. Accordingly, every computation of the modechart of Fig. 1 will consist of one or more occurrences of each of the 25 distinct events, the 20 entry and exit events of the 10 modes in the modechart, and the five transition events of the five transitions.

### 2.2.4 Mode Transition Conditions

Having defined the events of the specification, in order to define the computations of the specification, it remains only to establish when transitions among modes are taken. Conditions are associated with transitions to govern when they may be taken. Each transition has exactly one condition associated with it, and that condition is in disjunctive normal form. Each disjunct is either a *triggering condition* or a *timing condition*. (Disjuncts are indicated as follows: $\rightarrow Mode1 \mid \rightarrow Mode3$.)

A triggering condition is a conjunction (for example, $\rightarrow Mode1 \ \& \rightarrow Mode3$) of events and predicates, which is satisfied when all of the events in the triggering condition occur while all of the predicates are also true. Each conjunct of a triggering condition must be one of the following:

- The event $\rightarrow M$ is satisfied when mode $M$ is entered.
- The event $M \rightarrow$ is satisfied when mode $M$ is exited.
- The event $M1 \rightarrow M2$ is satisfied when the transition $M1 \rightarrow M2$ is taken.
- The predicate $M == true$ is satisfied if mode $M$ is active.
- The predicate $M == false$ is satisfied if mode $M$ is not active.
- The mode list predicate $\{(M1, \ldots, MN)\}$ is satisfied if any of the modes in the list are active.
- The before list predicate $(\{< M1, \ldots, MN)\})$ is satisfied if any of the modes in the list are active and have been active for at least one time unit.

Note that at the time instant when a mode is entered or exited, the mode is considered to be both active and inactive. It is important to consider this when constructing triggering conditions on transitions.

There are no triggering conditions in Fig. 1.

A timing condition consists of a *delay* and a *deadline*. The delay and deadline are both nonnegative integers, and the deadline must be greater than or equal to the delay. A timing condition with delay *r* and deadline *d* is denoted *(r, d)*. There are also a number of abbreviations for some special timing constraints. *(delay r)* is an abbreviation for $(r, \infty)$. *(alarm r)* is an abbreviation for $(r, r)$. *(deadline d)* is an abbreviation for $(0, d)$. Referring again to Fig. 1, the condition on the transition $Mode1 \rightarrow Mode2$ is the timing constraint *(1, 10)*, representing a delay of 1 and a deadline of 10, and the condition on the transition $Mode5 \rightarrow Mode6$ is *(alarm 10)*, representing a delay and deadline of 10. A timing condition is satisfied if the source mode of the transition with the timing condition has been active at least as long as the delay of the timing condition.

A transition may only be taken at a time at which one or more of the disjuncts of its condition are satisfied. The condition being satisfied is a necessary condition for the transition to be taken. The condition being satisfied does not force the transition to be taken. The deadline on a transition, on the other hand, imposes a requirement that the source mode of the transition must be exited at or before the time indicated by the deadline. In this sense, a triggering condition is considered to have a deadline of zero with respect to the time at which the condition becomes true. This is in contrast with the deadline (and the delay) of a timing condition, which is enforced with respect to the time at which the source mode is entered. Therefore, a deadline of infinity imposes no restriction at all and indicates that the transition need never be taken. Therefore, a timing condition $(r, \infty)$ indicates a transition that may be taken with complete discretion. The delay on a timing constraint and the condition of a triggering condition, can be regarded

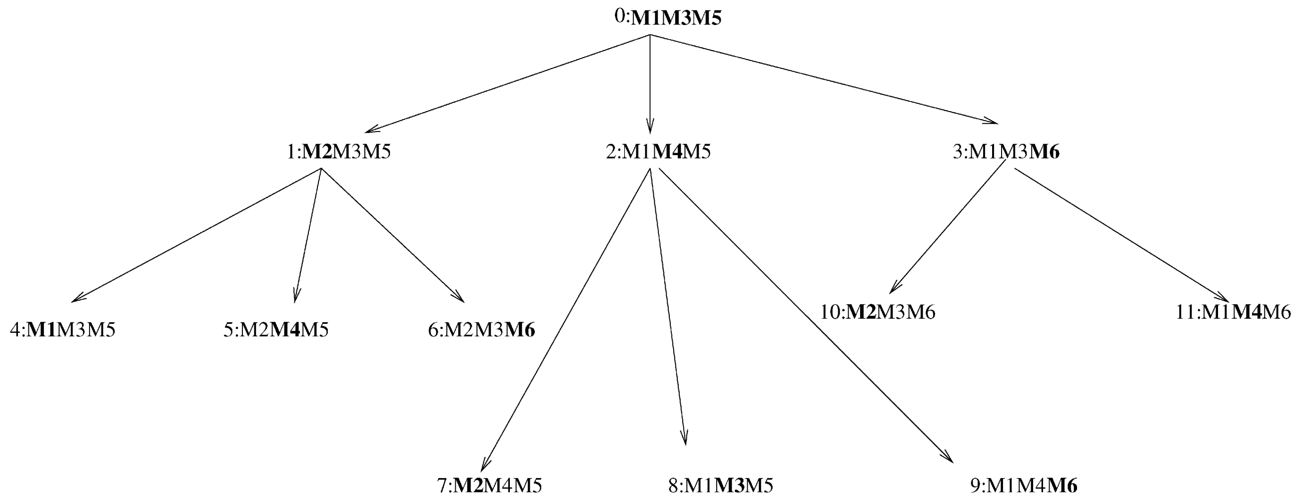


Fig. 2. Computation graph for simple Modechart.

as permitting the transition to be taken, while the deadline on either type of condition is a requirement for the source mode to be exited, by any means, not necessarily by taking the transition with the deadline. This distinction is only relevant, though, if there is more than one transition that exits the source mode.

#### 2.2.5 Root Mode

Finally, to complete this discussion of the semantics of modechart specifications, the *root mode* is the outermost mode in the specification and the only mode which has no parent. Each computation begins with the entry of the root mode at time 0, together with those modes that would be entered if the root mode were explicitly entered by a transition, that is, all children of each parallel mode entered, and the initial child of each serial mode entered. The remaining events in the computation are the result of transitions that are taken.

The root mode of the specification in Fig. 1 is *Root*. The modes initially entered are *Root*, $ModeA$, $ModeB$, $ModeC$, $Mode1$, $Mode3$, and $Mode5$.

### 2.3 Computation Graphs

Fig. 2 is a portion of the *computation graph* of the modechart of Fig. 1. The full computation graph contains 208 points even for this small specification. A computation graph is an unweighted directed graph representing all of the behaviors of the specification and is used for model-checking-based verification. The points in a computation graph represent the times at which events happen and are labeled by the configuration of the system after the occurrence of the event represented by the point. The edges in a computation graph represent the occurrence of transitions in the specification, so that the event represented by a point is the entry of the destination mode of the transitions corresponding to its in-edges.

The points in the graph in Fig. 2 have a unique numerical identifier. In addition, they are labeled with the names of the atomic modes that are active in the point. (The names of the modes are abbreviated in the figure.) The names of the nonatomic modes active at a given point may be inferred from the active atomic modes. The mode (or modes) in **bold** in the label of each point is the mode whose entry is represented by the point. For example, point 1 is labeled **M2**, *M3*, *M5* and represents entry of the mode $Mode2$ as a result of taking the transition $Mode1 \rightarrow Mode2$ while in point 0. This format will be used for simulation-verification graphs in Fig. 2.

The point corresponding to the entry of the root, or outermost, mode of the specification is called the root point of the computation graph. The rooted paths through the computation graph correspond to the sequences of events that can occur in computations of the specification. The actual computations of the specification are obtained by assigning times to the events in the sequence, or trace. The times assigned must be such that the resulting computation is consistent with the semantics of Modechart, so they must satisfy the constraints on the relative timing of events imposed by the semantics of transitions. For example, a transition governed by a timing condition *(r, d)* must occur no sooner than *r* time units after the entry event of its source mode and no more than *d* time units after that event. Since the timing constraints are enforced on traces to ensure that only those traces which can generate legal computations are represented in the computation graph, the points in the computation graph represent not only the occurrence of a particular event and configuration of active modes, but also a certain combination of timing constraints. Accordingly, there may be more than one point in a computation graph with the same label (i.e., the same modes are active), but which differ in the combined effects of the timing constraints in force on the paths through that point. For example, if the transition from Mode1 to Mode2 repeatedly occurs as early as possible, it will occur five times before the transition from Mode3 to Mode4 can occur and 10 times before the transition from Mode5 to Mode6 can occur. These 10 points will have the same label, but have different timing constraints in place since the fifth of these points will have a child where Mode4 is entered and the tenth will have a child where Mode6 is entered.

To construct the computation graph, the root point of the computation graph is generated first; then, the transitions from the modes active in the root point are examined to

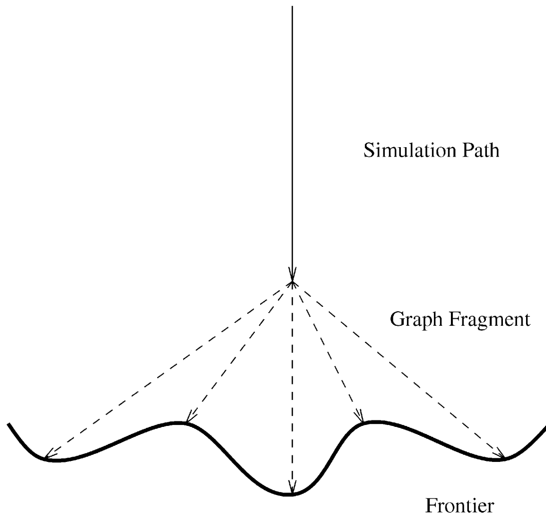


Fig. 3. Simulation-verification approach.

determine which transitions can be taken, either because the events necessary to satisfy a triggering condition have occurred, or because the constraints imposed by a timing condition result in a satisfiable set of timing constraints. Such reachable successors of the root point are added to the graph, and the process is repeated in a breadth-first fashion until no new points can be added to the computation graph.

Further details of Modechart, computation graphs, the MT toolset, and RTL (the query language for Modechart) are beyond the scope of this paper. Modechart is covered in detail in [24]. Computation graphs are described in [25]. A more thorough introduction to the MT toolset appears in [9] and the users guide is available as [36]. The logic RTL is introduced in [26].

## 3 Simulation-Verification: The Approach

The central concept of simulation-verification is to first simulate a specification, then extend the computation prefix generated by simulation as a computation graph. Since one of the objectives is to reduce the cost of the technique, and it is possible that extending the computation prefix may result in the full computation graph, a *frontier* described by a *termination condition* is used to terminate the generation of the computation graph fragment. Once the computation graph fragment is generated, it can be used to answer queries about those computations that agree with the computation prefix supplied by the simulator and are bounded by the frontier. Fig. 3 illustrates the approach. The remainder of this section will cover in detail five aspects of the simulation-verification process: the simulation path, termination conditions, generating the computation graph fragment, the semantics of simulation-verification graphs, and examining the generated graph.

### 3.1 The Simulation Path

The initial portion of the generated graph corresponds to a prefix of a computation, and can be considered to be generated by simulation, although this computation prefix can arise from some other source. The basic requirement is that this computation prefix consists of a sequence of *(event, time)* pairs, where the pair *(e, t)* indicates that event *e* occurred at time *t*. This time is measured, relatively, as an offset from the preceding event time pair in the sequence. As an example, Fig. 4 is one possible computation prefix for the modechart of Fig. 1, where only transition events are listed. This computation prefix corresponds to a computation of the path in the computation graph of Fig. 2 that begins at point 0 and continues through points 1, 4, and beyond.

- MODE1 → MODE2:1 /* time = 1 */
- MODE2 → MODE1:0
- MODE1 → MODE2:5 /* time = 6 */
- MODE2 → MODE1:0
- MODE1 → MODE2:1 /* time = 7 */
- MODE2 → MODE1:0
- MODE1 → MODE2:1 /* time = 8 */
- MODE2 → MODE1:0
- MODE3 → MODE4:0

Fig. 4. Simulation prefix for simple Modechart.

The simulation generated computation prefix is represented in the graph as a path, the *simulation path*, that begins the generated simulation-verification graph. There are other possibilities for representing this aspect of the simulation-verification process. One possibility is to modify the modechart specification so that the initial configuration of the modechart corresponds to the final configuration of the computation prefix. This technique has three main drawbacks. First, the configuration represented by the end of the computation prefix must reflect not only the modes active at the end of the prefix, but also the timing constraints in force. In particular, the end of the prefix may not correspond to the entry of all of the active modes, so that the timing constraints of the pending instances of transitions from modes active in the final configuration would have to be modified to reflect the time that has elapsed in the computation prefix. This would require duplication of modes with such transitions since future instances of the transitions should not be affected by any such modification. Similar problems exist for transitions with triggering conditions. The second drawback is that by not representing the entire computation prefix, any queries of the resulting graph can not refer to events in the prefix. Finally, there is an implementation advantage in representing the simulation prefix as a path in a computation graph since it allows all of the structure to be represented uniformly and simplifies the graph generation process. The advantage of using a modified specification is that if queries over the simulation path are not desired, then a potentially significant amount of space and time are used to store and generate information that is not used.

Finally, generation of the simulation path is accomplished by a simple modification of the standard computation graph generation algorithm. When deciding which transitions can

give rise to successors of a point in the graph, the computation prefix is used to exclude all of the events other than the next transition in the prefix. This also can be used for error detection and validation of the simulation path, by rejecting paths which use transitions that the computation graph algorithm does not identify as generating a reachable successor. The only other modification to the basic computation graph algorithm is that all points on the simulation path are automatically considered to be distinct since they are part of a specific trace with specific timing requirements.

### 3.2 The Frontier

The role of the frontier in simulation-verification is to limit the size of the graph generated. In the absence of such a constraint, it is possible that the entire computation graph could be generated. (Note that since the original computation graph is finite, the simulation-verification graph will also be finite.) For example, if the system being specified returns to its initial state after any activity, every trace, including any simulation path, will eventually return to the initial state and, from the start state, the entire computation graph will be generated. Accordingly, some means of restricting the generated graph is necessary if the cost of the technique is to be limited. However, if the technique is to be useful, the means of limiting the graph must be such that it still allows for a useful graph to be constructed. In addition, it should be described in a manner that is natural for the user. The approach we have taken is to allow the user to specify the frontier by means of a termination condition which is to be satisfied by points on the frontier.

Useful termination conditions, therefore, have two characteristics. First, they should result in a graph smaller than the full computation graph. Second, they should have some user-oriented justification. The current simulation-verification technique supports two types of termination conditions: Transition expression conditions and Event count conditions.

*Transition expression* termination conditions describe the frontier by a transition condition. This type of termination condition has the same syntax as transition expressions in Modechart (see Section 2), permitting disjunction of timing constraints and triggering conditions and conjunction of triggering conditions. The abbreviations *alarm n* and *deadline n* can also be used for transition expression termination conditions just as they can for ordinary transition expressions. A point is considered to be on the frontier if a hypothetical transition with the termination condition as transition condition could be taken from the point. For example, given the transition expression termination condition $(10, 10) \mid \rightarrow Mode6$, a point would be on the frontier if 10 time units could elapse from the end of the simulation path to the point, or if the event $\rightarrow Mode6$ occurs at the point. Using the same syntax as transition expressions provides flexibility and also allows the user to use simulation-verification to test possible transition conditions to determine when they would be taken.

*Event count* termination conditions describe the frontier by a set of *(event, count, relation)* triples, where *event* is an event of the specification, *count* is a nonnegative integer, and *relation* is an arithmetic relation. A point is considered to be on the frontier and its successors are not generated, if, along the path to the point from the end of the simulation path, for each triple *(e, c, ~)*, the number *d* of events *e* is such that *c~d*. That this type of termination condition could be of interest to the user is clear. It allows the user to specify termination conditions such as $(\rightarrow Mode2, 2, =)$, which would cause graph generation to halt along each path at the second point corresponding to the event $\rightarrow Mode2$. The one drawback of this type of termination condition is that it does not guarantee termination. The combination of constraints could be inconsistent, for example, $\{(\rightarrow Mode2, 2, <), (\rightarrow Mode2, 2, >)\}$. Alternatively, one or more of the designated events may not occur a sufficient number of times along some path. This issue can be resolved either by requiring the user to guarantee that the termination condition will, in fact, ensure termination, or by augmenting such a termination condition with a timing constraint as is the case for transition expression termination conditions.

### 3.3 Generating the Graph to the Frontier

The next step in the simulation-verification process is the generation of the computation graph fragment from the end of the simulation path to the frontier. This process is essentially the same as constructing the standard computation graph. The only difference between construction of a standard computation graph and construction of the computation graph fragment segment of a simulation-verification graph is that the termination condition must be evaluated at each point, with graph construction halting on each path independently when the termination condition is satisfied on that path. Unlike the simulation path portion of the simulation-verification graph, points in the computation graph fragment portion are not automatically distinct. If two points are generated that are equivalent, they may be collapsed into a single point. The only requirement is that in addition to the standard computation graph equivalence requirements of identical labels and equivalent timing constraints, the points must be equivalent with respect to the termination condition as well.

### 3.4 Semantics of the Simulation-Verification Graph

The preceding sections introduced the components of simulation-verification graphs: the simulation path, the computation graph fragment, and the frontier and termination condition. This section will introduce the simulation-verification graph generation algorithm and give a brief overview of the semantics of simulation-verification graphs.

The simulation verification graph generation algorithm is given in the following algorithm. This algorithm is almost identical to the standard computation graph generation algorithm of [25]. It differs only in the addition of steps which restrict the graph to the events on the simulation path during the simulation path phase (*phase = "simpath,"* lines 8-13) and other steps (line 20) which enforce the termination condition. Note that checking for an equivalent point only takes place while generating the computation graph fragment. The equivalence condition is satisfied when two points have the same label and the same set of timing constraints (line 15).

**Algorithm 1.** Simulation-Verification Graph Construction.
**Input:** A Modechart $\mathcal{M}$, a simulation path $\mathcal{S}$, and a termination condition, $\mathcal{T}$.
**Output:** A Simulation-Verification graph, $\mathcal{G}$.

1. Construct $P_0$, the root point in the simulation-verification graph, $\mathcal{G}$.
2. Designate $P_0$ an unexpanded point.
3. Set *phase* = *"simpath."*
4. **repeat**
5. Choose an unexpanded point, $P_m$.
6. Generate the potential successors of $P_m$.
7. Mark as unreachable those potential successors of $P_m$ which are not actual successors.
8. **if** *phase* = *"simpath"* **then**
9. Mark as unreachable those actual successors of $P_m$ that do not correspond to the next event on the simulation path, $\mathcal{S}$.
10. **if** the simulation path is exhausted **then**
11. Add an artificial end of simulation event $Q$ to the event set of the actual successors of $P_m$.
12. Set *phase* = *"cgfrag."*
13. **end if**
14. **else**
15. **if** there is an unexpanded point $P_n$ such that $P_m \approx P_n$ **then**
16. Replace all edges to $P_m$ with edges to $P_n$.
17. Delete $P_m$ and its actual successors.
18. **end if**
19. **else**
20. **if** $P_m$ satisfies $\mathcal{T}$ **then**
21. Mark $P_m$ as on the frontier.
22. Mark $P_m$ as expanded.
23. **end if**
24. **end if**
25. **until** all points are either expanded or on the frontier.

Having generated the simulation-verification graph, we now turn to its semantics. What follows mirrors the standard semantics of computation graphs as described in [25], to which the reader is referred for more details. A *trace* of a simulation-verification graph is a rooted path in the graph. A *constrained trace* of a simulation-verification graph is a trace together with a set of timing constraints which are imposed upon the events represented by the points on the trace. These include all of the standard timing constraints from [25], together with the timing constrains on the points on the simulation path represented by the *time* component of the *(event, time)* pairs which make up the simulation path. These constraints, which ensure that the trace conforms to the timing and trigger conditions of the underlying specification, are summarized in the Appendix. A computation of a constrained trace is an assignment of (integer) times to the events of the trace consistent with the timing constraints in force on the trace. The set of computations of a simulation-verification graph is then the set of all computations of all constrained traces of the graph. Note that since the simulation graph ends at points along the frontier and such points will generally have pending transitions with finite deadlines, the computations of a simulation-verification graph are not computations of the underlying modechart. Rather, they are prefixes of such computations. However, the following theorem holds. The theorem states that every constrained trace from a simulation verification graph is the prefix of some computation from some constrained trace in the corresponding computation graph. The proof follows from the above definitions and the simulation-verification graph construction algorithm of Algorithm 1. For more details, the reader is referred to [38]. A completed constrained trace extends the notion of constrained trace to infinte executions.

**Theorem.** *Let $M$ be a modechart with computation graph $G$. Let (path, tc) be a simulation path and termination condition. Let $S$ be the simulation-verification graph for $M$ with simulation path path and termination condition tc constructed by the algorithm of Algorithm 1. Then, for every constrained trace in $S$, there is a completed constrained trace in $G$ with the same events which has computations which agree with those of the constrained trace of $S$.*

### 3.5 Examining the Generated Graph

Having shown how to generate a simulation-verification graph, several issues of how to use such a graph remain to be discussed. A *query* is any kind of property or question which can be evaluated on a simulation-computation graph. This section describes some of the major types of queries which can be evaluated and discusses the primary differences between the evaluation of queries on a computation graph and on a simulation-verification graph.

As with standard computation graphs, there are two basic types of queries that can be applied to simulation-verification graphs. The first type, *graph structure* queries, are queries that are resolved by considering the individual points in the graph. These are answered in the same manner for both types of graphs. For example, the user may want to know if the modes $Mode2$, $Mode4$, and $Mode6$ are simultaneously active in the simulation-verification graph. To determine whether a particular mode is reachable in a simulation-verification graph, it is only necessary to examine the points in the graph and determine if the mode of interest appears in the label of at least one point. A quick inspection shows that these modes are not simultaneously active in the simulation-verification graph and, therefore, not simultaneously active in the computation prefixes corresponding to the subset of computations restricted by the initial prefix and the frontier condition.

Other graph structure queries, besides reachability conditions, include queries about the size of the simulation-verification graph and queries about adjacent points.

The second type of query that can be applied to either type of graph are *path queries*. These queries involve the examination of the timing among events along path segments. Generally, these are universal queries and are evaluated on all path segments. The simulation prefix is represented as points in the simulation-verification graph and is evaluated in the same way as the rest of the graph. These types of queries are more difficult to evaluate in the context of simulation-verification.

The difficulty, however, is not in the algorithms to evaluate the queries, which are essentially similar to those

for standard computation graphs, but in the intended semantics of the queries themselves. The basic problem is how to incorporate the effect of the frontier into the result of the query.

Evaluating a path query on a particular path typically involves identifying points in the graph with events in the query. In the case of a path query in a svgraph, there are two cases that can arise. If all of the events in the query can be identified with points on the path, then the query can be evaluated by examining the timing constraints in force along the path. The frontier introduces a second case. One or more of the events in the query may not correspond to points on the path because the events occur at points beyond the frontier. In this case, there are several choices for evaluating the query, based on what interpretation is placed on the frontier.

There are three possible ways to evaluate such an instance, and each may be reasonable in some context. Partial instances could be considered as satisfying the query. This might be appropriate if, for example, the query seeks to establish a minimum separation between two events and the first event appears on the path segment, while the second, if it occurs, is beyond the frontier and the timing constraints on the path segment guarantee that the frontier is farther from the first point than the minimum separation. In such a case, the query instance can reasonably be regarded as true. For example, consider the simulation-verification graph in Fig. 2. Let the query be the minimum separation between $\rightarrow Mode4$ and $\rightarrow Mode6 > 1$. Consider the path that includes points (0, 1, 4, 7, 10, 13, 16, 19, 22, 26, 31, 37, 42, *and* 54). $\rightarrow Mode4$ occurs at point 26 at time = 8, while $\rightarrow Mode6$ occurs at point 42 at time = 10. Since $\rightarrow Mode6$ occurs at the frontier, it is impossible to know from direct examination of the simulation-verification graph what how much time can elapse before the next occurrence of $\rightarrow Mode4$. But each occurrence of $Mode4$ must be separated from the previous one by five time units, at least five time units must pass from point 26 and only two time units have elapsed between points 26 and 42. Therefore, it can be concluded that at least three more time units must occur from point 42 and the next occurrence of $\rightarrow Mode4$.

Conversely, if the query in the previous example were changed to maximum separation, such a partial instance could reasonably be considered to falsify the query, since if the second event does occur, it occurs beyond the frontier and, hence, after the deadline. Alternatively, if the timing constraints along the path indicate that the frontier is closer to the first point than the minimum or maximum separation, in neither case is enough information available to evaluate the query. In this case, it might be reasonable to ignore the partial instance on the assumption that the choice of frontier was intended to eliminate from consideration such partial instances.

Similarly, while the frontier provides an absolute limit to each path segment and each instance, the information necessary to generate the graph beyond the frontier is available. For a given instance, there may be circumstances when it might be appropriate to exploit this information when evaluating queries. For example, if the frontier includes a point where a transition resulting in an event that would complete a partial instance is first enabled, then the timing of the event with respect to the frontier can be determined by inspection. In such a case, it might be reasonable to evaluate the partial instance in light of this additional timing information.

As illustrated by the examples in the preceding paragraphs, the appropriate semantics of path segment queries in simulation-verification cannot be determined a priori for all queries and all graphs The appropriate semantics are heavily dependent on the individual graphs and queries. Rather than making a single choice of frontier semantics that will apply to all queries, the approach we have taken is to require the user to indicate how the frontier is to be treated in each query. With all of the frontier semantics options available, the user will be able to choose the appropriate frontier semantics for each query and not be bound to a single frontier semantics.

## 4 The SVT Tool

A prototype simulation-verification tool, *XSVT*, has been implemented as part of the MT toolset. The MT toolset consists of four tools: a user interface tool for graphically creating, displaying, manipulating, and modifying Modechart specifications, a simulation tool for generating and examining computation prefixes, a verification tool for verifying properties of specifications, and the new simulation-verification tool, XSVT. The simulation tool takes a modechart as input, together with a set of simulation options, and generates an execution trace of the specification consistent with the simulation options, which provide guidance in making the nondeterministic choices necessary to generate a particular computation prefix. The verification tool uses model-checking to verify properties of a specification.

XSVT integrates the existing simulation and verification tools into a single tool for performing simulation-verification-based analysis of Modechart specifications. The XSVT tool currently runs on a Sun Sparcstation running X and UNIX and is implemented in C using XView. The tools' proper consists of a single window which displays the input specification with three menus. The *File* menu includes commands to exit the tool and load and clear specifications. The *Simulator* menu includes commands to invoke the simulator and to cause the simulator to generate a computation prefix that can be used to generate a simulation-verification graph. The *Verifier* menu includes commands to invoke the verifier, load a computation prefix and termination condition, and generate the resulting simulation-verification graph. The simulator and verifier invoked by XSVT retain all of their original functionality. The only significant modification necessary to the existing simulation and verification tools was to modify the simulator to generate output which can be used to generate computation prefixes. Otherwise, the tools invoked by the XSVT tool are unchanged. Similarly, most of the modifications necessary to the basic computation graph generation algorithm necessary for the creation of simulation-computation graphs have been discussed in Section 3. The only frontier semantics

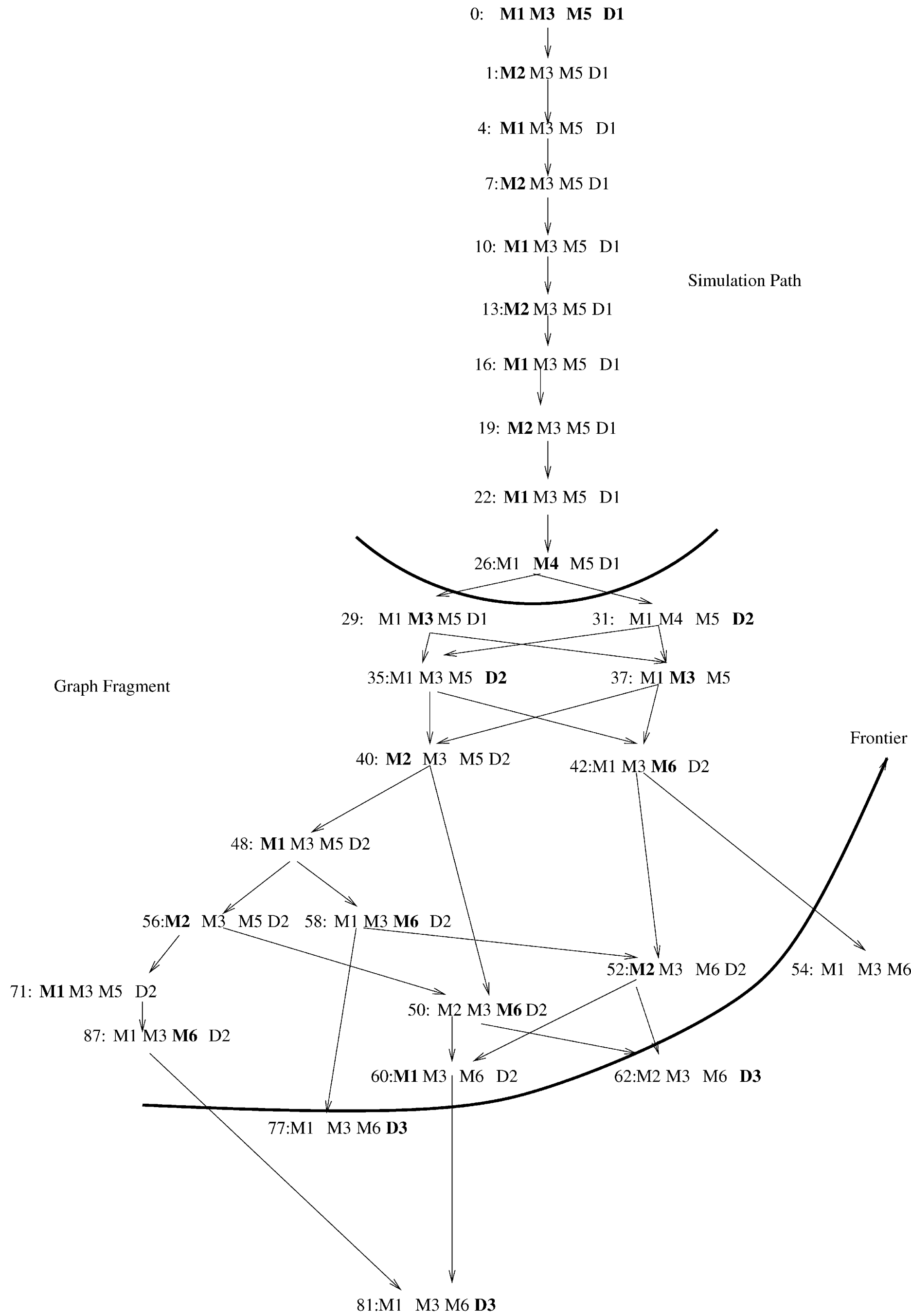


Fig. 5. Simulation-verification graph for simple Modechart.

currently implemented is to consider events beyond the frontier as not occurring.

Fig. 5 is the completed simulation-verification graph for the modechart of Fig. 2, using the path of Fig. 4 and transition expression termination condition $(10, 10)| \rightarrow MODE6$, as generated by the XSVT tool. Note that the simulation-graph has only 28 points compared to the 208 in the full computation graph.

## 5 Applications of Simulation-Verification

The previous sections have introduced the simulation-verification analysis technique for Modechart specifications and described a prototype tool that performs the analysis. In Section 1, simulation-verification was presented as an analysis technique combining simulation and verification, reducing the cost of verification by reducing the level of guarantee that the specification behaves as intended. Because the level of guarantee is reduced, the disadvantage of this approach is that the usual verification analysis techniques cannot be applied. However, the advantage of the simulation-verification approach is that this technique provides more information than standard simulation. As a consequence, the technique requires a new style of analysis to be effective.

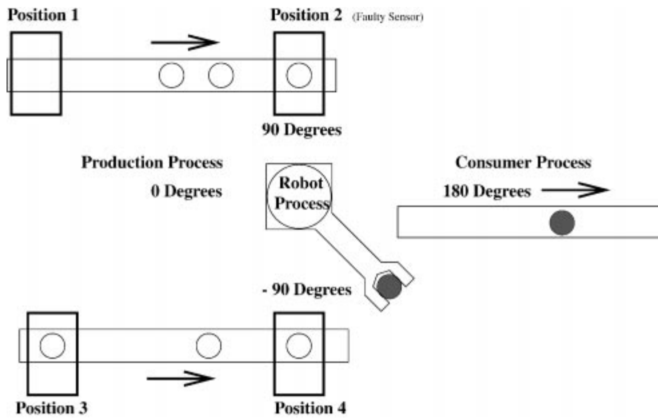


Fig. 6. Diagram of a robot controller for a manufacturing assembly line.

This section will introduce three broad applications of this technique. The first analysis style is closely related to standard specification simulation. The second style is analytical in nature. The final use proposed is also analytical in nature, but addresses potential problems in the implementation of conventional verification. In Section 5.1, a robot controller specification is introduced which will be used to illustrate these techniques.

### 5.1 A Robot Controller Example

Consider a robot in a manufacturing assembly line illustrated in Fig. 6. Two producer processes control two conveyer belts carrying items to be processed by a robot. These producer processes are responsible for moving items from Position 1 to Position 2 and from Position 3 to Position 4. When an item is in Position 2 or Position 4, the robot picks up the item, rotates away from the belt, and processes the item. (If the robot fails to pick the item up in a timely fashion and the conveyer belt is still moving, the item may move off the end of the conveyer belt and fall onto the floor.) Next, the robot rotates again and attempts to drop the item. Finally, the robot rotates in the opposite direction to return to the initial position.

These controller processes are physically distributed and communicate with each other (as well as with the environment) through sensors. These sensors are also specified in Modechart. Finally, Modechart is also used to model certain aspects of the environment in which the system operates. This permits the expression of various natural constraints on the behavior of the system.

Fig. 7 describes the robot controller process. The controller issues signals to the robot environment. These signals are modeled by the mode entry and mode transition events of the controller process. For example, the event $\rightarrow RC.ChangeDirection$ sends a signal to the robot arm to change the direction it is rotating. The controller sends two sets of signals to the robot environment. The first is the "change direction" signal. The change direction signal readies the robot to grab from the appropriate producer belt. The second set is the "rotate, extend, grab, process, rotate, wait, drop, retract, rotate" sequence which controls the robot processing. Each signal in this sequence is sent after the environment indicates that it has responded to the previous signal.

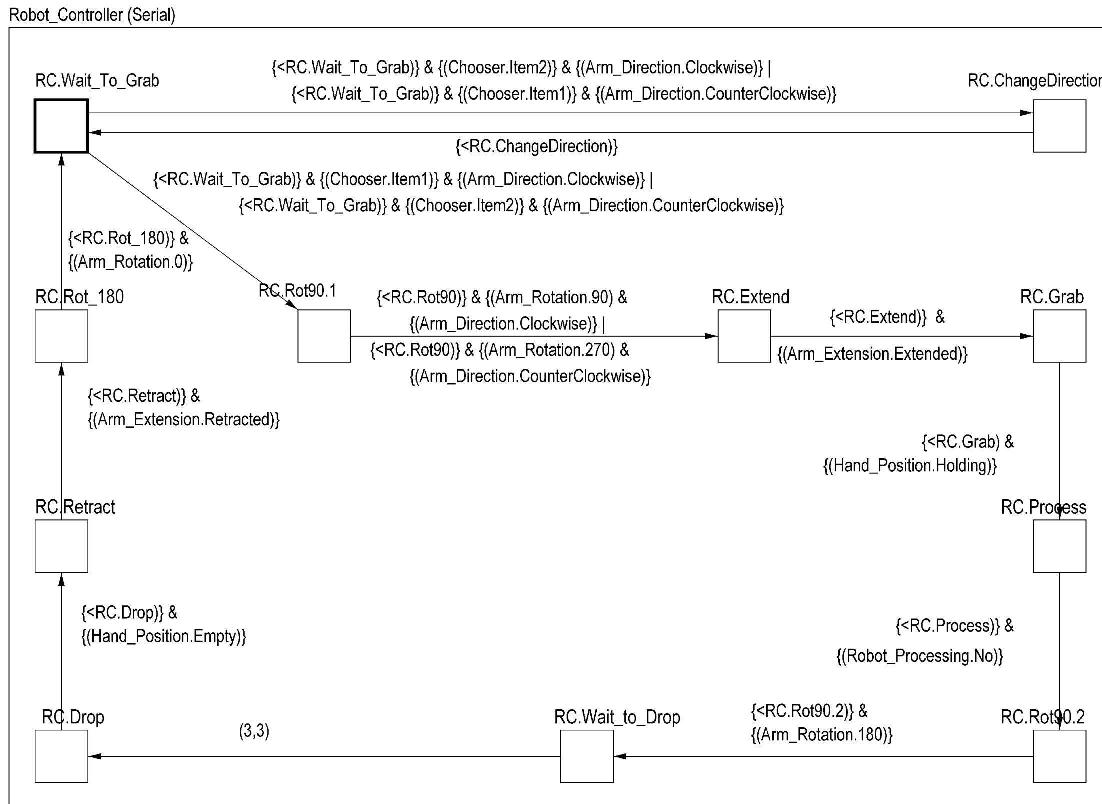


Fig. 7. Robot controller for a manufacturing assembly line.

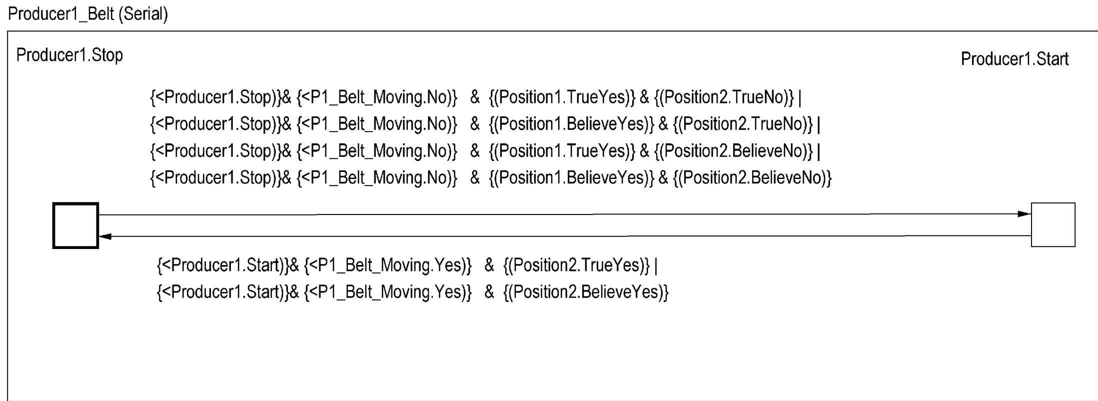


Fig. 8. Producer process for a manufacturing assembly line.

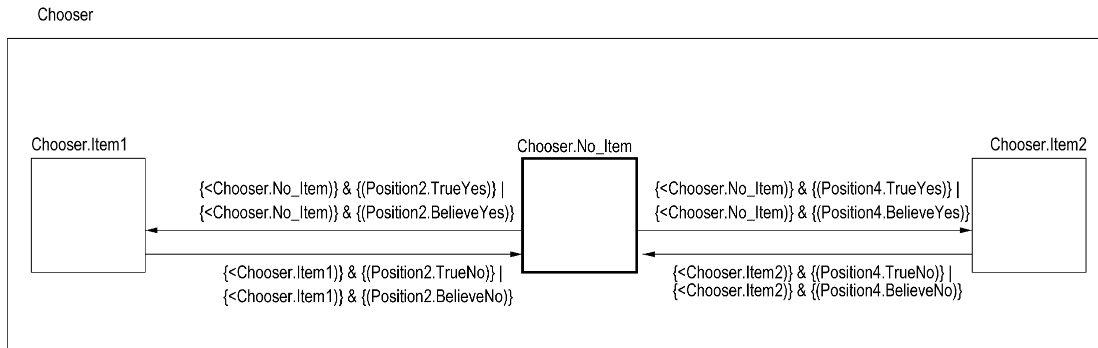


Fig. 9. Chooser process for a manufacturing assembly line.

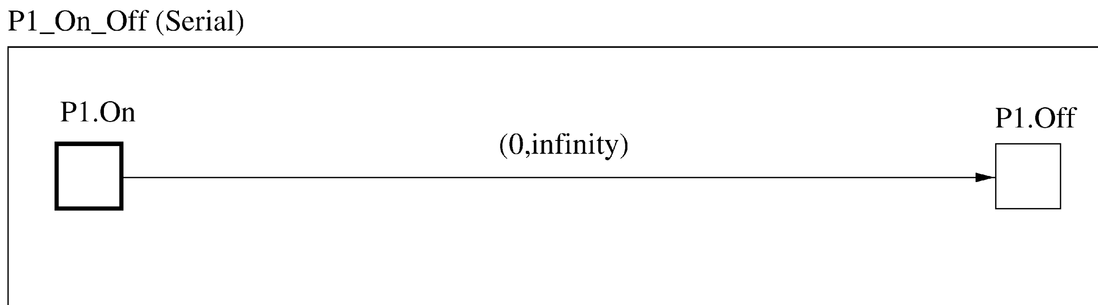


Fig. 10. On/Off switch for a manufacturing assembly line.

Fig. 8 depicts the process which controls production on a conveyer belt. The second producer is similar. Fig. 9 describes a small processor which chooses which item to be processed should two items be available simultaneously. Fig. 10 depicts a switch which controls whether a conveyer belt is turned on or not. Each belt can be turned off at any point.

The producer process, indicated by the *serial mode Producer_Belt*, has two major constraints on its behavior. If the belt has been stopped for one time unit and there is an item in Position 1 and no item in Position 2, the producer process starts the producer conveyer belt. If the belt has been moving and there is an item in Position 2, the producer process stops the conveyer belt. The producer belt observes the environment through sensors which indicate whether there is an item in Position 1 or Position 2. The second producer process operates in a similar manner, starting and stopping the conveyer belt to move items from Position 3 to Position 4.

Fig. 11 displays a specification for some typical sensors. Note that there is a delay between the time in which an item moves into a position and the time which that information is available to the controller processes.

The constraints on the robot controller's behavior may be stated informally as follows: If an item is available in Position 2 or Position 4 and the robot arm is set to move in the correct direction, the robot rotates directly to the correct position. If the robot is not set to move in the correct direction, it changes its direction and then rotates into position. When the robot is in position to grab an item, it extends its arm, opens its hand, and grabs the item. It then processes the item and then rotates 90 degrees. It then waits to be able to drop the item. Following dropping the item, the robot retracts its arm, then rotates 180 degrees in the opposite direction. The robot communicates with the producer belt through the sensors indicating whether there is an item in Positions 2 and 4. The chooser controller indicates to the robot controller which item is available.

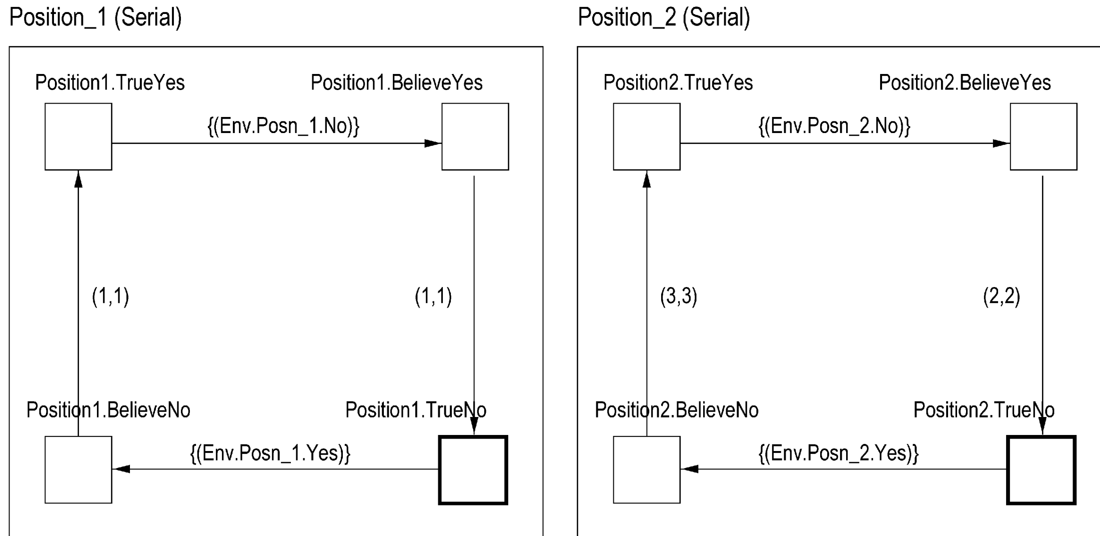


Fig. 11. Modechart specification of a typical sensor for a robot controller for a manufacturing assembly line.

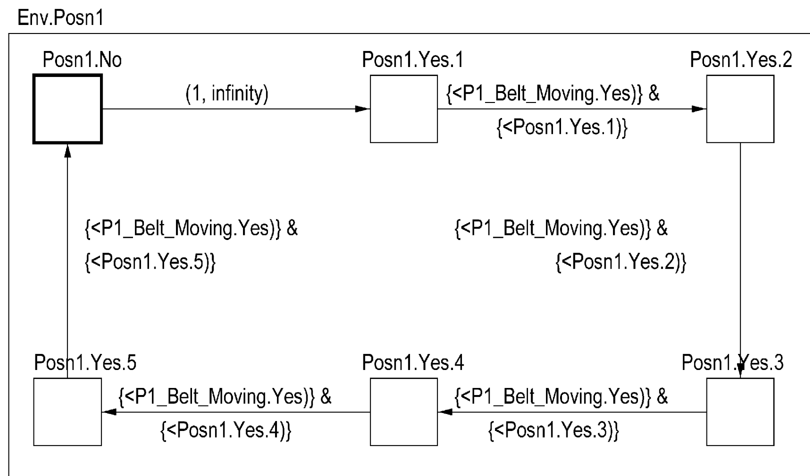


Fig. 12. Modechart specification for Environment–Position 1.

The sensors detect the state of the environment and relay this information to the robot controller. There is a communication delay in relaying a state change in the environment back to the processes relying on the sensors. As a consequence, it is possible that a sensor will indicate a stale value to the controllers. In Fig. 11, mode *Position1.BelieveNo* indicates a situation where there is an item in Position 1, but this fact has not yet been detected by the sensor. After some delay, the sensor tests the environment again and moves to mode *Position1.TrueYes* which indicates that the sensor now records the correct state of the environment. The mode *Position_1.Yes* indicates the state where the sensor records that there is an item in Position 1, while the mode *Position_1.No* indicates the state where the the sensor records that there is no item in Position 1. These modes are used by the processes to determine mode transitions.

Note that the sensor for Position 2 is slower than the one for Position 1. This models a communications path which is faulty in some way, leading to a delay in transmission of information to the robot controller. As a consequence, the controller does not learn about changes in the environment at Position 2 until three time units after they occur. The simulation-verification tool is used to explore the effect of this faulty sensor on the behavior of the robot.

Figs. 12, 13, 14, 15, 16, and 17 depict the modechart specifications for the major components of the environment. Fig. 12 describes the environment with regard to whether there is an item in Position 1. Similarly, Fig. 13 and Fig. 15 describe whether there is an item in Position 2 or on the floor. Fig. 16 depicts whether the robot is processing an item. Finally, Fig. 17 indicates whether the producer belt is moving.

As noted above, because sensors sample the environment periodically, there is a delay between the occurrence of an event in the environment and the time which that event is visible to the controller process. In this specification, the sensor at Position 2 is slightly slower than the other sensors, requiring three time units to detect a state change in the environment. Simulation-verification is used to examine the circumstances under which this delay might lead to items falling off the end of the producer belt on to the floor before the controllers have time to take the item for processing or stop the conveyer belt.

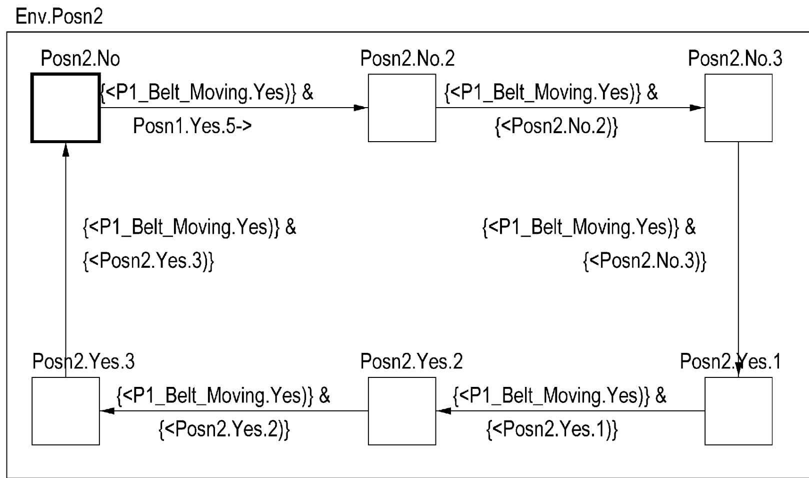


Fig. 13. Modechart specification for Environment–Position 2.

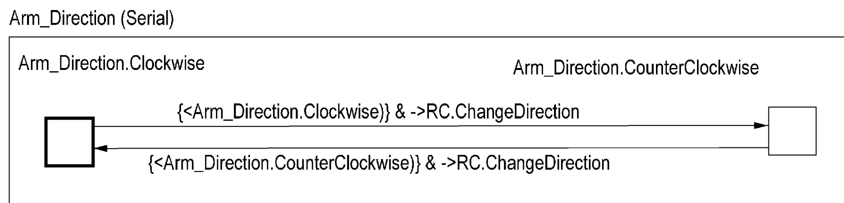


Fig. 14. Modechart specification for Environment–Arm Direction.

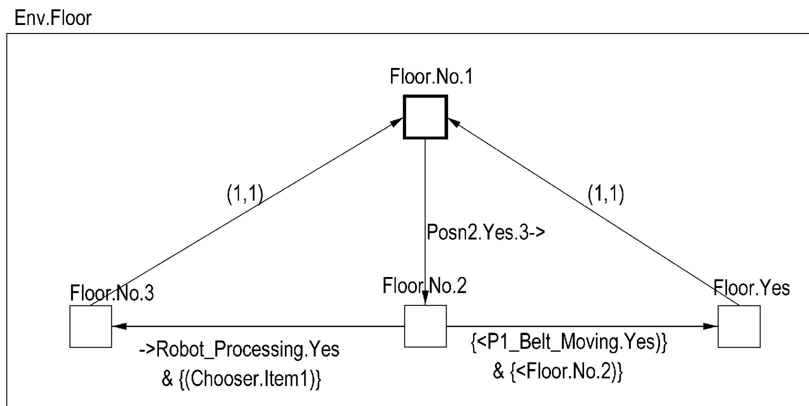


Fig. 15. Modechart specification for Environment–Floor.

### 5.2 Full Verification of the Robot Specification

In order to better evaluate the simulation-verification technique, the size of the full computation graph should be considered. For the above robot specification, the full computation graph generated 369,795 points, resulting in a computation graph containing 58,639 good points. (Equivalent points are collapsed and unreachable points are pruned). This computation required 833:03 minutes of cpu time on a Sun Ultra Sparc having 320 Megabytes of physical memory. Both the number of explored points and the number of good points are important baselines for comparision. The number of explored points is relevant because it indicates the complexity of generating a computation or simulation-graph. The number of good points is relevant because the complexity of queries on the computation or simulation-verification graph will be determined, in part, by the number of points.

### 5.3 Simulation-Verification as a Form of Simulation

The first approach to using simulation-verification is to use it as a sophisticated form of simulation. One of the uses of simulation is to exhibit typical behaviors of a specification to aid the system designer in understanding the specification, as an aid to determining if the specification behaves as desired or intended. Using simulation-verification allows the specification designer to consider all of the behaviors of a specification consistent with a particular computation prefix, allowing the designer to work with groups of computations at once. This can be particularly important if the specification is the result of a scenario-oriented specification process, such as is presented in [10]. In such a design paradigm, where the specification is generated from one or more scenarios provided by the user, the behavior of the specification outside of the given scenarios may be unclear to the user. Simulation-verification can be used in

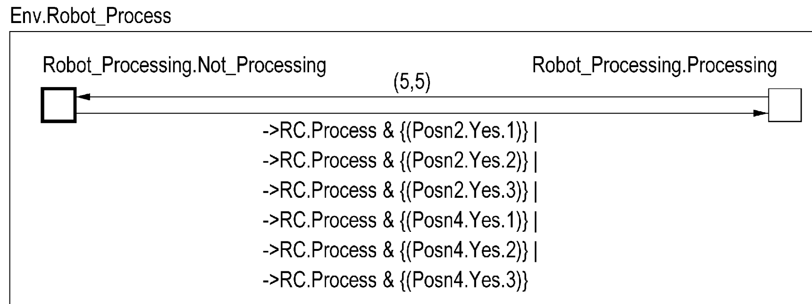


Fig. 16. Modechart specification for Environment–Robot Processing.

this situation to generate all of the computations of the specification that continue a scenario for comparison with the expectations of the user. Simulation-verification can also be used to test whether or not the specification actually reproduces the scenario or scenarios since a scenario is just a computation prefix. If the scenario is actually a computation of the specification, when used as a computation prefix by simulation-verification, a valid simulation-verification graph will be constructed.

To illustrate this approach, suppose the modechart specification of the robot described above were the result of a scenario-oriented design process involving only scenarios in which only one item was produced on the producer belt having the bad sensor. A simulation-verification graph having an initial trace with the generation of this item might help the user gain insight into the behavior of the specification under these circumstances. For example, simulation of the robot specification might indicate whether that item fell on the floor.

**Experiment 1.** Generate one item on the belt with the bad sensor. The simulation prefix in this case is simply:

- P2.On →*P*2.Off:0
- Posn1.No → Posn1.Yes.1:1
- P1.On → P1.Off:0

The frontier condition (5,000, 5,000) is used to guarantee that generation of the simulation-verification graph terminates. (This condition is technically unnecessary since turning off each conveyer belt guarantees that eventually the system comes to a halt.)

Not only does the simulation-verification tool permit the user to focus on a subset of computations, but the resulting simulation-verification graph is often smaller than the full computation graph. In this case, the savings was substantial. The simulation-verification graph generated 116 points, about 0.2 percent of the original. A total of 411 points were examined and the graph was generated in 0:07 seconds.

Examination of this simulation-verification graph indicates that this item does fall on the floor.

**Experiment 2.** Generate one item on the belt with the good sensor.

A complementary analysis considers whether a single item generated on the good belt falls on the floor. This simulation-verification graph contained 83 points (219 points were examined). This graph was generated in five seconds.

**Experiment 3.** Generate only (any number) items on the belt with the bad sensor. This was accomplished by turning off the good belt.

The simulation prefix was:

- P2.On →P2.Off:0

The frontier condition (→P1.Off,1,=) causes the generation of the simulation-verification graph to trim any computations having the first (bad) belt turned off. As a consequence, only computations having the bad belt on are considered and, only up to any point where the belt is turned off, should that occur.

The size of this computation graph was 5,271 points. 5,662 points were considered and it was generated in 0:26.

### 5.4 Simulation-Verification as an Analytical Technique

The second approach to using simulation-verification in the analysis of specifications is more analytical than the preceding approach. The advantage of simulation-verification over conventional verification is that simulation-verification performs analysis on only a portion of the computations of a specification. This advantage is also its weakness since it can offer no guarantees since a property could hold for a simulation-verification graph of a

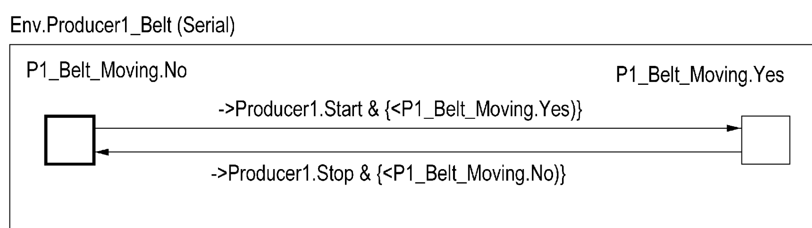


Fig. 17. Modechart specification for Environment–Producer Belt.

TABLE 1
Summary of Experiments

| Experiment | Points Generated | % of Full | Points Included | % of Full | Execution Time | % of Full |
|---|---|---|---|---|---|---|
| Full | 369,795 | n/a | 58,639 | n/a | 833:03 | n/a |
| 1 | 411 | 1.11% | 116 | 1.98% | 0:07 | 0.01% |
| 2 | 219 | 0.06% | 83 | 1.42% | 0:05 | 0.05% |
| 3 | 5,662 | 1.53% | 5,271 | 8.99% | 0:26 | 0.05% |
| 4 | 1,294 | 0.35% | 1,226 | 2.09% | 0:05 | 0.01% |
| 5 | 3,492 | 0.94% | 803 | 1.37% | 537 | 0.67% |

specification and, therefore, not hold for an implementation of the specification. There is one condition under which simulation-verification is equivalent to verification. If the property itself specifies that it need only hold for some of the computations of the specification, it is not necessary to check the property against the entire computation graph. One form such a property could take, expressed in temporal logic, might be *E path U (A(formula U frontier))*, where *path* is a temporal logic specification of a computation prefix, *formula* is a temporal logic formula intended to hold from the end of the simulation path to the frontier, and *frontier* is a temporal logic specification of a termination condition. This type of property is quite common and can be viewed intuitively as stating that under a particular scenario, some property holds at least for an interval after the scenario. Note that a *(path, formula, frontier)* simulation-verification triple might be the most efficient or practical way of specifying such a property.

**Experiment 4.** Consider the following property: "No items fall on the floor if only the belt with the good sensor is turned on." This is complementary to Experiment 3, where only the behavior of the belt with the bad sensor was explored. This could be expressed as an existential property with *path*: The simulation prefix was:

- P1.On →P1.Off:0

The frontier condition (→P2.Off,1,=) is identical to the one used in Experiment 3 except that it looks for the good belt to be turned off.

The property of interest is: *E → Floor2.Yes.1*.

Since the property can be expressed as an existential property having the structure described above, it can be verified on the smaller simulation-verification graph. The resulting graph is much smaller (1,226 points) than the full computation graph. (1,294 points were generated and the experiment required 0:05 cpu time.) Examination of the simulation-verification graph indicates that this property does not hold for the specification, i.e., no item falls on the floor.

This second approach can also be used to provide a lesser level of guarantee. It may be that the specification designer, due to limited resources, is unable to perform full verification of the specification, but using domain specific knowledge, or some other technique, the designer can identify a subset of the computations of a specification corresponding to one or more simulation-verification graphs of the specification that are most likely to contain computations that violate a property of interest. In such a situation, simulation-verification can be used to validate the property in the areas of highest concern, and the quality of the resulting correctness guarantee depends on the quality of the reasoning that identified the particular simulation-verification graphs as most likely to contain a computation violating the property.

Again, turning to the robot example, consider a specification similar in most respects to the robot description above, but having a correct sensor at Position 2. That is, the sensor communicates changes in the environment with a delay of one time unit. Does this robot specification permit items to fall on the floor? Rather than examine the full computation graph, the user might generate simulation-verification graphs for a sequences where items are generated closely together in time. If the items do not fall on the floor in these simulation-verification graphs, the user might infer that they do not for the full computation graph.

**Experiment 5.** For a specification having correct sensors, generate two items simultaneously and two items as quickly as possible afterwards. Do any of these items fall on the floor? In this experiment, the simulation prefix is:

- Posn1.No→Posn1.Yes.1:1
- Posn3.No→Posn3.Yes.1:0
- Position1.TrueNo→Position1.BelieveNo:0
- Position3.TrueNo→Position3.BelieveNo:0
- Position1.BelieveNo→Position1.TrueYes:1
- Position3.BelieveNo→Position3.TrueYes:0
- Producer1.Stop→Producer1.Start:0
- Producer2.Stop→Producer2.Start:0
- P1_Belt_Moving.No→P1_Belt_Moving.Yes:0
- P2_Belt_Moving.No→P2_Belt_Moving.Yes:0
- Posn1.Yes.1→Posn1.Yes.2:0
- Posn3.Yes.1→Posn3.Yes.2:0
- Posn1.Yes.2→Posn1.Yes.3:1
- Posn3.Yes.2→Posn3.Yes.3:0
- Posn1.Yes.3→Posn1.Yes.4:1
- Posn3.Yes.3→Posn3.Yes.4:0
- Posn1.Yes.4→Posn1.Yes.5:1
- Posn3.Yes.4→Posn3.Yes.5:0
- Posn1.Yes.5→Posn1.No:1
- Posn1.No→Posn1.Yes.1:0
- Posn2.No.1→Posn2.No.2:0
- Posn3.Yes.5→Posn3.No:0
- Posn3.No→Posn3.Yes.1:0
- Posn4.No.1→Posn4.No.2:0
- Position1.TrueYes→Position1.BelieveYes:0
- Position3.TrueYes→Position3.BelieveYes:0
- P1.On→P1.Off:0
- P2.On→P2.Off:0

The termination condition is (5,000, 5,000). Since both belts are turned off, we would expect that the computation would terminate in much less than 5,000 time units.

The resulting simulation-verification graph shows that none of these four items fall on the floor. The graph required 5:37 minutes of cpu time to generate. Three thousand four hundred and ninetytwo points were explored and the final graph contained 803 points.

This approach is similar in spirit to automated test case generation, as in [30], but is broader in scope in that it checks all of the computations that agree with a particular prefix (test case) simultaneously, and the use of the frontier provides a natural way to limit the growth of the resulting graph.

The third basic approach to using simulation-verification is also analytic, but is directed more toward using simulation-verification as a step in full verification, in the hope of saving space. Conventional verification constructs the entire computation graph for a specification in a breadth first fashion and evaluates a property of interest only when the entire graph has been generated. Since computation graphs can be large, this results in a heavy demand on storage. In this context, simulation-verification can be viewed as a depth-first approach to generating a computation graph. Accordingly, it may be possible to use simulation-verification to, in a sense, generate the computation graph of a specification a slice at a time, and evaluate the property of interest on each slice as it is generated, discarding the slice after the property has been evaluated. If such an approach is proven practical, verification could be performed without the complete computation graph ever existing, resulting in a lower demand for memory. This approach is in the very preliminary stages of consideration; much additional investigation is necessary.

## 6 Related Work

At this point, it is possible to make a few more observations on the position of the methods and tools 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 languages have 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 [1], [2], [14], [26], [27], [29], [31]. Some representative examples of process-algebraic languages for real-time systems are those of [16], [28], [35]. Also, several researchers [25], [32], 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 infinite 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 [2], [19], [22], [32], [37], including a report on an earlier version of the verification method employed in the Modechart Toolset [25]. More recently, model checking has been extended to hybrid systems that are modeled as linear hybrid automata [3].

Even using BDD-based techniques, all such techniques must eventually confront the state-explosion problem. Simulation-verification provides an additional mechanism for mitigating this problem. In [21], a parallel approach to the idea of simulation-verification is taken which focuses on simulation as the heart of the technique rather than verification. Rather than using simulation to restrict verification, verification is used to expand simulation, so that rather than building a simulation-verification graph, a *tableau-simulation* is performed, in which the idea of a computation graph is used to represent a family of simulations of a modechart as a graph. The advantage of the tableau-simulation approach is that it is more closely related to simulation and allows easier migration to simulation. The disadvantage is that it is more expensive to expand a simulation than reduce a computation graph.

Proof methods based on *process algebras* rely on descriptions of systems and requirements as algebraic expressions. The proof method relies heavily on algebraic identities to prove equivalence between algebraic expressions. Development of mechanical proof methods for for process algebras is still preliminary. In order to evaluate the effectiveness of these reasoning techniques, prototype tools have been developed, such as those in [7], [8].

While we have seen considerable progress in the development of formal specification and analysis methods for real-time computing, wide-spread application of these methods is unlikely without appropriate software tools. 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" [11], [12].[1] 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 SMV symbolic model checker supports the automated verification of real-time systems modeled as labeled state-transition graphs, where each path corresponds to an execution trace of the system. A state-transition graph is represented internally using binary decision diagrams (BDDs), which generally provide a more compact representation of the system behavior. The SMV symbolic model checker has been shown to handle very large state-spaces

1. A well-written summary of this report appeared in [17].

efficiently. In a recent work [6], algorithms for computing quantitative information about finite-state real-time system have been incorporated into the SMV model checker. HyTech is another symbolic model checker intended for automatic verification of hybrid systems. In this approach, systems are modeled as linear hybrid automata [3], machines with finite control and real-valued variables modeling continuous environment parameters. System properties specified in a real-time temporal logic are verified by symbolic computation. A recent report [20] describes a more portable implementation of HyTech which is shown to be significantly more efficient than the initial prototype implementation.

As mentioned earlier, proof methods based on *process algebras* rely on descriptions of systems and requirements in a process algebraic language. The Concurrency Workbench [8] is an automatic verification tool for finite-state processes specified in Temporal CCS. A recent case study [13] illustrated how the Modechart specification of a distributed real-time system for active structural control can be translated to Temporal CCS and then verified using the Concurrency Workbench. VERSA [7] is another prototype toolkit which has been developed to support reasoning about real-time systems modeled as ACSR processes.[2] It is intended to provide support for checking syntax and carrying out analysis automatically. In particular, the VERSA toolkit has three major functions: rewriting, equivalence testing, and interactive execution. To facilitate the use of ACSR by novice programmers, the graphical specification language GCSR [4], with semantics defined in ACSR, has been developed and has been incorporated into the VERSA toolkit.

Interactive proof-developing approaches, such as those supported by HOL [18], Isabelle [34], PVS [33], and the Boyer-Moore theorem-prover [5] are noteworthy since, like the approach described here, they involve user participation. Often, when developing a proof, the users of these tools exploit knowledge of the domain and of the specification to focus their efforts to a case that can be generalized to prove a property for the entire specification or to narrow attention to behavior where a counter example is likely. This is similar to the way a user of the simulation-verification approach might approach the problem of deciding what portion of the state-space to examine.

## 7 Conclusion

This paper has introduced a new analysis technique for specifications of real-time systems, simulation-verification, that combines the advantages of simulation and verification. Simulation-verification synthesizes simulation and verification by using simulation to generate a computation prefix, which is used to restrict the scope of verification to those computations which agree with the prefix. While simulation is relatively inexpensive in terms of execution time, it only validates system behavior for a single computation prefix. In contrast, full verification can be prohibitively expensive since the state-space explosion problem is exponential. This approach provides an intuitive way to verify a meaningful subset of the computation space with the potential to achieve dramatic savings in computation time. However, the drawback is that since only a portion of the state space is examined, a property may not hold for an implementation of the specification.

A tool, XSVT, for generating and querying simulation-verification graphs has also been introduced, as well as three paradigms for the use of the technique. Several examples demonstrating reductions of several orders of magnitude in computation time have been presented.

The current version of XSVT only allows information to be passed from the simulator to the verifier. A session using the XSVT, thus, consists of using simulation to generate a computation prefix, building and examining the corresponding simulation-verification graph and, then using the simulator to produce a new prefix, or perhaps building a simulation-verification graph for a different termination condition. It would also be desirable to use the simulation-verification graph as a means to select the next simulation. Given a particular simulation-verification graph, the user could then select a particular path in the computation graph fragment of the simulation-verification graph to extend the computation prefix and use that extended prefix as the starting point for future simulation. This type of directed simulation could prove particularly useful in the scenario-oriented design paradigm for using simulation-verification. At present, this technique is allowed by the toolset, through the simulator's interactive transition selection facility, but must be done by hand. Future versions of XSVT will mechanize this process. Other current and future work on improving the simulation-verification technique includes implementing alternate query semantics, developing a suitable interface to allow the user to select from among the available semantics at query time, and evaluation of the merits and utility of the various different query semantics, as discussed in Section 3.

## 8 Appendix

The constraints on a constrained trace consisting of the sequence of points $\sigma_i$, $i \geq 0$, are determined by the following rules, for all $i$.

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

2. ACSR is an extension of another real-time process algebra called CCSR [15], which was the first process algebra to support the notions of both resources and priorities.

The event set referred to by constraints of types (2) and (3) are used to ensure that triggering conditions are taken at the same time as the events which cause them and are included here for completeness. The reference point of a timing condition is the point where the source mode is entered. Constraints (6) and (7) ensure that the deadlines are observed when there are multiple transitions exiting a mode.

## Acknowledgments

This research was supported in part by a research grant from the Office of Naval Research under ONR contract number N0014-94-1-0582. At the time this research was conducted D.A. Stuart was with the University of Texas at Austin and M. Brockmeyer was with the Real-Time Computing Laboratory in the Electrical Engineering and Computer Science Department of the University of Michigan, Ann Arbor, Michigan.

## References

[1] R. Alur, C. Courcoubetis, and D. Dill, "Model-Checking for Real-Time Systems," *Proc. Fifth Ann. IEEE Symp. Logic in Computer Science,* June 1990.

[2] R. Alur and D. Dill, "The Theory of Timed Automata," *Proc. REX Workshop Real-Time: Theory in Practice*, pp. 45-73, June 1991.

[3] R. Alur, T.A. Henzinger, and P.-H. Ho, "Automatic Symbolic Verification of Embedded Systems," *Proc. IEEE Real-Time Systems Symp.,* pp. 2-11, Dec. 1993.

[4] H. Ben-Abdallah, I. Lee, and J.-Y. Choi, "A Graphical Language with Formal Semantics for the Specification and Analysis of Real-Time Systems," *Proc. Real-Time Systems Symp.,* pp. 276-286, Dec. 1995.

[5] R. Boyer and J. Moore, *A Computational Logic Handbook.* New York: Academic Press, 1988.

[6] S. Campos, E. Clarke, W. Marrero, and M. Minea, "Computing Quantitative Characterstics of Finite-State Real-Time Systems," *Proc. IEEE Real-Time Systems Symp.,* pp. 266-270, Dec. 1994.

[7] 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, Univ. of Pennsylvania, Dept. of Computer and Information Science, Sept. 1993.

[8] R. Cleaveland, J. Parrow, and B Steffen, "The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems," *Trans. Programming Languages and Systems,* vol. 15, pp. 36-72, 1993.

[9] P.C. Clements, C.L. Heitmeyer, B.G. Labaw, and A.T. Rose, "MT: A Toolset for Specifying and Analyzing Real-Time Systems," *Proc. IEEE Real-Time Systems Symp.,* Dec. 1993.

[10] P. Clements, "Requirements Definition Languages for Real-Time Embedded Systems," PhD thesis, The Univ. of Texas at Austin, 1993.

[11] 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, Technical Report 5546-93-9581, US Naval Research Laboratory, Washington, DC; Technical Report Info-0474-1, Atomic Energy Control Board of Canada, Ontario, Nat'l Technical Information Service, Springfield, VA, 1993.

[12] 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, Technical Report 5546-93-9582, US Naval Research Laboratory, Washington, DC; Technical Report Info-0474-2, Atomic Energy Control Board of Canada, Ontario, Nat'l Technical Information Service, Springfield, VA, 1993.

[13] W.M. Elseaidy, R. Cleaveland, and J.W. Baugh, "Verifying an Intelligent Structural Control System: A Case Study," *Proc. IEEE Real-Time Systems Symp.,* pp. 271-275, Dec. 1994.

[14] E. Emerson, A. Mok, A. Sistla, and J. Srinivasan, "Quantitative Temporal Reasoning," *Workshop Automatic Verification of Finite-state Systems,* 1989.

[15] R. Gerber and I. Lee, "Ccsr: A Calculus for Communicating Shared Resources," *Proc. Int'l Conf. Concurrency Theory, CONCUR '90,* Aug. 1990.

[16] R. Gerber and I. Lee, "A Resource-Based Prioritized Bisimulation for Real-Time Systems," Technical Report MS-CIS-90-69, Univ. of Pennsylvania, Dept. of Computer and Information Science, Sept. 1990.

[17] S. Gerhart, D. Craigen, and T. Ralston, "Experience with Formal Methods in Critical Systems," *IEEE Software,* vol. 11, no. 1, pp. 21-28, Jan. 1994.

[18] *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic,* M.J.C. Gordon and T.F. Melham, eds. Cambridge Univ. Press, 1993.

[19] H. Hanson, "Time and Probability in Formal Design of Distributed Systems," PhD thesis, Uppsala Univ., 1991.

[20] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi, "Hytech: The Next Generation," *Proc. IEEE Real-Time Systems Symp.,* pp. 56-65, Dec. 1995.

[21] A.T.H. Ho, "The Modechart Tableau Simulator," master's thesis, Univ. of Texas at Austin, 1994.

[22] J. Hooman, "Specification and Compositional Verification of Real-Time Systems," PhD thesis, Eindhoven Univ. of Technology, 1991.

[23] F. Jahanian and A.K. Mok, "Modechart: A Specification Language for Real-Time Systems," *IEEE Trans. Software Eng.,* vol. 20, no. 12, Dec. 1994.

[24] F. Jahanian and A.K. Mok, "Modechart: A Specification Language for Real Time Systems," *IEEE Trans. Software Eng.,* vol. 20, no. 10, Oct. 1994.

[25] F. Jahanian and D.A. Stuart, "A Method for Verifying Properties of Modechart Specifications," *Proc. IEEE Real-Time Systems Symp.,* pp. 12-21, Dec. 1988.

[26] F. Jahanian and A.K.-L. Mok, "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Trans. Software Eng.,* vol. 12, no. 9, pp. 890-904, Sept. 1986.

[27] L. Lamport, "The Temporal Logic of Actions," Technical Report 79, Digital Systems Research Center, Dec. 1991.

[28] I. Lee and 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, Jan. 1993.

[29] N. Lynch and H. Attiya, "Using Mappings to Prove Timing Properties," *Distributed Computing,* vol. 6, no. 2, pp. 121-139, 1992.

[30] D. Mandrioli, S. Marasca, and A. Morzenti, "Generating Test Cases for Real-Time Systems from Logic Specifications," *ACM Trans. Computer Systems.,* vol. 13, no. 4, pp. 365-398, Nov. 1995.

[31] Z. Manna, O. Maler, and A. Pnueli, "The Theory of Timed Automata," *Proc. REX Workshop Real-Time: Theory in Practice,* June 1991.

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

[33] S. Owre, N. Shankar, and J. Rushby, "User Guide for the PVS Specification and Verification System," technical report, Computer Science Lab., SRI Int'l, Menlo Park, Calif., 1993.

[34] L.C. Paulson, "Introduction to Isabelle," Technical Report 280, Univ. of Cambridge, Computer Laboratory, 1993.

[35] G.M. Reed and A.W. Roscoe, "A Timed Model for Communicating Sequential Processes," *Theoretical Computer Science,* pp. 249-261, 1988.

[36] 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., Feb. 1994.

[37] D.A. Stuart, "Implementing a Verifier for Real-Time Systems," *Proc. IEEE Real-Time Systems Symp.,* pp. 62-71, Dec. 1990.

[38] D.A. Stuart, "Formal Methods for Real-Time Systems," PhD thesis, The Univ. of Texas at Austin, 1996.

**Douglas A. Stuart** received the BS degree in mathematics and computer science from The Ohio State University in 1983 and the PhD and MSCS degrees in computer science from the University of Texas in 1996 and 1989, respectively, researching specification languages for real-time systems. He has been a member of the technical staff at the Microelectronics and Computer Technology Corporation in Austin, Texas and worked on architecture-based product line development and software testing. Currently, he is with the Boeing Company in St. Louis, Missouri.

**Monica Brockmeyer** received the BS and MS degrees from the University of Michigan in 1986 and 1995, respectively, and the PhD degree in electrical engineering and computer science from the University of Michigan in 1999. She is an assistant professor at Wayne State University, Detroit, Michigan. Her research interests include formal methods, monitoring and testing, and real-time and fault-tolerant computing. She is a member of the IEEE.

**Aloysius K. Mok** received the BS degree in electrical engineering and the MS and PhD degrees in computer science, from the Massachusetts Institute of Technology. Since 1983, he has been on the faculty of the Department of Computer Sciences at the University of Texas at Austin, where he is a professor of computer science. Professor Mok has done extensive research on the technical problems of real-time systems design and has consulted for both government and industry on software engineering issues of embedded systems. His current interests include the design of robust distributed real-time systems, network-centric computing, and real-time knowledge-based systems. He is a past Chairman of the Technical Committee on Real-Time Systems of the IEEE. He is also member of the IEEE.

**Farnam Jahanian** received the MS and PhD degrees in computer science from the University of Texas at Austin in 1987 and 1989, respectively. He is currently an associate professor of electrical engineering and computer science at the University of Michigan. Prior to joining the faculty at the University of Michigan in 1993, he had been a research staff member at the IBM T.J. Watson Research Center where he led several experimental projects in distributed and fault-tolerant systems. His current research interests include real-time and fault-tolerant computing and network protocols and architectures. He is a member of the IEEE.

▷ **IEEE Computer Society publications cited in this article can be found in our Digital Library at** http://computer.org/publications/dlib.