Model Checking Timing Requirements *

Joanne M. Atlee
University of Waterloo
Waterloo, Ontario

John Gannon
University of Maryland
College Park, Maryland

Abstract

Model checking has been used successfully to analyze concurrent, finite-state systems. The behavioral specification of the system is transformed into a finite representation of the specification's reachability graph. System properties to be verified are expressed as temporal logic formulae. A model checker accepts a system's reachability graph and a system property, and through exhaustive analysis determines whether or not the property holds in the system. In this paper, we extend the Software Cost Reduction (SCR) requirements notation to specify systems' timing requirements. We describe an analysis tool that transforms timed SCR specifications into timed reachability graphs, and show how some real-time properties can be verified with a model checker for branching-time temporal logic. In addition, we compare our system for analyzing SCR requirements with other model checkers that verify properties of real-time systems.

1 Introduction

Software errors frequently arise from incorrect system requirements. Successful requirements acquisition requires a thorough review process in which both domain experts and implementors can participate. Research groups have developed requirements notations with precise meanings that can be read by both groups of reviewers [14, 19]. In [3], we showed how one such notation, the Software Cost Reduction (SCR) requirements notation [14], could be analyzed using a model checker. SCR requirements are not always rigorously defined: redundant information is often excluded from the SCR behavioral requirements to enhance readability. We developed methods for detailing SCR tabular requirements with information that appears elsewhere in the SCR requirements document, translating the detailed requirements into a finite state machine that represents the system's reachability graph, and proving safety assertions with a model checker for branching-time temporal logic.

In this paper, we extend the SCR requirements notation to specify systems' timing properties. We describe an analysis tool which automates the detailing and translating steps of our analysis technique and produces input for the model checker. We also compare our system for analyzing SCR specifications with other model checkers that verify real-time properties.

At present, model checkers exist for three types of temporal logics[1] All three types of temporal logics and representative model checkers can be used to specify and analyze timing requirements.

*A previous version of this paper was presented at ISSTA'93.

[1]This paper only looks at model checkers that analyze pre-constructed reachability graphs. Model checkers that

- *Branching-time temporal logic.* In a logic with branching-time semantics, a formula is interpreted at discrete states of the system's reachability graph. In any system state, the future of a system's execution looks like a tree, where each branch represents a possible execution path. Branching-time logic formulae, when interpreted at a particular state, cannot refer to *the* next state since it is not known. Instead, the logic formulae must quantify over the set of possible next states. To prove that a formula is an invariant property of the software requirements, the property must be determined true for all states in the system's reachability graph. Examples of model checkers for branching-time logics include CTL [4, 6], TPCTL [12], XESAR [11], and SVM [5]; the Concurrency Workbench [7] also contains a CTL–based model checker.

- *Linear-time temporal logic.* In a logic with linear-time semantics, a formula is interpreted at discrete points in time with respect to a particular path in the system's reachability graph. The logic model being analyzed is an execution path: for each state, there exists a unique next state, thereby defining a unique linear execution path. To prove that a formula is an invariant property of the software requirements, the property must be interpreted over all possible execution paths in the reachability graph. Model checkers exist for the TRIO [10] and RTTL [22, 21] linear-time logics.

- *Interval temporal logic.* In a logic with interval semantics, a formula is interpreted over an interval of time. For example, in Modechart [15, 16, 18], a time interval is represented by a sequence of states that is delimited by two event occurrences. The formula to be verified, expressed in Real-Time Logic (RTL), specifies the pair(s) of events that delimit the intervals of interest. To prove that an RTL formula is an invariant property of a Modechart specification, the model checker searches the system's reachability graph for all time intervals delimited by the events referenced in the logic formula and determines whether the formula is true with respect to those intervals. HMS [9] is another interval logic for which model checking algorithms have been developed, though not implemented.

In a real-time system, the set of possible futures for a particular system state may depend on how the system reached that state. Different execution paths to the state may cause different timing constraints to be unsatisfiable in that state, thereby disabling some of the transitions leaving the state. To distinguish between system states that have different future behaviors, the reachability graph of a real-time system contains duplicate nodes, where each copy of a replicated node is annotated with the same system properties but has a different set of future behaviors. Unfortunately, this replication of graph nodes adds to the state explosion problem that plagues reachability analysis techniques; the timed reachability graphs of some concurrent systems are so large that model checking becomes inefficient.

We are able to generate and analyze timed reachability graphs that do not contain duplicate nodes. The analysis tool we are using is a model checker for Computational Tree Logic (CTL) [4]. CTL has branching-time semantics, and all of CTL's temporal operators are quantified over the set of possible futures. For example, there are two *nextstate* operators, one for asking whether property p is true in *some* next state and one for asking whether p is true in *all* next states. Because the temporal operators are quantified, the CTL model checker does not always need to

analyze symbolic representations of a system's reachability graph and checkers that construct the reachability graph during analysis were not included in this study.

distinguish between system states that have different futures: if system state x is represented by duplicate nodes and property p is true in all next states of x_i, where x_i is the i^{th} copy of node x, then p is true in all next states of x. We take advantage of the fact that CTL's temporal operators are quantified and construct timed reachability graphs in which each system state is represented by a single graph node. The disadvantage of this approach is that only a subset of CTL can be verified against our compact representation of a system's timed reachability graph.

To determine if we could verify interesting properties of existing system requirements, we used our extended SCR notation and analysis tool to analyze safety and timing requirements for two well-known small problems: a railroad crossing system and a nuclear control rods system. These problems are often used to demonstrate the effectiveness of real-time specification and verification techniques. For comparison, we analyzed the same problems using model checkers for TRIO (a temporal logic with linear-time semantics) and Modechart/RTL (a temporal logic with interval semantics). The three model checkers analyze reachability graphs of varying degrees of detail: the more timing information a reachability graph contains, the richer the logic that can be checked against it. However, a more detailed reachability graph usually means a larger graph, and a rich logic is not always needed to express a system's safety and real-time properties. We conclude the paper with some preliminary guidelines for choosing the most efficient model checker based on the types of properties one wants to verify.

2 SCR/CTL

This section describes Software Cost Reduction (SCR) requirements specifications and the Computational Tree Logic (CTL) model checker. A more formal presentation of the combined SCR/CTL methodology and how it can be used to analyze behavioral requirements appears in [2, 3]. This section extends the SCR/CTL methodology to specify and verify timing requirements.

System specification. SCR requirements specifications were developed by a research group at the Naval Research Laboratory as part of a general Software Cost Reduction project [1, 13, 14]. An SCR document specifies a software system's behavior as a finite set of concurrent, event-driven, state-transition machines called *modeclasses*. Each modeclass is composed of a set of *modes* (so named because they represent the system's different modes of operation) and transitions among the modes; at least one of the modes must be an initial mode of the modeclass. The modeclasses' sets of modes are finite and mutually disjoint. Informally, each modeclass describes one aspect of the system's behavior, and the global behavior of the entire system is defined by the composition of the system's modeclasses. The system is in exactly one mode of each modeclass at all times.

The system's environment is represented by a set of boolean environmental conditions[2]. An *event* occurs when there is a change in the values of these conditions. Event @T(A) occurs when environmental condition A becomes true; similarly, event @F(A) occurs when A becomes false. The occurrence of an event can depend on the values of other environmental conditions:

$$@\mathrm{T}(A)\ \mathrm{WHEN}\ [B]$$

occurs if A becomes true *while* B is true; more formally, the event occurs at time t if A is false and B is true at time t–1, and A and B are both true at time t. In the above event, A is called a *triggering condition* and B is called a WHEN *condition*. The model of time is discrete. Event

[2]Although conditions are boolean, first-order predicate conditions that can be represented by a finite number of boolean conditions (such as integer ranges) are also expressible.

Monitor:

Current Mode	Approaching	Train	TrainXing	In(BC,299)	In(Passed,99)	New Mode
Approach	–	@T	–	–	–	BC
BC	–	–	@T	t	–	Crossing
Crossing	–	–	@F	–	–	Passed
Passed	@T	–	–	–	t	Approach

Initial Mode: Approach (~Train)

Gate−Controller:

Current Mode	In(BC)	GateDown	In(MoveDown,19)	In(MoveDown,50)	In(Passed)	GateUp	In(MoveUp,19)	In(MoveUp,100)	New Mode
Up	@T	–	–	–	–	–	–	–	MoveDown
MoveDown	–	@T	t	f	–	–	–	–	Down
	–	–	–	@T	–	–	–	–	
Down	–	–	–	–	@T	–	–	–	MoveUp
MoveUp	@T	–	–	–	–	–	–	f	MoveDown
	–	–	–	–	–	@T	t	f	Up
	–	–	–	–	–	–	–	@T	

Initial Mode: Up

Relationships:

Approaching | Train | TrainXing
GateDown −>> ~GateUp

Figure 1: SCR requirements specification of the railroad crossing system.

occurrences trigger *mode transitions* between modes in the same modeclass. The mode transitions are instantaneous and occur at the same time as their respective transition events.

Figure 1 is an SCR requirements specification for the classic railroad crossing problem. The specification consists of two modeclasses: a MONITOR that monitors the location of the train, and a GATE-CONTROLLER that controls the position of the railroad crossing gate based on the train's location. The four MONITOR modes describe the four equivalence classes of train locations (Approach, BC (Before Crossing), Crossing, and Passed). The four GATE-CONTROLLER modes represent the four positions of the gate (Up, MoveDown, Down, and MoveUp). The initial modes of the system are Approach and Up, assuming that the initial environmental conditions satisfy predicate $\sim$*Train*; the system is not defined if a train is initially present.

SCR requirements have a tabular format. Each row in the table specifies an event causing a transition from the mode on the left to the mode on the right. Each column in the center of the table represents an environmental condition. A table entry of "@T" or "@F" represents the condition *becoming* true or *becoming* false, respectively. A table entry of "t" or "f" signifies that the condition must already *be* true or false, respectively. If the value of a condition does not affect the transition event, then the corresponding table entry is marked with a hyphen ("–"). For example, if the railroad crossing MONITOR has been in mode BC for at least 300 time units (*In(BC,299)*="t") and the train enters the crossing (*TrainXing*="@T"), then the MONITOR will transition into mode Crossing.

The set of relationship declarations at the bottom of the specification describe constraints on the values of the environmental conditions. The syntax and semantics of the relationship specifications are described in [2]; for the purposes of this paper, relation "|" denotes an enumeration and relation

"$-\gg$" denotes a type of implication. The first declaration states that at most one of the conditions representing the train's location (*Approaching*, *Train*, and *TrainXing*) can be true at any time. The second declaration states that condition *GateUp* is false whenever *GateDown* is either true or becoming true, and vice versa.

Timed system specification. We adapted van Schouwen's Inmode() and Drtn() functions [24] to represent state and timing constraints as boolean "environmental" conditions. A *state condition* specifies whether or not the system is in a particular mode. Such a condition is especially useful for synchronizing the activities of concurrent modeclasses. State conditions have format

In(*mode*)

where *mode* must be the name of a mode in one of the specification's modeclasses.

A *timing condition* specifies whether or not the system has been in a particular mode for a particular length of time. Timing conditions have format

In(*mode*, *time*)

where *mode* is the name of a mode in the specification and *time* is a positive integer. Timing conditions can be used either to synchronize concurrent modeclasses or to specify delay and deadline constraints on transitions. Delay constraints are specified by using timing conditions in WHEN clauses: the WHEN condition *In(BC,299)* in the second row of the MONITOR modeclass ensures that the transition from BC to CROSSING is delayed until the system has been in mode BC for 299 time units. Deadline constraints are expressed as negated timing conditions in WHEN clauses: WHEN condition ~*In(MoveDown,50)* in the second row of the GATE-CONTROLLER modeclass ensures that the first transition from MOVEDOWN to DOWN cannot occur if the system has been in mode MOVEDOWN for more than 50 time units. Hard deadlines are specified as unconditional events: event @T(*In(MoveDown,50)*) in the second transition from MOVEDOWN to DOWN specifies that the system must exit mode MOVEDOWN within 50 time units of entering the mode.

Timed reachability graph. We have built an analysis tool, **tcart** [2, 3], that accepts an SCR requirements specification and builds the system's reachability graph. Each node in the reachability graph (called a *global mode*) represents a composite mode consisting of exactly one mode from each of the system's modeclasses. Each edge in the graph (called a *global transition*) represents an event that causes the system to transition from one global mode to another. Initially, **tcart** ignores the mode transitions' timing constraints and generates an *untimed* reachability graph. To ensure that only reachable global modes are represented in the graph, **tcart** starts with the system's initial global modes and adds a new node to the graph if and only if there is a satisfiable transition from a reachable global mode to a new global mode that is not yet represented in the graph.

Next, **tcart** removes all global transitions whose timing constraints are not satisfiable. There are five conditions under which a transition's timing constraints cannot be satisfied:

1. The transition's delay constraint is greater than its deadline constraint.
2. The transition's delay constraint is greater than the hard deadline for leaving the transition's source global mode.
3. The transition's deadline constraint has already passed when the transition's source global mode is entered.
4. The transition's event contains a state condition *In(A)* or timing condition *In(A,t)* that must be true, but A is not a component mode of the source global mode.
5. The transition's event contains a state condition *In(A)* that must be false, but A is a component mode of the source global mode.

Whether or not a global transition's timing constraints are satisfiable depends on how long the system has been in the component modes of the transition's source global mode when this mode is entered. Let T be a global transition from global mode S to global mode D whose timing constraints are satisfiable. We use T's timing constraints and information on how long the component modes of S had been active (upon entry into S) to calculate how long the component modes of D have been active (upon entry into D). If A is a component mode of D, then the minimum and maximum amounts of time the system could have spent in A at the time D is entered ($A_D.min$ and $A_D.max$, respectively) are calculated as follows[3]:

1. If component mode A is entered when global mode D is entered, then the system has spent zero time units in A upon entering D.

   $$A_D.min = A_D.max = 0$$

2. If T's transition event contains triggering event @T(*In(A)*), then the system has spent zero time units in A upon entering D.

   $$A_D.min = A_D.max = 0$$

   Similarly, if T's transition event contains triggering event @T(*In(A,t)*), then the system has spent exactly t time units in A upon entering D.

   $$A_D.min = A_D.max = t$$

3. Otherwise, the amount of time the system has spent in A upon entering D is not exactly known, and bounds on how long the system has been in A must be determined.

   (a) If T has no timing constraints, then it is unknown how long the system has been in A upon entering D.

   $$A_D.min \geq 1$$
   $$A_D.max \leq \infty$$

   (b) If the system must exit mode A within d time units of entering A, then the maximum time spent in A upon entering mode D must be less than d:

   $$A_D.max \leq d - 1$$

   (c) If T's event contains delay constraint *In(A,t)*, then the system has spent at least $t + 1$ time units in A upon entering D:

   $$A_D.min \geq t + 1$$

   If T's event contains deadline constraint *~In(A,t)*, then the system has spent at most $t - 1$ time units in A upon entering D:

   $$A_D.max \leq t - 1$$

   (d) If T's event contains timing constraints based on component modes other than A, then the minimum amount of time spent in A upon entering D is at least as large as the minimum amount of time spent in A upon entering S ($A_S.min$) plus the minimum amount of time T is delayed ($T.delay$):

   $$A_D.min \geq A_S.min + T.delay$$

[3]If mode A is not a component mode of D, then the amount of time the system has spent in A is undefined.

The maximum amount of time spent in A upon entering D is at most as large as the maximum time spent in A upon entering S ($A_S.max$) plus the maximum abount of time T is delayed ($T.deadline$):

$$A_D.max \leq A_S.max + T.deadline$$

(e) Finally, the minimum and maximum time bounds on how long A has been active are affected by any tightening of the minimum and maximum bounds of component modes C that were entered after A was entered. The increase from $A_S.min$ to $A_D.min$ cannot be less than the increase from $C_S.min$ to $C_D.min$ for any component mode C that was entered after A:

$$A_D.min \geq MAX(\{A_S.min + (C_D.min - C_S.min) \mid C_S.max \leq A_S.max\})$$

Likewise, the increase from $A_S.max$ to $A_D.max$ cannot be more than the increase from $C_S.max$ to $C_D.max$ for any component mode C that was entered after A:

$$A_D.max \leq MIN(\{A_S.max + (C_D.max - C_S.max) \mid C_S.max \leq A_S.max\})$$

If more than one of the above rules (a)-(e) applies to the same transition T, then $A_D.min$ is the largest of all the applicable $A_D.min$ values and $A_D.max$ is the smallest of all the applicable $A_D.max$ values.

The above rules (1)-(3) calculate the bounds on how long a component mode of a destination global mode has been active with respect to a *single* global transition T. When the calculation terminates, the minimum amount of time that a component mode A has been active upon entering global mode D is the smallest of all the minimum values calculated for all the satisfiable global transitions entering D. Likewise, the maximum amount of time that A has been active upon entering D is the largest of all the maximum values calculated for all the satisfiable global transitions entering D.

Removing unsatisfiable transitions from an untimed reachability graph is an iterative process that continuously updates the timing information of the global modes' component modes based on incoming satisfiable transitions and tests exiting transitions to determine if their timing constraints are satisfiable with respect to their source global mode's current timing information. This iterative process terminates when the global modes' timing information reach a fixed point and none of the unsatisfied timing constraints are any closer to being satisfied. The result of this computation is a global, event-driven, state-transition machine that represents the specification's *timed* reachability graph.

We demonstrate our algorithm for transforming untimed reachability graphs into compact timed reachability graphs by applying the algorithm to the following SCR requirements specification. Each of the transitions is triggered by a timing condition on how long the system has been in the transition's source mode.

ModeClass1:

Current Mode	In(M1,30)	In(M2,10)	**New Mode**
M1	@T	–	M2
M2	–	@T	M1

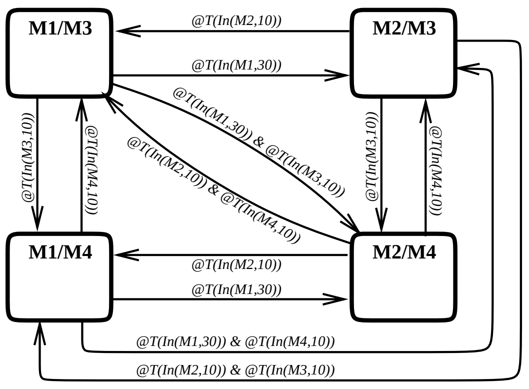
Initial Mode: M1

ModeClass2:

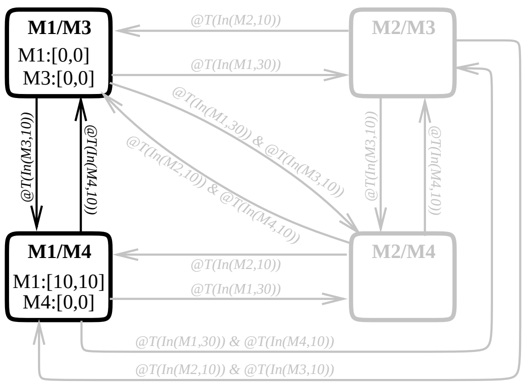
Current Mode	In(M3,10)	In(M4,10)	**New Mode**
M3	@T	–	M4
M4	–	@T	M3

Initial Mode: M3

The untimed reachability graph for the above SCR requirements specification appears below. In this example, the untimed reachability graph is the Cartesian product of the two modeclasses.


M1/M3
M2/M3
M1/M4
M2/M4
@T(In(M2,10))
@T(In(M1,30))
@T(In(M3,10))
@T(In(M4,10))
@T(In(M1,30)) & @T(In(M3,10))
@T(In(M2,10)) & @T(In(M4,10))
@T(In(M1,30)) & @T(In(M4,10))
@T(In(M2,10)) & @T(In(M3,10))

All of the transitions leaving all of the reachable global modes must be tested to determine if the transitions have satisfiable timing constraints. The transition-testing algorithm starts with set of initial global modes. When initial global mode M1/M3 is first entered, it has been in both component modes for 0 time units. Transition M1→M2 will be activated after the system has been in component mode M1 for 30 time units, and transition M3→M4 will be activated after the system has been in M3 for 10 time units. Based on how long the component modes of M1/M3 have been active upon entering the initial global mode, only transition M3→M4 has satisfiable timing constraints; the system will have exited global mode M1/M3 before the timing constraints for transition M1→M2 can be satisfied. When the system enters global mode M1/M4, it has been in mode M1 for exactly 10 time units and mode M4 for 0 time units. From global mode M1/M4, transition M1→M2 again has unsatisfiable timing constraints, and only the transition from M4 to M3 is allowed. At this point, all of the transitions leaving all of the reachable global modes have been tested. The resultant reachability graph from this first iteration is shown in Figure 2(a). The unsatisfiable transitions and the unreachable global modes are displayed in light grey.


M1/M3
M1:[0,0]
M3:[0,0]
M2/M3
M1/M4
M1:[10,10]
M4:[0,0]
M2/M4
@T(In(M2,10))
@T(In(M1,30))
@T(In(M3,10))
@T(In(M4,10))
@T(In(M1,30)) & @T(In(M3,10))
@T(In(M2,10)) & @T(In(M4,10))
@T(In(M1,30)) & @T(In(M4,10))
@T(In(M2,10)) & @T(In(M3,10))

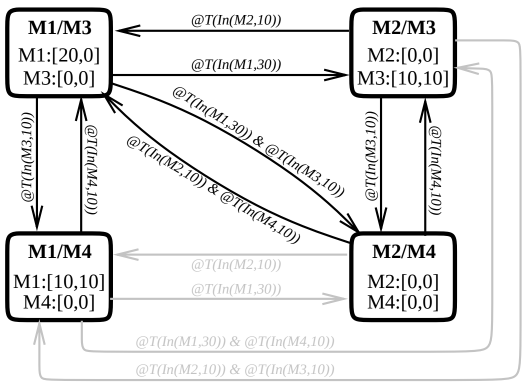
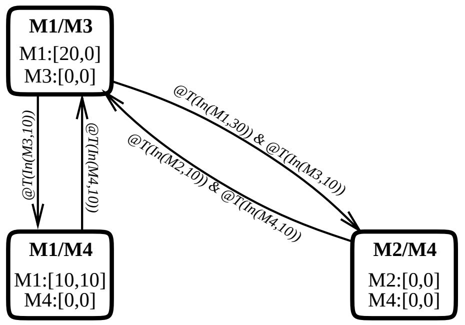


Figure 2: (a) First iteration. (b) Second iteration.

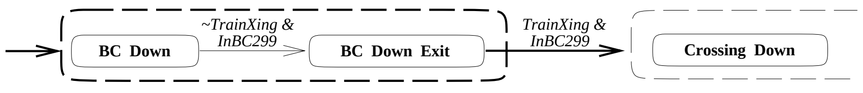
The timing information stored in global mode M1/M3 reflects the information known when the global mode's transitions were tested. Sometime after it's transitions were tested, a new transition M1/M4→M1/M3 was added to the timed reachability graph that changed the timing information of component mode M1 in M1/M3: according to the new timing information, when the system enters M1/M3 it could have been in M1 anywhere from 0 time units (if the system has just started) to 20 time units (if the global mode is entered from M1/M4). Whenever the timing information of one of the global modes changes after the global mode's transitions were tested and the changes are such that one of the unsatisfied timing constraints is closer to being satisfied, all of the transitions are retested using the all of the global modes' new timing information. In our current example, the changes to M1/M3's timing information causes the transition-testing algorithm to iterate a second time. The timed reachability graph after second iteration is shown in Figure 2(b); as before, the unsatisfiable transitions are displayed in light grey. The transition-testing algorithm terminates at the end of the second iteration because none of the unsatisfied transitions are any closer to being satisfied at the end of the iteration than when they were tested.

Finally, some of the satisfiable global transitions in the timed reachability graph may (or must) be activated as soon as the system enters their source global mode. In the above example, the global transition from M2/M3 to M2/M4 must occur immediately upon entering M2/M3 since the system has been in component mode M3 for 10 time units by the time M2/M3 is entered. In effect, events *@T(In(M1,30))* and *@T(In(M3,10))* occur at the same time and the system really transitions from M1/M3 to M2/M4. To capture this effect, sequences of transitions that occur at the same time are represented as single compound transitions from the source global mode of the first segment of the sequence to the destination global mode of the last segment of the sequence; the individual segments that compose the sequence are removed from the reachability graph. The result of collapsing all sequences of zero-time transitions in Figure 2(b) produces the following final timed reachability graph.



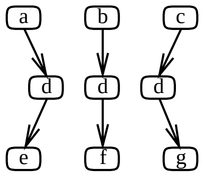
CTL machine. At this point, the requirements specification is in a format that can be formally analyzed. To use a particular analysis tool, one needs to transform the timed reachability graph into an appropriate representation that the analysis tool will accept. **tcart** converts the reachability graph into a Computational Tree Logic (CTL) machine, which can then be analyzed with the CTL model checker. Informally, a CTL machine is an extended finite state machine, in which each state is annotated with *attributes* (properties of the state) and *transition conditions* (environmental conditions). The values of the transition conditions determine which of the current state's transitions is enabled. If more than one transition can be enabled simultaneously, then the CTL machine is nondeterministic.

A CTL machine cannot naturally model events; CTL state transitions occur based on the current state and the current values of the environmental conditions. To model transitions that are activated by event occurrences, two CTL states are used to represent a global mode: a *CTL mode state* and a *CTL exit state*. The CTL states and transitions below model SCR transition BC/DOWN→CROSSING/DOWN.



The CTL mode state represents the system *in the global mode* and is annotated with the names of the global mode's component modes (e.g., BC and DOWN). The CTL exit state represents the system *leaving the global mode* due to the occurrence of an event. It is annotated with the names of the global mode's component modes plus the additional state attribute EXIT. The transition leaving the CTL exit state (and entering the CTL mode state of the transition's destination global mode) is annotated with the values of the environmental conditions *when* the transition occurs (*TrainXing* and *In(BC,299)*). The transition from the CTL mode state to its exit state is annotated with the values of the conditions *immediately before* the transition occurs (~*TrainXing* and *In(BC,299)*). The two CTL transitions together represent the event's triggering conditions changing value (@T(*TrainXing*)) while its WHEN conditions are satisfied (*In(BC,299)* is true). Multiple CTL exit states are needed to represent the events of multiple transitions leaving the same global mode.

Assertion language. The property specifications are expressed in an assertion language that we have developed and are translated into Computational Tree Logic (CTL) formulae. However, given the compact representation of our timed reachability graph, the CTL model checker is no longer sound with respect to the entire CTL language. Consider the following two sub-graphs:



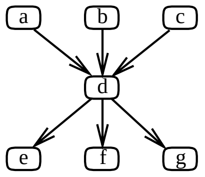


Figure 3: (a) Traditional timed reachability graph. (b) Compact timed reachability graph.

On the left is a sub-graph of a traditional timed reachability graph, in which system states that have different future behaviors are represented by multiple graph nodes. System state **d** is represented by three graph nodes, and the future behavior of the system in state **d** depends on which state preceded state **d**. For example, the future state is **e** if the preceding state was **a**. The sub-graph on the right is our compact representation of the same sub-graph. CTL formula

$$AG(a \rightarrow EX(EX(f)))$$

states that from state **a** it is possible to reach state **f** in two transitions. This formula is false in the traditional timed reachability graph but may be true in the compact timed reachability graph; the formula is true in the compact graph if all transitions leaving state **a** lead to states that have possible next states **f**.

An assertion language provides a more intuitive representation of the properties to be verified, and it restricts the user to the subset of CTL that can be validly checked against our compact timed reachability graph. The assertions used in this paper are described below. The definition of the full assertion language, including representations of the assertions in CTL, appears in Appendix A.

Strong Mode Invariance: *smi(M,p)*
: The *smi* assertion is used to express mode invariants. The assertion is true if propositional logic formula p is true *whenever* the system is in mode M.

Transition Delay Invariance: *tdelay(S,D,tc)*
: This assertion is used to express an invariant delay constraint on all transitions from mode S to mode D. The assertion states that the system cannot transition from source mode S to destination mode D if timing constraint tc is *not* true.

Mode Delay Invariance: *mdelay(M,tc)*
: This assertion is used to express an invariant delay constraint on all transitions leaving mode M. The assertion states that the system cannot exit mode M if timing constraint tc is *not* true.

Transition Deadline Invariance: *tdead(S,D,tc)*
: This assertion is used to express a hard deadline that causes the system to invariantly transition from mode S to mode D. The assertion states that if the system is in source mode S and the timing constraint tc is true (i.e., the deadline has been reached), then the system must be in destination mode D in the next system state. Note that this assertion will not hold if there exist other transitions leaving S that have the same delay constraint or no timing constraint and have a different destination mode $D2$; in such a case, the system is nondeterministic and will not *invariantly* transition to destination mode D when the deadline is reached.

Mode Deadline Invariance: *mdead(M,tc)*
: This assertion is used to express a hard deadline constraint for leaving mode M. The assertion states that if the system is in mode M and timing constraint tc is true (i.e., the deadline has been reached), then the system will *not* be in mode M in the next system state.

Some of the invariant properties that should hold in the railroad crossing example are

$$smi(Crossing, Down)$$
$$mdelay(Passed, In(Passed, 99))$$
$$tdead(MoveDown, Down, In(MoveDown, 50))$$

The first assertion is the safety property associated with the railroad crossing problem; it is a strong mode invariant that states if the train is in the railroad crossing, then the gate must be down. The second assertion is a mode delay invariant; it states that all of the transitions leaving PASSED are delayed until timing constraint *In(Passed,99)* is true. The last assertion is a transition deadline invariant; it states that the transition from MOVEDOWN to DOWN must be activated next if the system has been in mode MOVEDOWN for 50 time units.

We have created a translator that will accept a set of assertions and generate the set of equivalent CTL formulae. The set of CTL formulae can then be input to the model checker directly. Appendix A describes the mapping from our assertion language into CTL formulae. In Appendix B, we prove that the CTL model checker [4] is sound with respect to the subset of CTL we use in our assertion language and to the compact timed reachability graph we are analyzing.

Because CTL is a propositional temporal logic, there is no notion of real-time in the logic; the model checker is only capable verifying temporal boolean formulae. While the names of state and timing conditions imply that they possess some concept of time, they do not. They are simply the names of boolean conditions. Thus, we can only verify real-time formulae that reference those state and timing conditions that appear in the SCR tabular requirements. For example in the railroad crossing problem, we can verify that whenever the system enters mode PASSED, it remains in that mode for at least 100 time units; we can verify this property because the specification contains timing condition *In(Passed,99)*. We cannot verify that the system remains in PASSED for at least 50 time units under these same circumstances because the appropriate timing condition does not appear in the tabular specification.

Model checker. The MCB model checker accepts a CTL machine and a CTL formula, and determines whether or not the formula holds in the machine. The model checking algorithm determines the truth of a formula F in phases, first processing F's subformulae of length one, then F's subformulae of length two, etc., until finally processing the entire formula F. During phase i, all subformulae of length i are evaluated at each state with respect to that state's annotated propositions, its transition conditions, and its evaluations of subformulae of length less than i. Formula F is a property of the system if it is evaluated *true* in the machine's initial state.

3 Modechart

This section briefly describes the Modechart specification language and its verifier. More thorough descriptions of this system can found in [18, 23].

Timed system specification. Modechart requirements have a hierarchical, graphical representation. The modes of operation are represented by graph nodes, and mode transitions are represented by edges annotated with transition conditions. Each mode is either parallel, serial, or primitive. Parallel and serial nodes contain sets of child nodes. When a parallel node is entered, all of its children are entered simultaneously and executed independently. When a serial node is entered, one of its children (the initial node) is entered and only one child is executed at any time.

Transitions between modes may be conditioned on the occurrence of events (e.g., mode entry or action execution), the values of predicates, or delay and/or deadline constraints. A transition activated by an event occurs at the same time as the event. A transition conditioned on delay-deadline pair $(r,\ d)$ is enabled when at least r and at most d time units have passed since the transition's source mode was entered; the transition must occur after d time units have passed, if no other transition from the source mode has been activated. Mode transitions are instantaneous, and it is possible for a sequence of transitions to occur in zero time units. The Modechart specification language is actually richer than the description presented here, but the verifier will only accept specifications in the above format.

Figure 4 contains a Modechart specification for the railroad crossing problem. Machine mode MONITOR monitors the location of the train, and machine mode GATE-CONTROLLER controls the position of the crossing gate. The lower-level modes correspond to the modes used in the SCR specification of the same system. All of the transitions in the MONITOR mode are enabled by timing conditions, while those in GATE-CONTROLLER are either activated by timing conditions or by mode entry events in the MONITOR mode. The GATE-CONTROLLER transition from MOVEUP to UP is annotated with timing constraint (20,100), specifying a transition delay of 20 time units and a deadline of 100 time units. The transition from MOVEUP to MOVEDOWN is annotated with $\rightarrow$BC, indicating it is activated when the MONITOR enters BC. Delay-deadline pair (20,100) corresponds to the timing conditions *In(MoveUp,19)* and *~In(MoveUp,100)* in the

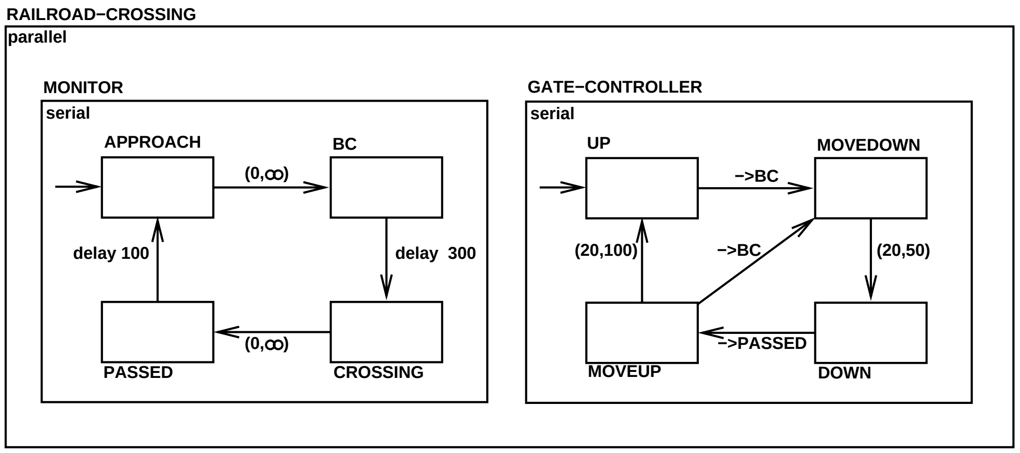


Figure 4: Modechart requirements specification of the railroad crossing system.

SCR specification[4], and mode entry event $\rightarrow$BC corresponds to triggering condition @T(*In(BC)*).

Computation graph. Modechart's verifier builds a finite computation graph representing the system's state space. Each node is labeled with the set of current system modes, the current values of state variables, the set of actions currently being performed, and a list of simultaneous events. Each edge represents a single change in the annotations of the edge's source and destination nodes (e.g., a new system mode or new variable value); thus, if multiple state changes occur simultaneously, a sequence of nodes may be entered and exited in a single time instant.

For each node N, a set of potential successors and a separation graph are generated. The separation graph nodes are nodes which lie on the path from the system's initial node to N plus N's potential successors. The separation graph edges are weighted to represent timing constraints; positive weights are delays and negative weights are deadlines. A potential successor P_i is pruned from the computation graph if a transition to another potential successor P_j must activate (due to timing constraints) before any of the transitions to P_i is enabled. Let the *distance* between two points be the maximum sum of the weights along any path from the first point to the second. Two nodes in the computation graph are equivalent if they have the same label, their sets of potential successors have the same labels, and the distances between the nodes and their potential successors (and vice versa) are identical. In such a case, one of the nodes is deleted from the graph and edges to it are replaced by edges to the other node.

Assertion language. The Modechart verifier checks a subset of Real-Time Logic (RTL), and a user-friendly interface ensures that the formulae to be verified are restricted to this subset of RTL. There are two sets of operators which specify assertions about modes and mode entry events, respectively. We only define the meanings of the formulae used in our case studies.

An M-interval is a set of consecutive nodes in a computation graph, each of which is labeled with system mode M. The following formulae state relationships between M-intervals.

cm M1 M2 evaluates to true if and only if each M1-interval contains a subsequence which rep-

[4]In this paper, one time unit is subtracted from SCR delay constraints so that a transition annotated with a *WHEN* delay constraint is enabled at the same time as a transition annotated with the "equivalent" Modechart delay.

resents an M2-interval.

xm M1 M2 evaluates to true if and only if there is no overlap between any M1-interval and M2-interval.

et M1 gives the minimum and maximum times the system spends in all its M1-intervals.

An M-interval formula is only verifyable if the modes referenced in the formula cannot starve (i.e., there is no infinite execution trace in which the modes do not occur infinitely often).

An MN-interval is a set of consecutive nodes in a computation graph that starts with a mode entry event into mode M and ends with a mode entry event into mode N. The second set of assertions supported by the Modechart verifier express relationships between MN-interval. One such formula is *iu*.

iu $\rightarrow$M1 $\rightarrow$M2 $\rightarrow$M3 $\rightarrow$M4 evaluates to true if and only if every (x^{th}) M1M2-interval is contained within some (y^{th}) M3M4-interval.

MN-interval formulae can only be used if the formula is *preserved* along every path and every cycle in the computation tree. A formula is preserved on a computation path (cycle) if and only if for each MN-interval referenced in the formula, the endpoints of that interval appear an equal number of times along that path (cycle). For example, an interval formula is not preserved if the beginning endpoint of an interval occurs before a cycle and the ending endpoint of the interval occurs within the cycle; in this case, the formula is not preserved in the cycle because after every traversal of the cycle, the beginning endpoint of the interval will have occurred more often (once more) than the endpoint of the interval.

The safety assertion we want to verify in the railroad crossing example (that the gate is down if the train is in the railroad crossing) can be expressed as either a *cm* formula or a *iu* formula.

cm(Crossing, Down)
iu($\rightarrow$Crossing, $\rightarrow$Passed, $\rightarrow$Down, $\rightarrow$MoveUp)

The first formula states that every CROSSING-interval is contained within a DOWN-interval. The second formula states that every interval delimited by entry events into modes CROSSING and PASSED is contained within an interval delimited by entry events into modes DOWN and MOVEUP.

Model checker. The decision procedures for RTL formulae are graph-search algorithms. For example, the decision procedure for **xm M1 M2** sequentially searches the computation graph nodes for all M1-intervals. For each of these, recursive searches locate all subsequent and prior M2-intervals. The distances between the entries and exits of each pair of M1- and M2-intervals are compared to determine if any of the intervals overlap[5].

4 TRIO

This section briefly describes the TRIO logic and model checker. More thorough descriptions of this system can be found in [8, 10].

[5]The decision procedure for **xm M1 M2** cannot simply search for nodes annotated with both M1 and M2. Mode entry and exit events may occur within a sequence of simultaneous events; thus, there may be graph nodes that represent the time instant that M1 (or M2) is being entered or exited but that are not labeled with M1 (or M2).

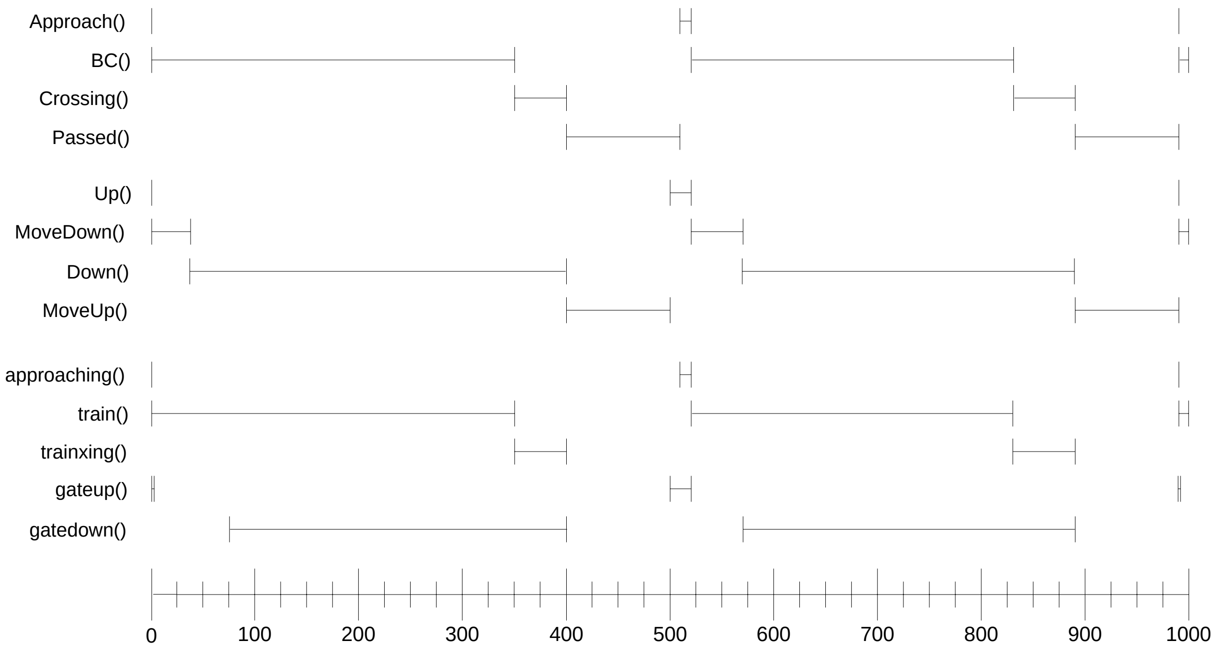


Figure 5: A TRIO history specification for the railroad crossing system.

Timed system specification. A TRIO timed system specification is a history specification that represents a possible execution of the system. Variables, predicates, and functions are either time-independent (i.e., their values are constant) or time-dependent (i.e., their values change over time). The history specification assigns invariant values to time-independent variables, predicates, and functions; and for each time-dependent variable and predicate, it assigns values for each time instant in the temporal domain.

Figure 5 depicts a TRIO history specification for one execution of the railroad crossing system. The specification uses the boolean and temporal domains and consists entirely of time-dependent predicates. There is a time-dependent predicate for each of the MONITOR modes (*Approach(), BC(), Crossing(),* and *Passed()*), each of the GATE-CONTROLLER modes (*Up(), MoveDown(), Down(),* and *MoveUp()*), each of the three possible locations of the train (*approaching(), train(),* and *trainxing()*), and each of the stable positions of the crossing gate (*gatedown()* and *gateup()*). The horizontal lines in Figure 5 depict the time instances when each predicate is true. For example, predicate *Approach()* is true during time intervals 0–1, 510–520, and 990–991. One can see that the system cycles among the modes in the two modeclasses (MONITOR and GATE-CONTROLLER), and that exactly one mode in each of the two modeclasses is true all all times, except during a mode transition when both the source and then destination modes are true.

Assertion language. TRIO is a first-order logic language, extended with temporal operators *Futr* and *Past* that define the truth of predicates in the future and past, respectively. TRIO terms are defined inductively.

- Every variable is a term.
- Every n-ary function applied to n terms is a term.

TRIO formulae are also defined inductively.

- Every n-ary predicate applied to n terms is a formula.
- If A and B are formulae, then $\neg A$ and $A \wedge B$ are formulae.
- If A is a formula and x is any time-independent variable, then $\forall x A$ is a formula.
- If A is a formula and t is a temporal term, then $Futr(A, t)$ and $Past(A, t)$ are formulae.

$Futr(A,t)$ means that formula A will be true t time units in the future. $Past(A,t)$ is defined similarly. A number of other temporal operators can be defined in terms of the *Futr* and *Past* operators [10]; the operators used in this paper are:

$$\begin{array}{lll}
AlwF(A) & =_{def} & \forall t((t > 0 \wedge Futr(true, t)) \rightarrow Futr(A, t)) \\
AlwP(A) & =_{def} & \forall t((t > 0 \wedge Past(true, t)) \rightarrow Past(A, t)) \\
Always(A) & =_{def} & AlwP(A) \wedge A \wedge AlwF(A) \\
Sometimes(A) & =_{def} & \neg AlwP(\neg A) \vee A \vee \neg AlwF(\neg A) \\
Lasted(A, t) & =_{def} & \forall t'(0 < t' < t \rightarrow Past(A, t')) \\
Until(A_1, A_2) & =_{def} & \exists t(Futr(A_2, t) \wedge (\forall t'(0 < t' < t \rightarrow Futr(A_1, t'))
\end{array}$$

The first two operators state that A will hold in every future time instant and that A was true in every past time instant, respectively. $Always(A)$ means that A holds in all time instances of the temporal domain, and $Sometimes(A)$ means that A holds in some time instant of the temporal domain. $Lasted(A,t)$ means that A has been true for the last $t-1$ time instances, not including the current time instant. As demonstrated above, TRIO is at least as expressive as temporal logic since the latter's operators (e.g., $AlwF(A)$ and $Until(A_1, A_2)$) can be defined in the former.

The following TRIO formulae represent some of the invariant properties that should hold in the railroad crossing example.

$$\begin{array}{l}
Always(Crossing() \rightarrow Down()) \\
Always((MoveDown() \wedge \neg(Lasted(MoveDown(), 20))) \rightarrow not(Futr(Down(), 1))) \\
Always((MoveDown() \wedge Lasted(MoveDown(), 50)) \rightarrow Futr(Down(), 1))
\end{array}$$

The first TRIO formula is the railroad crossing safety property: if the train is in the *Crossing()* then the gate must be *Down()*. The second formula states that if the system is in mode *MoveDown()* but has not consistently been in that mode for the last 19 time units, then the system cannot be in mode *Down()* in the next time instant. The third formula states that if the system has been in mode *MoveDown()* for the last 50 time units, then the system will be in mode *Down()* in the next time unit.

Model checker. Model checking of TRIO formulae is decidable if all of the specification's domains, including the temporal domain, are finite. A version of the tableaux algorithm is used to evaluate a formula with respect to a single state (representing an instant of time) of the history specification. To verify a formula over the entire history specification, the model checker must check the formula against all time instances, which is why the temporal domain must be finite.

5 Case Studies

To evaluate the different types of model checkers, we used the SCR/CTL, Modechart, and TRIO verification systems to analyze two existing requirements specifications. The railroad crossing specification used in this study was originally specified using Petri nets [20], and was adapted to Modechart in [23]; we created equivalent SCR and TRIO specifications. The specification for the nuclear rods control system was originally specified as a set of RTL formulae [17]; from these formulae we produced SCR, Modechart, and TRIO requirements.

Railroad crossing. This system's MONITOR and GATE–CONTROLLER modeclasses are so tightly coupled that the system's reachability graph is virtually a simple cycle. For example, the Modechart computation graph contains 13 nodes, only one of which has more than a single successor. Thus, verifying this specification was quite straightforward.

The SCR system specification for the railroad crossing example was displayed previously in Figure 1. The most important formula to verify is the safety property: if the train is in the railroad crossing, then the gate must be down. This property is expressed by the following assertion:

$$smi(Crossing, Down)$$

In addition, the specification has several delay and deadline constraints. The following assertions express invariant transition delays: if there is a delay constraint on a transition and that amount of time has not yet passed, then the transition is not allowed. For example, the first assertion states that all transitions from MoveDown to Down are delayed until the system has been in MoveDown for 19 time units.

$$tdelay(MoveDown, Down, In(MoveDown, 19))$$
$$tdelay(MoveUp, Up, In(MoveUp, 19))$$
$$tdelay(BC, Crossing, In(BC, 299))$$
$$tdelay(Passed, Approach, In(Passed, 99))$$

The next two assertions express invariant deadline requirements: if there is a hard deadline constraint on a transition and that amount of time has passed, then the transition must be activated. For example, the first assertion states that the system must transition from MoveDown to Down if it has been in mode MoveDown for 50 time units.

$$tdead(MoveDown, Down, In(MoveDown, 50))$$
$$tdead(MoveUp, Up, In(MoveUp, 100))$$

All of the safety and timing properties listed above were translated into equivalent CTL formulae and were successfully verified using the CTL model checker.

The Modechart system specification of the railroad crossing example was shown in Figure 4. Either of the following operators could be used to express system's safety property. The first formula states that every Crossing-interval is contained within a Down-interval. The second formula states that every interval that starts with an entry into mode Crossing and ends with an entry into mode Passed is contained within an interval that starts when Down is entered and ends when MoveUp is entered.

$$cm(Crossing, Down)$$
$$iu(\rightarrow Crossing, \rightarrow Passed, \rightarrow Down, \rightarrow MoveUp)$$

One can verify delay and deadline constraints using the verifier's **et** command, which determines the minimum and maximum times the system spends in any mode. The Modechart verifier calculated the following bounds on the length of time the system could spend in modes BC, Passed, MoveDown, and MoveUp.

$$et(BC) = [300, \mathrm{inf}]$$
$$et(Passed) = [100, \mathrm{inf}]$$

$et(MoveDown) = [20, 50]$
$et(MoveUp) = [20, 100]$

The TRIO history specification that we analyzed (which only represents one possible execution path) was depicted in Figure 5. The system's safety property is expressed by the following TRIO formula:

$$Always(Crossing() \rightarrow Down())$$

The formula states that whenever predicate *Crossing()* is true, predicate *Down()* must be true. This formula is valid in the history specification we chose to analyze. We were also able to verify the following delay constraints.

$$Always((BC() \wedge \sim Lasted(BC(), 300)) \rightarrow \sim Futr(Crossing(), 1))$$
$$Always((Passed() \wedge \sim Lasted(Passed(), 100)) \rightarrow \sim Futr(Approach(), 1))$$
$$Always((MoveDown() \wedge \sim Lasted(MoveDown(), 20)) \rightarrow \sim Futr(Down(), 1))$$
$$Always((MoveUp() \wedge \sim Lasted(MoveUp(), 300)) \rightarrow \sim Futr(Up(), 1))$$

The first formula states that if predicate *BC()* is true but has not been true for the last 300 consecutive time units, then predicate *Crossing()* cannot be true in next time unit. The other formulae are similarly structured. We were also able to verify the following deadline constraints.

$$Always((MoveDown() \wedge Lasted(MoveDown(), 50)) \rightarrow Futr(Down(), 1))$$
$$Always((MoveUp() \wedge Lasted(MoveUp(), 100)) \rightarrow Futr(Up(), 1))$$

These formulae state that if a mode predicate has been true for the last t consecutive time units, where t equals the transition's hard deadline constraint, then a new mode predicate must be true in the next time unit.

Nuclear control rods. The nuclear control rods system regulates the movement of two control rods in a nuclear reactor. Each rod is controlled by a separate SUBSYSTEM modeclass, and a MANAGER modeclass coordinates the operation of the two subsystems to ensure that the control rods never move at the same time. A human operator can request that a rod be moved by pushing a button. The associated SUBSYSTEM then runs tests to ensure that it is safe to move the rod, requests permission to move the rod, waits until it receives permission from the MANAGER process, and then moves the rod. Timing constraints limit the period during which a rod can be moved and the frequency with which the Manager can grant requests.

The SCR and Modechart specifications, shown in Figures 6 and 7, each contain three modeclasses running in parallel: modeclasses SUBSYS1 and SUBSYS2 describe the subsystems that operate the two control rods and modeclass MANAGER determines which of the subsystems (if any) can move its rod. The MANAGER modeclass consists of four modes: MSTART[6], STABLE (no rods are moving), GRANT1, and GRANT2. The subsystem modeclasses each have six modes, describing the various stages of a control rod's movement. The modes for modeclass SUBSYS1 are: S1NONE, S1CHECK, S1REQ (request permission to move rod), S1WAIT, S1RECGRANT (receive permission to move rod), and S1MOVEROD. Modeclass SUBSYS2 has the same modes as modeclass SUBSYS1, prefixed with S2 rather than S1.

The original specification included a safety property that stated the two control rods could not move at the same time. Two *smi* assertions are needed to express this property. Together, they imply that the system cannot be in both S1MOVEROD and S2MOVEROD at the same time.

[6]Mode MSTART is only used in the Modechart specification; it is needed to ensure that the *ui* assertions we want to verify are preserved along all cycles in the system's computation graph.

Manager:

Current Mode	In(S1Wait)	In(S2Wait)	In(Grant1,30)	In(Grant2,30)	New Mode
Stable	@T	–	–	–	Grant1
	–	@T	–	–	Grant2
Grant1	f	–	@T	–	Stable
	t	–	@T	–	Grant2
Grant2	–	f	–	@T	Stable
	–	t	–	@T	Grant1

Initial Mode: Stable

SubSys1:

Current Mode	Button1	Checked1	MakeReq1	Move1	Done1	In(S1MoveRod,20)	In(Grant1)	In(Grant1,5)	New Mode
S1None	@T	–	–	–	–	–	–	–	S1Check
S1Check	–	@T	f	–	–	–	f	–	S1Req
S1Req	–	–	@T	–	–	–	–	–	S1Wait
S1Wait	–	–	–	–	–	–	@T	–	S1RecGrant
S1RecGrant	–	–	–	@T	–	–	–	f	S1MoveRod
	–	–	–	–	–	–	–	@T	
S1MoveRod	–	–	–	–	@T	f	–	–	S1None
	–	–	–	–	–	@T	–	–	

Initial Mode: S1None

SubSys2:

Current Mode	Button2	Checked2	MakeReq2	Move2	Done2	In(S2MoveRod,20)	In(Grant2)	In(Grant2,5)	New Mode
S2None	@T	–	–	–	–	–	–	–	S2Check
S2Check	–	@T	f	–	–	–	f	–	S2Req
S2Req	–	–	@T	–	–	–	–	–	S2Wait
S2Wait	–	–	–	–	–	–	@T	–	S2RecGrant
S2RecGrant	–	–	–	@T	–	–	–	f	S2MoveRod
	–	–	–	–	–	–	–	@T	
S2MoveRod	–	–	–	–	@T	f	–	–	S2None
	–	–	–	–	–	@T	–	–	

Initial Mode: S2None

Figure 6: SCR specification of the nuclear rods control system.

$$smi(S1MoveRod, \sim S2MoveRod)$$
$$smi(S2MoveRod, \sim S1MoveRod)$$

In addition to the above assertions, we were able to verify several related invariant properties.

$$smi(S1RecGrant, \sim S2RecGrant)$$
$$smi(S2RecGrant, \sim S1RecGrant)$$
$$smi(S1RecGrant, Grant1)$$
$$smi(S2RecGrant, Grant2)$$
$$smi(S1MoveRod, Grant1)$$
$$smi(S2MoveRod, Grant2)$$

The conjunction of the first two assertions states that the two subsystems cannot have permission to move their rods at the same time. The next two assertions state that if a subsystem has received permission to move its rod, then the MANAGER has granted such permission. Finally, the last two assertions state that if a subsystem is moving its rod, then the the MANAGER has granted the subsystem permission to do so. The original specification also stated that several timing constraints must hold invariantly.

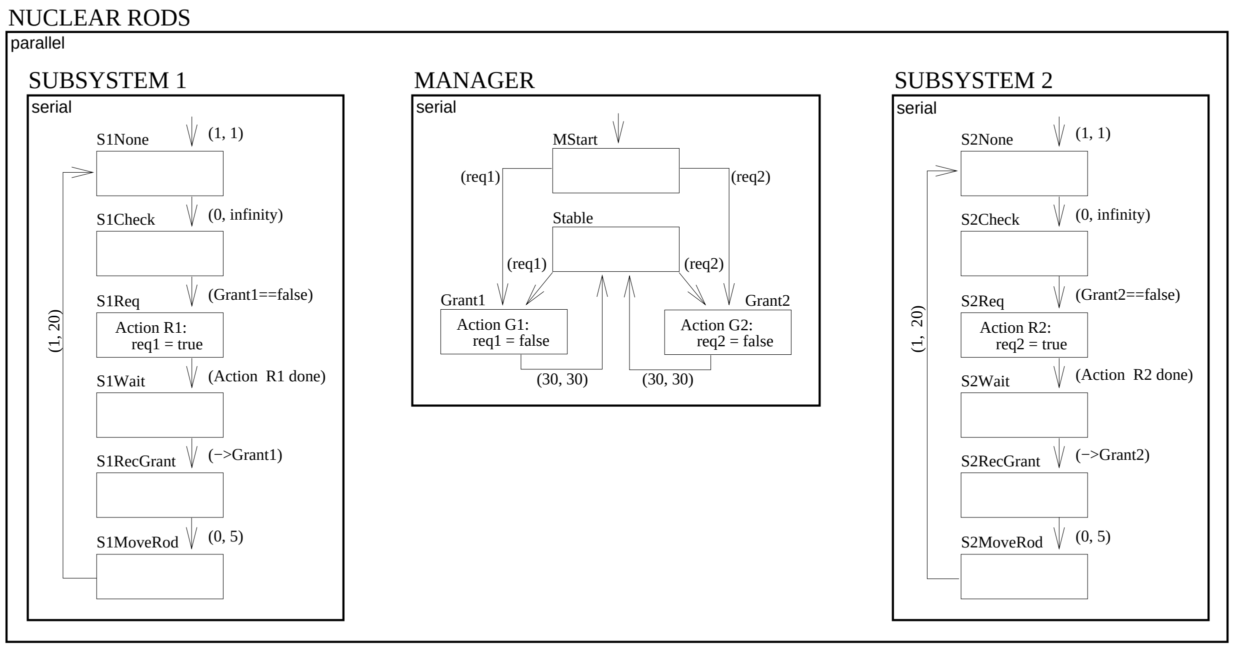


Figure 7: Modechart specification of the nuclear rods control system.

$mdelay(Grant1, In(Grant1, 30))$
$mdead(Grant1, In(Grant1, 30))$
$mdead(S1RecGrant, In(Grant1, 5))$
$mdead(S1MoveRod, In(S1MoveRod, 20))$

The first assertion states that the system must remain in mode GRANT1 until it has been in GRANT1 for 30 time units; the second assertion states that the system must exit mode GRANT1 if it has been in GRANT1 for 30 time units. Together, these two assertions state that the system always exits mode GRANT1 after the system has been in GRANT1 for *exactly* 30 time units. The third and fourth assertions state that there are also hard deadlines for leaving modes S1RECGRANT and S1MOVEROD. All of the assertions listed above (plus similar assertions expressing invariant timing properties for modes GRANT2, S2RECGRANT, and S2MOVEROD) were successfully verified against the SCR specification using the CTL model checker.

The Modechart verifier assertions that correspond to the nuclear systems' safety assertions are

$xm(S1RecGrant, S2RecGrant)$
$xm(S1MoveRod, S2MoveRod)$
$cm(S1RecGrant, Grant1)$
$cm(S2RecGrant, Grant2)$
$cm(S1MoveRod, Grant1)$
$cm(S2MoveRod, Grant2)$

The first formula states that the subsystems' RECGRANT modes are mutually exclusive (i.e., the two subsystems cannot both be in mode RECGRANT at the same time). Similarly, the subsystems' MOVEROD modes are also mutually exclusive. The second pair of formulae state that all instances of RECGRANT are contained within an instance of the appropriate GRANT mode. The last pair

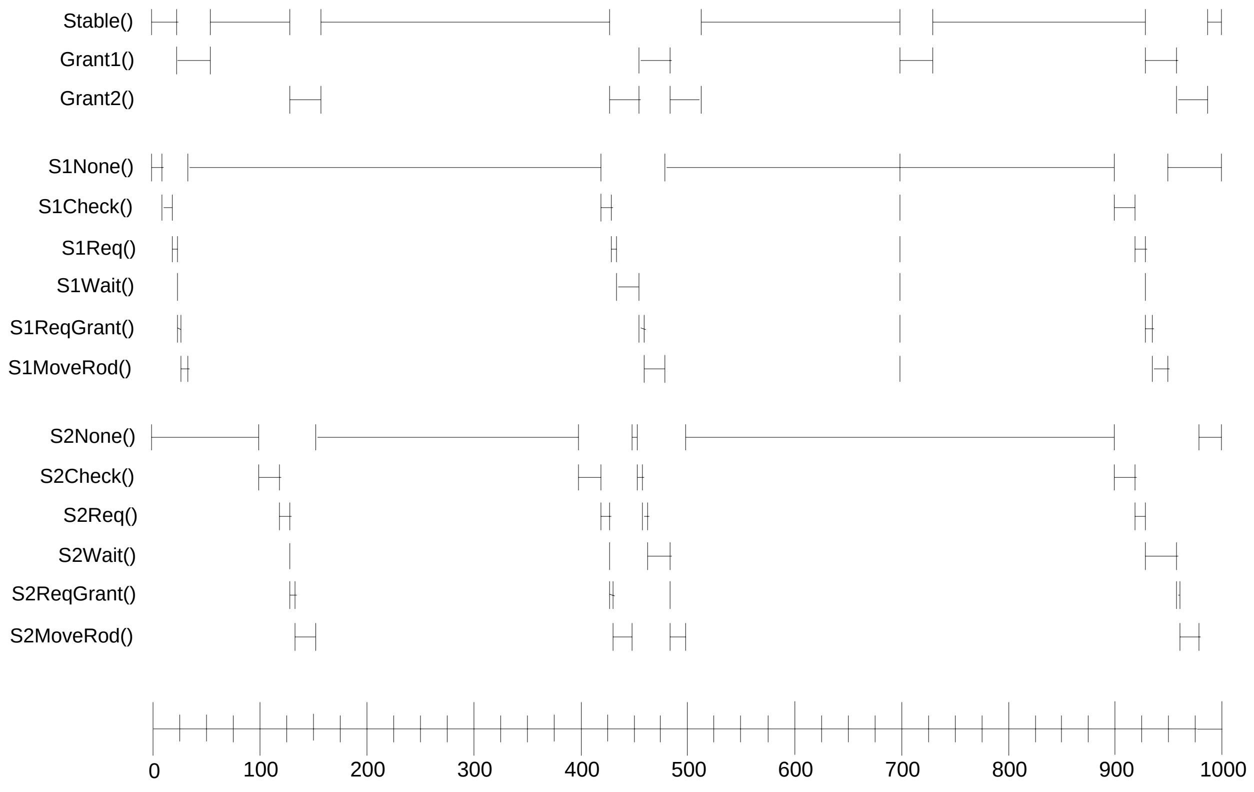


Figure 8: A TRIO history specification for the nuclear rods control system.

of formulae state that all instances of MoveRod are also contained within an instance of the appropriate Grant mode. None of these safety properties could be checked with the version of the verifier we used[7].

We were able to verify all of the system's timing constraints, using the **et** operation to calculate the minimum and maximum times the system spent in modes Grant, RecGrant, and MoveRod.

$et(Grant1) = [30, 30]$	$et(Grant2) = [30, 30]$
$et(S1RecGrant) = [0, 5]$	$et(S2RecGrant) = [0, 5]$
$et(S1MoveRod) = [1, 20]$	$et(S2MoveRod) = [1, 20]$

The TRIO history specification that we analyzed is depicted in Figure 8. We were able to verify all of the following TRIO formulae, which represent the system's safety properties.

$$Always(\sim(S1MoveRod() \wedge S2MoveRod()))$$
$$Always(\sim(S1RecGrant() \wedge S2RecGrant()))$$
$$Always(S1RecGrant() \rightarrow Grant1())$$
$$Always(S2RecGrant() \rightarrow Grant2())$$
$$Always(S1MoveRod() \rightarrow Grant1())$$
$$Always(S2MoveRod() \rightarrow Grant2())$$

[7]We used a version of the Modechart verifier which was dated February 1993. We aborted each verification attempt after 600 cpu minutes. We do not know whether the verifier would have terminated had we waited longer.

The conjunction of the first two formulae states that the system cannot be in the two MOVEROD or the two RECGRANT modes simultaneously. The next four formulae state that the SUBSYS modeclasses cannot be in their respective RECGRANT or MOVEROD modes unless the MANAGER modeclass is in the associated GRANT mode.

We were also able to verify all the required delay and deadline constraints.

$$Always((Grant1() \wedge \sim Lasted(Grant1(), 30)) \rightarrow Futr(Grant1(), 1))$$
$$Always((Grant1() \wedge Lasted(Grant1(), 30)) \rightarrow \sim Futr(Grant1(), 1))$$
$$Always((S1RecGrant() \wedge Lasted(S1RecGrant(), 5)) \rightarrow \sim Futr(S1RecGrant(), 1))$$
$$Always((S1MoveRod() \wedge Lasted(S1MoveRod(), 20)) \rightarrow \sim Futr(S1MoveRod(), 1))$$

The first two formulae state that if the system is in mode GRANT1, then the system will exit GRANT1 after *exactly* 30 time units. The last two formulae state that SUBSYS1 must exit its S1RECGRANT and S1MOVEROD modes whenever the deadlines for leaving those modes have passed. The delay and deadline constraints that regulate the execution of modeclass SUBSYS2 are similar to those given above.

6 Comparison

Each of the systems used in the above experiments have their strengths and weaknesses. Based on our experience in running these case studies, we draw the following comparisons.

Ease of property specification. The ease of property specification relates to the ease with which one can formally express the properties one wants to verify about the system specification. The specification of RTL and CTL properties is fairly straightforward because there exist intuitive assertions languages whose assertions are automatically translated into logic formulae. One of the main purposes of these assertion languages is to restrict the type of RTL and CTL formulae one can input to the respective verifiers.

More expertise is needed to formulate TRIO property specifications, since one needs to be proficient at formulating first order logic expressions. A front-end, similar to the one for the Modechart verifier or the assertion language translator we introduced in this paper, could be constructed for the TRIO model checker, which would ease the phrasing of the formulae to be verified.

Ease of system specification. SCR and Modechart system specifications are fairly easy to create and to understand because both specification languages are based on extended finite state machines. The Modechart specification language is especially convenient because one can specify persistent state variables and actions to be performed upon mode entry. However, since the verification formulae can only reference modes and mode entry events, no analysis can be performed on the values of variables or the status of actions. Furthermore, there is a problem with the semantics of Modechart that allows a state variable to be both true and false in the same time instant (in different nodes along a zero-time path)[8]. Also, a transition's timing constraints

[8]Our initial Modechart specification for the nuclear control rods system contained a zero-time transition cycle (which went undetected). We had assumed that actions R1 and R2 (which respectively assign request variables req1 and req2 the value *true*) would cause delays in the sub-system cycles because actions cannot be performed in zero time units; action R1 (R2) is started when mode S1REQ (S2REQ) is entered, and mode S1REQ (S2REQ) cannot exit until action R1 (R2) terminates. However, if the action terminates at the same time as mode S1REQ is entered (i.e., if zero time units have passed since mode S1REQ last exited), then the transition out of S1REQ is

can only depend on how long the system has been in the transition's source mode; one cannot annotate a transition **A**$\rightarrow$**B** with a delay or deadline constraint on how long the system has been in a third mode **C**.

The most difficult aspect of using the Modechart specification language is that it is possible to write specifications that cannot be verified. For example, **cm**, **xm**, and **et** formulae can only be verified if the modes referenced by these formulae cannot starve, which is possible in specifications that have infinite deadlines on transitions. Furthermore, MN-interval formulae such as **iu** can only be verified if the mode intervals referenced by the formula are preserved along every path and every cycle in the computation graph. For example, an interval formula is not preserved in a cycle if the beginning endpoint of an interval occurs before the cycle and the end of the interval occurs within the cycle: after every traversal of the cycle, the beginning endpoint of the interval will have occurred once more often than the terminating endpoint of the interval. The use of unique initial modes helps to avoid unpreserved interval formulae in most cases; we added initial mode MSTART into the MANAGER modeclass of the Modechart specification of the nuclear rods example to ensure that *ui* assertions about rod movements would be preserved. However, the introduction of unique initial modes sometimes causes other interval formulae that originally were preserved to become unpreserved.

TRIO system specifications (execution paths) are conceptually simple to create, but the process is time consuming and prone to error because the specifications are created manually. There is work on a tool that will transform ASTRAL specifications into TRIO logic formulae, and there exists another tool that will accept a TRIO formula and generate a partial history specification that satisfies the formula. The composition of these two tools may allow one to generate history specifications from ASTRAL specifications in the future.

Reachability graph. The greatest difference between SCR/CTL, Modechart, and TRIO analyses is the representation of the specifications' reachability graphs. A TRIO history specification is an execution path that consists of one node per time unit. Each node specifies which predicates are true at that time unit. The size of the specification is bounded by the number of time instances the human verifier chooses to analyze.

In a Modechart computation graph, each edge represents a single event or a single change in the system state. If multiple state changes occur simultaneously, these changes are represented by a sequence of nodes that are traversed in zero time units. Furthermore, the same system state (defined by the current modes, variable values, and active actions) may be replicated, where each copy of the state has a different set of future behaviors because a different set of timing constraints are satisfiable in that node. As a result, the size of the computation graph explodes whenever the system consists of loosely coupled processes. The nuclear control rods system, for example, contains two subsystem modeclasses that operate asynchronously; the computation graph for this system consisted of 2525 good nodes, 17,683 total nodes, and 26,204 edges[9].

In a CTL machine, each edge represents a compound state change. If multiple events occur simultaneously or if multiple transitions are concurrently activated by the same event, all the resultant state changes are collapsed into a single compound transition. In addition, each system state is represented by a single node in the system's timed reachability graph. The result is a

still enabled and the action is not restarted. We added a one unit time delay to the transition from MOVEROD to NONE to remove the zero-time cycle.

[9]We list the number of total nodes in the computation graph because some of the verification algorithms search through all of the nodes and edges.

more compact representation of the reachability graph: the CTL machine for the nuclear control rods system consisted of 393 nodes and 744 edges. As an added convenience, **tcart** detects the existence of zero-time cycles in the reachability graph and issues appropriate error messages. The Modechart and TRIO verifiers do not yet support this capability.

Analytical ability. The timed reachability graph that we generate from SCR specifications has a compact representation in which each system state is represented by one graph node. The disadvantage of compacting all of the system's behavior into this reduced state space is that the CTL model checker is no longer sound with respect to the entire CTL language. As stated above, one of the purposes of our assertion language is to restrict the property specifications to the subset of CTL that can safely be checked against a compact timed reachability graph. Appendix A defines the subset of CTL that we use, and Appendix B contains proofs that the CTL model checker is sound with respect to this subset of CTL and compact timed reachability graphs.

In a Modechart computation graph, the precise length of time a mode has been active is always known because a different graph node is used to represent each equivalence class of the possible system states. Since information about how long a mode has been active is retained in the computation graph, time bounds can be computed. For example, formula **et M1** gives the minimum and maximum times the system spends in mode M1 in a Modechart specification. However, a mode's minimum and maximum times are bounded by the delay and deadline constraints on its transitions, and SCR/CTL analysis can verify invariant delay and deadline constraints. The biggest problem we had using the Modechart verifier was the verifier's response time. On small examples like the railroad crossing system, the response time was negligible. The verifier also responded quickly when calculating timing properties of the nuclear control rods system. However, we were not able to check M-interval and MN-interval formulae against the nuclear control rods specification.

A TRIO history specification explicitly states which predicates are true at every point in time. In addition, TRIO assertions can reference the value of a predicate at any point in time. For example, one can ask if a predicate A has been true for the last 10 time units (*Lasted(A,10)*), the last 20 time units (*Lasted(A,20)*), etc. In contrast, the CTL machine that is input to the CTL model checker contains no timing information. Imprecise (but accurate) timing information is used to construct the timed reachability graph; but this information is discarded when the system's reachability graph is transformed into a CTL machine because the model checker cannot use it. As a result, arbitrary formulae about time cannot be verified against an SCR specification using the CTL model checker. Only assertions that reference timing conditions that have been declared in the system specification (e.g., *In(MoveDown,50)*) can be verified. In the above experiments, this capability was sufficient.

7 Conclusion

The most important lesson learned from these experiments is that a real-time analysis tool is not always needed to analyze real-time properties. All the safety and timing properties one wanted to verify in the railroad crossing and nuclear rods examples could be verified with respect to the systems' compact timed reachability graphs using the CTL model checker.

The type of analysis tool one needs depends on the types of formulae one wants to verify. If one wants to verify safety assertions and invariant delay and deadline requirements, the most efficient analysis technique is to generate the system's compact timed reachability graph and to analyze the graph using the CTL model checker. If one is not concerned with verifying safety properties and is most interested in a tool that calculates time bounds (as opposed to verifying time bounds), then one should generate the system's computation graph and use the Modechart verifier. If one wants to verify safety assertions, liveness assertions, and complex timing requirements, then one will need to generate a timed reachability graph (with replicated nodes) and use a model checker for a real-time logic such as TRIO or RTTL.

A Assertion Language

The syntax and semantics of the assertion language is given below. The following symbols are used in the assertion definitions.

M, S, D represent modes in one of the system's modeclasses.

$(M_1, M_2, ..., M_n)$ is a set of modes, each in a different modeclass.

tc is a timing condition (e.g., *In(BC,299)*).

p is a propositional logic formula. The modes and environmental conditions (including state and timing conditions) defined in the system's specification represent the set of atomic propositions. Every atomic proposition is a propositional logic formula. If p_1 and p_2 are propositional logic formulae, then so are $\sim p_1$, $p_1 \,\&\, p_2$, and $p_1 \mid p_2$ (where symbols $\sim$ (not), & (and), and | (or) are logic connectives and have their usual meanings).

Below are the set of assertions composing our assertion language. We assume that the reader is familiar with the syntax and semantics of CTL [6].

Strong Mode Invariance: *smi(M,p)*

The *smi* assertion is used to express mode invariants. The assertion is true if propositional logic formula p is true *whenever* the system is in mode M.

$$smi(M,p) =_{def} AG(M \rightarrow p)$$

The *smi* assertion can also be used to verify global mode invariants. In this case, the assertion is true if propositional logic formula p is true whenever the system is in the set of modes M_1, M_2, ..., and M_n. The syntax of the *smi* assertion for verifying global mode invariants is

$$smi((M_1, M_2, ..., M_n),p)$$

and its CTL representation is

$$smi((M_1, M_2, ..., M_n), p) =_{def} AG((M_1 \,\&\, ... \,\&\, M_n) \rightarrow p)$$

Weak Mode Invariance: *wmi(M,p)*

The *wmi* assertion is used to verify a weaker form of mode invariant; the assertion is true if propositional logic formula p holds when the system is in mode M, but p need not hold while the system is *exiting* mode M. For example, if event $@F(p)$ causes the system to exit mode M, assertion $wmi(M,p)$ might be true, depending on the rest of the system specification, but $smi(M,p)$ would not be true.

$$wmi(M,p) =_{def} AG((M \,\& \sim Exit) \rightarrow p)$$

The *wmi* assertion can also be used to verify a weaker form of global mode invariant. In this case, the assertion is true if propositional logic formula p holds when the system is in modes M_1, M_2, ..., M_n; however, p need not hold while the system is *exiting* one of these component modes. For example, if event $@F(p)$ causes the system to exit mode M_1, then assertion $wmi((M_1, M_2, ..., M_n), p)$ might be true, depending on the rest of the system specification, but assertion $smi((M_1, M_2, ..., M_n), p)$ would not be true. The syntax of the *wmi* assertion for verifying weak global mode invariants is

$wmi((M_1, M_2, ..., M_n), p)$

and its CTL representation is

$$wmi((M_1, M_2, ..., M_n), p) =_{def} AG((M_1 \,\&\, ... \,\&\, M_n \,\&\, \sim Exit) \rightarrow p)$$

Reachability: *reach(p)*

This assertion is used to verify that a particular relationship among the system's modes and conditions is possible.

$$reach(p) =_{def} EF(p)$$

Causality: *cause(p,M)*

The *cause* assertion is used to verify that a system property invariantly causes the system to enter (or to be in) a certain mode. The assertion is true if whenever propositional logic formula p holds, either the system is already in mode M or will be in mode M in the next system state.

$$cause(p, M) =_{def} AG(p \rightarrow (M \mid AX(M)))$$

Transition Delay Invariance: *tdelay(S,D,tc)*

This assertion is used to express an invariant delay constraint on all transitions from mode S to mode D. The assertion states that the system cannot transition from source mode S to destination mode D if timing constraint tc is *not* true.

$$tdelay(S, D, tc) =_{def} AG((S \,\&\, \sim tc) \rightarrow \sim EX(D))$$

Mode Delay Invariance: *mdelay(M,tc)*

This assertion is used to express an invariant delay constraint on all transitions leaving mode M. The assertion states that the system cannot exit mode M if timing constraint tc is *not* true.

$$mdelay(M, tc) =_{def} AG((M \,\&\, \sim tc) \rightarrow AX(M))$$

Transition Upper-Bound Invariance: *tub(S,D,tc)*

This assertion is used to express an invariant upper bound on all transitions from mode S to D. The assertion states that the system cannot transition from source mode S to destination mode D if timing constraint tc is true (i.e., if the upper bound has passed).

$$tub(S, D, tc) =_{def} AG((S \,\&\, tc) \rightarrow \sim EX(D))$$

Mode Upper-Bound Invariance: *mub(M,tc)*

This assertion is used to express an invariant upper bound on all transitions leaving mode M. The assertion states that the system cannot exit mode M if the timing constraint tc is true (i.e., if the upper bound has passed).

$$mub(M, tc) =_{def} AG((M \,\&\, tc) \rightarrow AX(M))$$

Transition Deadline Invariance: *tdead(S,D,tc)*

This assertion is used to express a hard deadline that causes the system to invariantly transition from mode S to mode D. The assertion states that if the system is in source mode S and the timing constraint tc is true (i.e., the deadline has been reached), then the system must be in destination mode D in the next system state. Note that this assertion will not hold if there exist other transitions leaving S, having a satisfiable timing constraint or no timing constraint and having a different destination mode $D2$; in such a case, the system is nondeterministic and will not *invariantly* transition to destination mode D when the deadline is reached.

$$tdead(S, D, tc) =_{def} AG((S \,\&\, tc) \rightarrow AX(D))$$

Mode Deadline Invariance: *mdead(M,tc)*

This assertion is used to express a hard deadline constraint for leaving mode M. The assertion states that if the system is in mode M and timing constraint tc is true (i.e., the deadline has been reached), then the next transition must take the system out of mode M.

$$mdead(M, tc) =_{def} AG((M \,\&\, tc) \rightarrow \sim EX(M))$$

B Soundness of CTL model checker

As system's timed reachability graph can be modeled by a 6-tuple $\mathcal{M} = \langle S, q, I, O, P, R \rangle$, where

- S is a finite set of states
- $q \in S$ is the machine's unique initial state
- I is a finite set of input propositions used to activate transitions.
- O is a finite set of output propositions used to annotated states with boolean properties. $I \cap O = \emptyset$.
- $P : S \rightarrow 2^O$ is an assignment of output propositions to states.
- $R \subseteq (S \times 2^I \times S)$ is the transition relation. If $(s, D, t) \in R$, then whenever the machine is in state s and receives input $D \in 2^I$, it may transition to state t. The term $dom(R)$ refers to the domain of relation R:

$$\forall s \in S, \forall D \in 2^I((s, D) \in dom(R) \leftrightarrow \exists t \in S((s, D, t) \in R))$$

Our **tcart** analysis tool produces a compact representation of the system's timed reachability graph. It can be modeled by a 6-tuple $\mathcal{M}^C = \langle S^C, q^C, I, O, P^C, R^C \rangle$, where

- S^C is a finite set of compact states. S^C is defined to be a partition on the set of states S such that all states $t \in S$ that are labeled with the same output propositions belong to the same block of the partition.

$$\begin{array}{ll} S^C = \{S_1, S_2, \ldots, S_{|2^O|}\} & \forall t \in S(\exists i\; 0 \leq i \leq 2^O(t \in S_i)) \\ \bigcup_{i=1}^{|2^O|} S_i = S & \forall s, t \in S((s \in S_i \wedge t \in S_i) \rightarrow P(s) = P(t)) \\ \forall i, j(i \neq j \rightarrow S_i \cap S_j = \emptyset) & \end{array}$$

  Since all states $t \in S$ that are in the same block of the partition are labeled with the same output propositions $P(t)$, we define block $S_{P(t)}$ to be the partition block containing state t:

$$\forall t \in S(\exists i\; 1 \leq i \leq 2^O(S_{P(t)} = S_i))$$
$$\forall s, t \in S(P(s) = P(t) \rightarrow S_{P(s)} = S_{P(t)})$$

- $q^C \in S^C$ is the machine's initial compact state. q^C is block $S_{P(q)}$ in partition S^C.
- I is the same set of input propositional variables.
- O is the same set of output propositional variables. $I \cap O = \emptyset$.
- $P^C : S^C \rightarrow 2^O$ is an assignment of output propositions to compact states.

$$\forall t \in S(P(t) = P^C(S_{P(t)}))$$

- $R^C \subseteq (S^C \times 2^I \times S^C)$ is the transition relation of our compact Moore machine. If there exists a transition from state $s \in S$ to $t \in S$ in machine $\mathcal{M}$, then in machine $\mathcal{M}^C$ there is a transition from the block containing s to the block containing t:
$$\forall s, t \in S, \forall D \in 2^I \left[(s, D, t) \in R \rightarrow (S_{P(s)}, D, S_{P(t)}) \in R^C\right]$$
Furthermore, if there is a transition from block $S_i \in S^C$ to block $S_j \in S^C$ in machine $\mathcal{M}^C$, then machine $\mathcal{M}$ must have a representative transition from a state in block S_i to a state in block S_j.
$$\forall S_i, S_j \in S^C, \forall D \in 2^I \left[(S_i, D, S_j) \in R^C \rightarrow \exists s \in S_i, \exists t \in S_j((s, D, t) \in R)\right]$$
We will use the term $dom(R^C)$ to refer to the domain of relation R^C:
$$\forall S_i \in S^C, \forall D \in 2^I((S_i, D) \in dom(R^C) \leftrightarrow \exists S_j \in S^C((S_i, D, S_j) \in R^C))$$

In model $\mathcal{M}$, the truth of a formula f is determined with respect to the behavior of the model $\mathcal{M}$, the current state s in the model, and a particular transition out of s annotated with some input condition D. Thus, the MCB model checker determines whether or not the following property holds:
$$\mathcal{M}, s, D \models f$$

In model $\mathcal{M}^C$, the truth of a formula f is determined with respect to the behavior of the model $\mathcal{M}^C$, the current compact state S_i in the model, and a particular transition out of S_i annotated with some input condition D. Thus, the MCB model checker determines whether or not the following property holds:
$$\mathcal{M}^C, S_i, D \models f$$

We want to verify that a formula f is true in a timed reachability graph $\mathcal{M}$ if and only if f is true in $\mathcal{M}$'s representative compact timed reachability graph $\mathcal{M}^C$. This statement is true with respect to propositional logic formulae and temporal logic formulae containing universally quantified temporal operators (e.g., AX). Thus, if f holds with respect to state s and input condition D in model M, then in model $\mathcal{M}^C$, f holds with respect to compact state $S_{P(s)}$ (which contains state s) and input condition D:
$$\mathcal{M}, s, D \models f \quad \rightarrow \quad \mathcal{M}^C, S_{P(s)}, D \models f$$

In addition, if f holds with respect to compact state S_i and input condition D in model $\mathcal{M}^C$, then in model $\mathcal{M}$, f holds respect to input condition D and all the states s belonging to partition block S_i that have transitions labeled with D:
$$\mathcal{M}^C, S_i, D \models f \quad \rightarrow \quad \forall s \in S_i((s, D) \in dom(R) \rightarrow \mathcal{M}, s, D \models f)$$

The above properties also hold for temporal logic formulae f containing existentially quantified temporal operators (e.g., EX) *if those operators are evaluated at the initial state of the model.* In general, the evaluation of existentially quantified temporal operators is not sound because the set of transitions out of states $s \in S_i$ and $t \in S_i$ may be different; thus, for example, the determination that f is true in some next state of S_i does not imply that f is true in some next state of s.

This appendix consists of proofs that the MCB model checker is sound with respect to a restricted set of the CTL branching-time temporal logic. The restricted set of CTL correponds to the set of CTL operators used in our assertion language, described in Appendix A.

The first three theorems use the property that a block in partition S^C has the same output propositions as its member states: $\forall t \in S(P(t) = P^C(S_{P(t)}))$

Theorem B.1 *An atomic proposition* ap *is true with respect to input condition D and state S_i of a compact timed reachability graph $\mathcal{M}^C$ if and only if* ap *is true with respect to input condition D and all states $s \in S_i$ that have transitions labeled D in timed reachability graph $\mathcal{M}$:*

$$\mathcal{M}^C, S_i, D \models ap \quad \leftrightarrow \quad \forall s \in S_i((s,D) \in dom(R) \rightarrow \mathcal{M}, s, D \models ap)$$

Proof:

$$\begin{aligned} \mathcal{M}, s, D \models ap &\Rightarrow ap \in D \vee ap \in P(s) \\ &\Rightarrow ap \in D \vee ap \in P^C(S_{P(s)}) \\ &\Rightarrow \mathcal{M}^C, S_{P(s)}, D \models ap \end{aligned}$$

$$\begin{aligned} \mathcal{M}^C, S_i, D \models ap &\Rightarrow ap \in D \vee ap \in P^C(S_i) \\ &\Rightarrow \forall s \in S_i((s,D) \in dom(R) \rightarrow (ap \in D \vee ap \in P(s))) \\ &\Rightarrow \forall s \in S_i((s,D) \in dom(R) \rightarrow \mathcal{M}, s, D \models ap) \end{aligned}$$

□

Theorem B.2 *An negated formula* ∼f *is true with respect to input condition D and state S_i of a compact timed reachability graph $\mathcal{M}^C$ if and only if* ∼f *is true with respect to input condition D and all states $s \in S_i$ that have transitions labeled D in timed reachability graph $\mathcal{M}$:*

$$\mathcal{M}^C, S_i, D \models \sim f \quad \leftrightarrow \quad \forall s \in S_i((s,D) \in dom(R) \rightarrow \mathcal{M}, s, D \models \sim f)$$

Proof:

$$\begin{aligned} \mathcal{M}, s, D \models \sim f &\Rightarrow (D \wedge P(s)) \rightarrow \sim f \\ &\Rightarrow (D \wedge P^C(S_{P(s)})) \rightarrow \sim f \\ &\Rightarrow \mathcal{M}^C, S_{P(s)}, D \models \sim f \end{aligned}$$

$$\begin{aligned} \mathcal{M}^C, S_i, D \models \sim f &\Rightarrow (D \wedge P^C(S_i)) \rightarrow \sim f \\ &\Rightarrow \forall s \in S_i((s,D) \in dom(R) \rightarrow ((D \wedge P(s)) \rightarrow \sim f)) \\ &\Rightarrow \forall s \in S_i((s,D) \in dom(R) \rightarrow \mathcal{M}, s, D \models \sim f) \end{aligned}$$

□

Theorem B.3 *A disjunction $f \vee g$ is true with respect to input condition D and state S_i of a compact timed reachability graph $\mathcal{M}^C$ if and only if $f \vee g$ is true with respect to input condition D and all states $s \in S_i$ that have transitions labeled D in timed reachability graph $\mathcal{M}$:*

$$\mathcal{M}^C, S_i, D \models f \vee g \quad \leftrightarrow \quad \forall s \in S_i((s,D) \in dom(R) \rightarrow \mathcal{M}, s, D \models f \vee g)$$

Proof:

$$\begin{aligned} \mathcal{M}, s, D \models f \vee g &\Rightarrow (D \wedge P(s)) \rightarrow (f \vee g) \\ &\Rightarrow (D \wedge P^C(S_{P(s)})) \rightarrow (f \vee g) \\ &\Rightarrow \mathcal{M}^C, S_{P(s)}, D \models f \vee g \end{aligned}$$

$$\begin{aligned} \mathcal{M}^C, S_i, D \models f \vee g &\Rightarrow (D \wedge P^C(S_i)) \rightarrow (f \vee g) \\ &\Rightarrow \forall s \in S_i((s,D) \in dom(R) \rightarrow ((D \wedge P(s)) \rightarrow (f \vee g))) \\ &\Rightarrow \forall s \in S_i((s,D) \in dom(R) \rightarrow \mathcal{M}, s, D \models f \vee g) \end{aligned}$$

□

The following theorems use the concept of a path in a reachability graph. Let a path in a timed reachability graph be denoted by a sequence of ordered pairs of states and input conditions

$(< t_0, D_0 >, < t_1, D_1 >, \ldots)$, where all ordered pairs $< t_i, D_i >$ belong to the domain of R and all subsequences of the path of length 2 are connectd by R (i.e., if $< t_i, D_i >, < t_{i+1}, D_{i+1} >$ appears in the path, then $(t_i, D_i, t_{i+1}) \in R$). Likewise, let a path in a compact timed reachability graph be denoted by a sequence of ordered pairs of compact states and input conditions $(< T_0, D_0 >, < T_1, D_1 >, \ldots)$, where all ordered pairs $< T_i, D_i >$ belong to the domain of R^C and all subsequences of the path of length 2 are connected by R^C (i.e., if $< T_i, D_i >, < T_{i+1}, D_{i+1} >$ appears in the path, then $(T_i, D_i, T_{i+1}) \in R^C$).

Because the following theorems use the concept of a path in a reachability graph, they use the property that if there exists a transition from state $s \in S$ to $t \in S$ in machine $\mathcal{M}$, then in machine $\mathcal{M}^C$ there is a transition from the block containing s to the block containing t:

$$\forall s, t \in S, \forall D \in 2^I \left[(s, D, t) \in R \rightarrow (S_{P(s)}, D, S_{P(t)}) \in R^C\right]$$

They also use the property that if there is a transition from block $S_i \in S^C$ to block $S_j \in S^C$ in machine $\mathcal{M}^C$, then machine $\mathcal{M}$ must have a representative transition from a state in block S_i to a state in block S_j.

$$\forall S_i, S_j \in S^C, \forall D \in 2^I \left[(S_i, D, S_j) \in R^C \rightarrow \exists s \in S_i, \exists t \in S_j((s, D, t) \in R)\right]$$

Theorem B.4 *Formula* AXf *is true with respect to input condition D and state S_i of a compact timed reachability graph $\mathcal{M}^C$ if and only if* AXf *is true with respect to input condition D and all states $s \in S_i$ that have transitions labeled D in timed reachability graph $\mathcal{M}$:*

$$\mathcal{M}^C, S_i, D \models AXf \quad \leftrightarrow \quad \forall s \in S_i((s, D) \in dom(R) \rightarrow \mathcal{M}, s, D \models AXf)$$

Proof: AXf is true with respect to input condition D in a state $s \in S$ (or $S_i \in S^C$) if and only if for all next states $t \in S$ (or all next states $T \in S^C$) reachable via a transition labeled with input condition D, f is true in state t (or state T). More precisely, f must be true with respect to all next states t (T) and all input conditions E that activate transitions out of t (T).

$$\begin{aligned}
\mathcal{M}, s, D \models AXf \;\; &\Rightarrow \;\; \forall \text{paths in } \mathcal{M} \text{ leaving state } s \text{ via a transition labeled } D \\
&\qquad (< s, D >, < t, E >, \ldots): \quad \mathcal{M}, t, E \models f \\
&\Rightarrow \;\; \forall t \in S, \forall E \in 2^I(((s, D, t) \in R \wedge (t, E) \in dom(R)) \rightarrow ((P(t) \wedge E) \rightarrow f)) \\
&\Rightarrow \;\; \forall T \in S^C, \forall E \in 2^I(((S_{P(s)}, D, T) \in R^C \wedge (T, E) \in dom(R^C)) \rightarrow \\
&\qquad ((P^C(T) \wedge E) \rightarrow f)) \\
&\Rightarrow \;\; \forall \text{paths in } \mathcal{M}^C \text{ leaving } S_{P(s)} \text{ via a transition labeled } D \\
&\qquad (< S_{P(s)}, D >, < T, E >, \ldots): \quad \mathcal{M}^C, T, E \models f \\
&\Rightarrow \;\; \mathcal{M}^C, S_{P(s)}, D \models AXf
\end{aligned}$$

$$\begin{aligned}
\mathcal{M}^C, S_i, D \models AXf \;\; &\Rightarrow \;\; \forall \text{paths in } \mathcal{M}^C \text{ leaving state } S_i \text{ via a transition labeled } D \\
&\qquad (< S_i, D >, < T, E >, \ldots): \quad \mathcal{M}^C, T, E \models f \\
&\Rightarrow \;\; \forall T \in S^C, \forall E \in 2^I(((S_i, D, T) \in R^C \wedge (T, E) \in dom(R^C)) \rightarrow \\
&\qquad ((P^C(T) \wedge E) \rightarrow f)) \\
&\Rightarrow \;\; \forall s \in S_i, \forall t \in S, \forall E \in 2^I(((s, D, t) \in R \wedge (t, E) \in dom(R)) \rightarrow \\
&\qquad ((P(t) \wedge E) \rightarrow f)) \\
&\Rightarrow \;\; \forall \text{paths in } \mathcal{M} \text{ leaving each state } s \in S_i \text{ via a transition labeled } D \\
&\qquad (< s, D >, < t, E >, \ldots): \quad \mathcal{M}, t, E \models f \\
&\Rightarrow \;\; \forall s \in S_i((s, D) \in dom(R) \rightarrow \mathcal{M}, s, D \models AXf
\end{aligned}$$

□

The following theorems use the concept of reachable states (reachable via some path in the model). The existence of a path from state s to state t in model $\mathcal{M}$ imples a path from compact state $S_{P(s)}$ to compact state $S_{P(t)}$ in model $\mathcal{M}^C$. However, the existence of a path from S_i to S_j in model $\mathcal{M}^C$ does not necessarily imply a path from some state $s \in S_i$ to some state $t \in S_j$ in model M: a path from S_i to S_j through intermediate state S_m could be representative of two disjoint paths in model M, one from $s \in S_i$ to $m \in S_k$ and the other from $n \in S_k$ to $t \in S_j$.

Fortunately, because the technique used to build a compact timed reachability graph actually traces the system's timed reachability graph, only states $s \in S$ that are reachable from the system's initial state q are represented in the compact timed reachability graph. Therefore, the existence of a path from initial compact state q^C to state S_i in model $\mathcal{M}^C$ *does* imply the existence of paths from initial state $q \in q^C$ to all states $s \in S_i$, although we do not know the sequences of ordered paths that comprise those paths. The following theorems refer to the truth of formulae AGf and EFf with respect to the initial state of the model.

Theorem B.5 *Formula* AGf *is true with respect to input condition D and initial state q^C of a compact timed reachability graph $\mathcal{M}^C$ if and only if* AGf *is true with respect to input condition D and initial state q in timed reachability graph $\mathcal{M}$:*

$$\mathcal{M}^C, q^C, D \models AGf \quad \leftrightarrow \quad \forall s \in S_i((s, D) \in dom(R) \rightarrow \mathcal{M}, q, D \models AGf)$$

Proof: AGf is true with respect to input condition D in initial state q (or q^C) if and only if for all reachable states $t \in S$ (or all reachable states $T \in S^C$) via an initial transition labeled with input condition D, f is true in state t (or state T). More precisely, f must be true with respect to all reachable states t (T) and all input conditions E that activate transitions out of t (T).

$$\begin{aligned}
\mathcal{M}, q, D \models AGf \;\; & \Rightarrow \;\; \forall \text{states } t \text{ reachable from state } q \text{ via an initial transition labeled } D \\
& \qquad (< q, D >, ..., < t, E >): \quad \mathcal{M}, t, E \models f \\
& \Rightarrow \;\; \forall t \in S \, ((\exists \text{ a path in } \mathcal{M}(< t_0, E_0 >, < t_1, E_1 >, ..., < t_n, E_n >) \wedge \\
& \qquad q = t_0 \wedge D = E_0 \wedge \forall i < n((t_i, E_i, t_{i+1}) \in R) \wedge (t_n, E_n) \in dom(R) \wedge \\
& \qquad t = t_n \wedge E = E_n \,) \quad \rightarrow ((P(t) \wedge E) \rightarrow f) \,) \\
& \Rightarrow \;\; \forall T \in S^C \left(\left(\exists \text{ a path in } \mathcal{M}^C(< S_{P(t_0)}, E_0 >, ..., < S_{P(t_n)}, E_n >) \wedge \right.\right. \\
& \qquad q = t_0 \wedge q^C = S_{P(t_0)} \wedge D = E_0 \wedge \forall i < n((S_{P(t_i)}, E_i, S_{P(t_{i+1})}) \in R^C) \wedge \\
& \qquad (S_{P(t_n)}, E_n) \in dom(R^C) \wedge T = S_{P(t_n)} \wedge E = E_n \,) \rightarrow \\
& \qquad ((P^C(T) \wedge E) \rightarrow f) \,) \\
& \Rightarrow \;\; \forall \text{states } T \text{ reachable from } q^C \text{via an initial transition labeled } D \\
& \qquad (< q^C, D >, ..., < T, E >): \quad \mathcal{M}^C, T, E \models f \\
& \Rightarrow \;\; \mathcal{M}^C, q^C, D \models AGf
\end{aligned}$$

$$\begin{aligned}
\mathcal{M}^C, q^C, D \models AGf \;\; &\Rightarrow \;\; \forall \text{states } T \text{ reachable from state } q^C \text{ via an initial transition labeled } D \\
&\qquad (< q^C, D >, ..., < T, E >): \quad \mathcal{M}^C, T, E \models f \\
&\Rightarrow \;\; \forall T \in S^C \Big(\Big(\exists \text{ a path in } \mathcal{M}^C(< T_0, E_0 >, < T_1, E_1 >, ..., < T_n, E_n >) \wedge \\
&\qquad q^C = T_0 \wedge D = E_0 \wedge \forall i < n((T_i, E_i, T_{i+1}) \in R^C) \wedge \\
&\qquad (T_n, E_n) \in dom(R^C) \wedge T = T_n \wedge E = E_n \;\Big) \quad \rightarrow ((P^C(T) \wedge E) \rightarrow f) \;\Big) \\
&\Rightarrow \;\; \forall t \in T((\exists \text{ a path in } \mathcal{M}(< t_0, E_0 >, < t_1, E_1 >, ..., < t_m, E_m >) \wedge \\
&\qquad q = t_0 \wedge D = E_0 \wedge \forall i < m((t_i, E_i, t_{i+1}) \in R) \wedge (t_m, E_m) \in dom(R) \wedge \\
&\qquad t = t_m \wedge E = E_m \;) \quad \rightarrow ((P(t) \wedge E) \rightarrow f) \;) \\
&\Rightarrow \;\; \forall \text{states } t \text{ reachable from } q \text{ via an initial transition labeled } D \\
&\qquad (< q, D >, ..., < t, E >): \quad \mathcal{M}, t, E \models f \\
&\Rightarrow \;\; \mathcal{M}, q, D \models AGf
\end{aligned}$$

□

Theorem B.6 *Formula* EFf *is true with respect to input condition D and initial state q^C of a compact timed reachability graph $\mathcal{M}^C$ if and only if* EFf *is true with respect to input condition D and initial state q in timed reachability graph $\mathcal{M}$:*

$$\mathcal{M}^C, q^C, D \models EFf \quad \leftrightarrow \quad \forall s \in S_i((s, D) \in dom(R) \rightarrow \mathcal{M}, q, D \models EFf)$$

Proof: EFf is true with respect to input condition D in initial state q (or q^C) if and only if in some state $t \in S$ (or in some state $T \in S^C$) reachable via an initial transition labeled with input condition D, f is true in state t (or state T). More precisely, f must be true with respect to some reachable state t (T) and some input condition E that activates a transition out of t (T).

$$\begin{aligned}
\mathcal{M}, q, D \models EFf \;\; &\Rightarrow \;\; \exists \text{ some state } t \text{ reachable from state } q \text{ via an initial transition labeled } D \\
&\qquad (< q, D >, ..., < t, E >): \quad \mathcal{M}, t, E \models f \\
&\Rightarrow \;\; \exists t \in S (\exists \text{ a path in } \mathcal{M}(< t_0, E_0 >, < t_1, E_1 >, ..., < t_n, E_n >) \wedge \\
&\qquad q = t_0 \wedge D = E_0 \wedge \forall i < n((t_i, E_i, t_{i+1}) \in R) \wedge (t_n, E_n) \in dom(R) \wedge \\
&\qquad t = t_n \wedge E = E_n \wedge ((P(t) \wedge E) \rightarrow f) \;) \\
&\Rightarrow \;\; \exists T = S \in S^C \big(\exists \text{ a path in } \mathcal{M}^C(< S_{P(t_0)}, E_0 >, ..., < S_{P(t_n)}, E_n >) \wedge \\
&\qquad q = t_0 \wedge q^C = S_{P(t_0)} \wedge D = E_0 \wedge \forall i < n((S_{P(t_i)}, E_i, S_{P(t_{i+1})}) \in R^C) \wedge \\
&\qquad (S_{P(t_n)}, E_n) \in dom(R^C) \wedge T = S_{P(t_n)} \wedge E = E_n \wedge \\
&\qquad ((P^C(T) \wedge E) \rightarrow f) \;) \\
&\Rightarrow \;\; \exists \text{ some state } T \text{ reachable from } q^C \text{ via an initial transition labeled } D \\
&\qquad (< q^C, D >, ..., < T, E >): \quad \mathcal{M}^C, T, E \models f \\
&\Rightarrow \;\; \mathcal{M}^C, q^C, D \models EFf
\end{aligned}$$

$$\begin{aligned}
\mathcal{M}^C, q^C, D \models EFf \;\Rightarrow\;& \exists \text{ some state } T \text{ reachable from state } q^C \text{ via an initial transition labeled } D \\
& \quad (< q^C, D >, ..., < T, E >): \quad \mathcal{M}^C, T, E \models f \\
\Rightarrow\;& \exists T \in S^C \big(\exists \text{ a path in } \mathcal{M}^C(< T_0, E_0 >, < T_1, E_1 >, ..., < T_n, E_n >) \wedge \\
& \quad q^C = T_0 \wedge D = E_0 \wedge \forall i < n((T_i, E_i, T_{i+1}) \in R^C) \wedge (T_n, E_n) \in dom(R^C) \wedge \\
& \quad T = T_n \wedge E = E_n \wedge ((P^C(T) \wedge E) \rightarrow f)\,) \\
\Rightarrow\;& \exists t \in T\,(\exists \text{ a path in } \mathcal{M}(< t_0, E_0 >, < t_1, E_1 >, ..., < t_m, E_m >) \wedge \\
& \quad q = t_0 \wedge D = E_0 \wedge \forall i < m((t_i, E_i, t_{i+1}) \in R) \wedge (t_m, E_m) \in dom(R) \wedge \\
& \quad t = t_m \wedge E = E_m \wedge ((P(t) \wedge E) \rightarrow f)\,) \\
\Rightarrow\;& \exists \text{ some state } t \text{ reachable from } q \text{ via an initial transition labeled } D \\
& \quad (< q, D >, ..., < t, E >): \quad \mathcal{M}, t, E \models f \\
\Rightarrow\;& \mathcal{M}, q, D \models EFf
\end{aligned}$$

□

References

[1] T. Alspaugh, S. Faulk, K. Britton, R. Parker, D. Parnas, and J. Shore. Software Requirements for the A-7E Aircraft. Technical report, Naval Research Laboratory, March 1988.

[2] J. Atlee. *Automated Analysis of Software Requirements*. PhD thesis, Department of Computer Science, University of Maryland, 1992.

[3] J. Atlee and J. Gannon. "State-Based Model Checking of Event-Driven System Requirements". *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[4] M. Browne. *Automatic Verification of Finite State Machines Using Temporal Logic.* PhD thesis, Department of Computer Science, Carnegie Mellon University, 1989.

[5] J. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang. "Symbolic Model Checking: 10^{20} States and Beyond". In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pages 428–439, June 1990.

[6] E. Clarke, E. Emerson, and A. Sistla. "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications". *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[7] R. Cleaveland, J. Parrow, and B Steffan. The Concurrency Workbench: Operating Instructions. Technical Report 10, Laboratory for Foundations of Computer Science, University of Edinburgh, 1988.

[8] M. Felder and A. Morzenti. "Validating Real-Time Systems by Executing Logic Specifications in TRIO". In *Proceedings of TAV4: the Symposium on Testing, Analysis, and Verification*, 1991.

[9] A Gabrielian and R. Iyer. "Integrating Automata and Temporal Logic: A Framework for Specification of Real-Time Systems and Software". In *Proceedings of the Unified Computation Laboratory*, 1990.

[10] C. Ghezzi, D. Mandrioli, and A. Morzenti. "TRIO, a logic language for executable specifications of real-time systems". *Journal of Systems and Software*, 12(2):107–123, May 1990.

[11] S. Graf, J.-L. Richier, C. Rodriguez, and J. Voiron. "What are the Limits of Model Checking Methods for the Verification of Real Life Protocols?". In *Automatic Verification Methods for Finite State Systems*, pages 275–285, June 1989.

[12] H. Hansen. *Time and Probability in Formal Design of Distributed Systems.* PhD thesis, Department of Computer Systems, Uppsala University, 1991.

[13] C. Heitmeyer and B. Labaw. "Consistency Checks for SCR-Style Requirements Specifications". (in preparation).

[14] K. Heninger. "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications". *IEEE Transactions on Software Engineering*, SE-6(1):2–12, January 1980.

[15] F. Jahanian, R. Lee, and A. Mok. "Semantics of Modechart in Real Time Logic". In *Proceedings of the 21st Hawaii International Conference on System Sciences*, pages 479–489, 1988.

[16] F. Jahanian and A. Mok. "Modechart: a Specification Language for Real-Time Systems". (to appear).

[17] F. Jahanian and A. Mok. "A Graph-Theoretic Approach for Timing Analysis and its Implementation". *IEEE Transactions on Computers*, C-36(8):961–974, August 1987.

[18] F. Jahanian and D. Stuart. "A Method for Verifying Properties of Modechart Specifications". In *Proceedings of the Real-Time Systems Symposium*, pages 12–21, 1988.

[19] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. "Requirements specification for process-control systems.". Technical report, Information and Computer Science Dept., University of California, Irvine, November 1992.

[20] N. Leveson and J. Stolzy. "Safety Analysis Using Petri Nets". *IEEE Transactions on Software Engineering*, SE-13(3):386–397, March 1987.

[21] J. Ostroff. "Deciding Properties of Timed Transition Models". *IEEE Transactions on Parallel and Distributed Systems*, 1(2):170–183, April 1990.

[22] J. Ostroff. "A Verifier for Real-Time Properties". *The Journal of Real-Time Systems*, 4:5–35, 1992.

[23] D. Stuart. Implementing a verifier for real-time systems. In *Proceedings of the Real-Time Systems Symposium*, pages 62–71, 1990.

[24] J. van Schouwen. The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application to Monitoring Systems. Technical Report TR-90-276, Department of Computing and Information Science, Queen's University, Kingston, Ontario, May 1990.