ACM Copyright Notice

© ACM 1993
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Published in: ***Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'93),*** June 1993

"Analyzing Timing Requirements"

Cite as:

Joanne M. Atlee and John Gannon. 1993. Analyzing timing requirements. In Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '93), Thomas Ostrand and Elaine Weyuker (Eds.). ACM, New York, NY, USA, 117-127.

BibTex:

```
@inproceedings{Atlee:1993:ATR:154183.154264,
 author = {Atlee, Joanne M. and Gannon, John},
 title = {Analyzing Timing Requirements},
 booktitle = {Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis},
 series = {ISSTA '93},
 year = {1993},
 pages = {117--127}
}
```

DOI: http://dx.doi.org/10.1145/154183.154264

Analyzing Timing Requirements

Joanne M. Atlee
University of Waterloo
Waterloo, Ontario

John Gannon*
University of Maryland
College Park, Maryland

1 Introduction

Software errors frequently arise from incorrect system requirements. Successful requirements acquisition requires a thorough review process in which both domain experts and implementers can participate. Research groups [6, 10] have developed notations with precise meanings that can be read by both groups of reviewers. In [3], we showed how such requirements, in particular Software Cost Reduction (SCR) requirements, could be analyzed with formal methods. 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 (representing the system's global 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 also describe an analysis tool which automates the detailing and translating steps of our analysis technique and produces input for the model checker. To determine if we could verify interesting properties of existing system requirements, we use our new notation and tool to analyze requirements for two well-known small problems. In addition to performing successful verifications of safety and timing properties of these systems, we compare our reachability graphs and formulas with those of the Modechart verifier [12], a model checker for Real-Time Logic (RTL) [7] which is based on interval semantics.

*This work was supported by the Office of Naval Research under Contract N00014-91-K-2029 and by the Air Force Office of Scientific Research under Contract AFOSR 90-0031.

2 SCR/CTL Methodology

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

System specification. SCR requirements were developed by a research group at the Naval Research Laboratory as part of a general Software Cost Reduction project [1, 5, 6]. An SCR requirements document specifies a system's behavior as a finite set of event-driven, state-transition machines that execute concurrently. Each machine i is a tuple $\langle M_i, K_i, SatE(C), \delta_i \rangle$:

- M_i is a finite set of states. The states are called *modes*, so named because they represent the system's different modes of operation. The set of modes associated with a particular machine is called a *modeclass*, and the name of the modeclass is used to refer to the machine. All of a system's modeclasses are finite and mutually disjoint.
- $K_i \subseteq M_i$ is the set of initial modes. Each modeclass has at least one initial mode, which is specified by the initial conditions of the system.
- $SatE(C)$ is the input alphabet for all the system's machines. C is a set of monitored variables (boolean conditions) that represent the system's environment[1], and $SatE(C)$ is the set of satisfiable events over set C. An *event* is a change in the environmental state. Event @T(A) occurs when environmental condition A becomes true, and event @F(A) occurs when A becomes false. The occurrence of an event can depend on the values of other environmental conditions: event @T(A) WHEN [B] occurs if A becomes true *while* B is true; that is, 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. A is the *triggering condition* and B is the WHEN

[1]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

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

condition. The input alphabet of the machines is the set of *satisfiable events* $SatE(C)$ over C, where an event is satisfiable if the conjunction of its triggering and WHEN conditions is logically satisfiable and is logically consistent with all declared constraints on the values of environmental conditions. (The declared constraints are described below.)

- $\delta_i \subseteq (M_i \times SatE(C) \times M_i)$ is the machine's transition relation. A *mode transition* occurs between modes in the same modeclass as a result of an event occurrence. Mode transitions are instantaneous and occur the at the same time as their respective transition events.
- The model of time is discrete.

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 all the system's modeclasses. The system is in exactly one mode of each modeclass at all times.

SCR requirements have a tabular format that is intuitive, easy to write and change, and scalable to large systems (e.g., the software requirements for the A7 aircraft [1]). Table 1 is a requirements specification for the classic railroad crossing problem. The specification consists of two modeclasses. The MONITOR modeclass monitors the location of the train and partitions all possible locations into four equivalence classes: APPROACH, BC (Before Crossing), CROSSING, and PASSED. The GATE-CONTROLLER controls the position of the railroad crossing gate based on the train's location; the modes represent the gate's possible positions: UP, MOVEDOWN, DOWN, 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. 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 containing an upper-case letter ('@T' or '@F') signifies the condition is a triggering condition of the transition event; the condition must change value (to true or false, respective) to activate the mode transition. A table entry containing a lower-case letter ('t' or 'f') signifies the condition is a WHEN condition of the transition event; the condition must have a particular value (true or false, respectively) both immediately before and at the time of the event occurrence. If a condition is neither a triggering condition nor a WHEN condition of a transition event, then the corresponding

table entry is marked with a hyphen ('–'). For example if the railroad crossing MONITOR is in mode APPROACH and a train is detected @T(*Train*), then the MONITOR transitions into mode BC.

We adapted van Schouwen's Inmode() and Drtn() functions [13] to represent state and timing constraints as boolean "environmental" conditions. A *state condition* specifies whether or not the system is in a particular mode. A *timing condition* specifies whether or not the system has been in a particular mode for a particular length of time. We express delay constraints as 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; for example, transition event @T(*In(MoveDown,50)*) of the second transition from MOVEDOWN to DOWN specifies that the system must exit mode MOVEDOWN within 50 time units of entering the mode.

We have built an analysis tool, **tcart** [2, 3], that transforms an SCR requirements specification into a format that can be formally analyzed. First, the SCR requirements must be detailed with missing (but known) information concerning the values of system conditions. To enhance readability of SCR requirements, events reference the least number of environmental conditions that need to be monitored to detect the event. In the railroad crossing example, an event triggered by the gate being lowered (@T(*GateDown*)) must depend on *GateUp* being false, though this may not be an explicit WHEN condition of the event. Likewise, at most one of the conditions representing the train's location (*Approaching*, *Train*, and *TrainXing*) can be true at any time. 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. **tcart** accepts a list of condition relationships and details the mode transition tables with additional triggering and WHEN conditions that explicate these relationships. In our machine mode, an event e is not satisfiable ($e \notin SatE(C)$) if it violates any of the listed condition relationships.

Second, **tcart** derives all possible sequences of simultaneous mode transitions and explicitly adds these sequences to the transition relations δ_i as "new" mode transitions. A simultaneous mode transition occurs if a transition is enabled at the time its source mode is entered; the enabled transition is immediately activated and the system effectively spends no time in the transition's source mode. For each sequence of simultaneous mode transitions, **tcart** creates a new transition from the source mode of the sequence's first transition to the destination mode of the last transition; the transition event of the new compound transition is the conjunction of all the sequence's transition events. If a modeclass contains a cycle of mode transitions that can occur simultaneously, an error message is issued.

If the specification consists of multiple concurrent modeclasses, then the next phase composes the modeclasses into a single global specification. A node in the global specification (called a *global mode*) represents several current modes, one mode from each modeclass. Starting with the set of possible initial global modes, the composition algorithm determines whether the mode transitions leaving the component modes of the reachable global modes lead to new reachable global modes. The result of composing n machines is a global, event-driven, state-transition machine $\mathcal{G} = \langle \mathcal{M}, \mathcal{K}, SatE(C), \Delta \rangle$, where

- $\mathcal{M} \subseteq (M_1 \times \cdots \times M_n)$ is the set of global modes,
- $\mathcal{K} \subseteq (K_1 \times \cdots \times K_n)$ is the set of initial global modes,
- $SatE(C)$ is the input alphabet, and
- $\Delta \subseteq (\mathcal{M} \times SatE(C) \times \mathcal{M})$ is the global transition relation.

This global machine represents the system's *un-timed* reachability graph.

Fourth, **tcart** prunes from the global system specification all global transitions whose timing constraints are not satisfiable. In the previous step, timing requirements were ignored when constructing the system's reachability graph. There are five reasons why a global transition's timing constraints might never 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 source global mode.
3. The transition's deadline constraint has already passed when the 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 G.
5. The transition's event contains a state condition In(A) that must be false, but A is a component mode of G.

For example if the railroad crossing system is in global mode BC/MoveDown, then according to the system's untimed reachability graph, the system can either transition into global mode Crossing/MoveDown via transition BC–Crossing or it can transition into BC/Down via one of the MoveDown–Down transitions. However, the system can never transition from BC/MoveDown to Crossing/MoveDown because of the transitions' timing constraints: the component modes BC and MoveDown are entered at the same time (because the only entry into MoveDown is triggering by the system's entry into BC) and the system must exit mode MoveDown before the delay constraint on transition BC–Crossing can be satisfied. To determine whether or not a global transition's timing requirements are satisfiable, one needs to know how long the system has been in the component modes of the transition's source global mode (at the time the system enters the transition's source global mode). At the same time, the calculation of how long the system has been in a global mode's component modes is based on which global transitions into that global mode are satisfiable. Therefore, the process of pruning unsatisfiable global transitions is iterative:

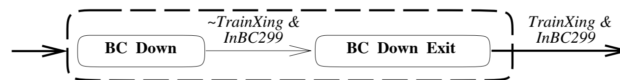
- Global transitions are tested to determine if their timing constraints are satisfiable with respect to the current information on how long the source global mode's component modes had been active upon entry into the source global mode.
- The timing information on how long a global mode's component modes had been active upon entry into the global mode is updated based on (1) the set of satisfiable global transitions entering that global mode, (2) the timing constraints of these satisfiable global transitions, and (3) the current timing information of their source global modes' component modes.

The result of this phase is a global, event-driven, state-transition machine that represents the system's *timed* reachability graph.

Finally, some of the satisfiable global transitions in the timed reachability graph have delay (or deadline) constraints of zero time units. These transitions may (or must) be activated as soon as the system enters their source global mode. The technique for collapsing simultaneous global transitions is similar to the technique for representing sequences of simultaneous mode transitions as new compound transitions, described above in **tcart**'s second step. The only difference is that in this phase, global transitions with zero-time deadline constraints are removed from the system's global transition relation. If the railroad crossing system is in global mode Approach/Up, transition Approach–BC in modeclass MONITOR causes a simultaneous transition from Up to MoveDown in modeclass GATE-CONTROLLER. The result is effectly a global transition from global mode Approach/Up to global mode BC/MoveDown. **tcart** creates a new global transition from Approach/Up to BC/MoveDown, whose transition event is @T(*Train*) & @T(*In(BC)*); the new transition is added to the system's global transition relation, and intermediate transitions Approach/Up—BC/Up and BC/Up—BC/MoveDown are removed from the relation. If the reachability graph contains a cycle of simultaneous global transitions, then an error message is issued.

At this point, the requirements specification is in a detailed 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 final event-driven, state-transition machine 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 *transition conditions* (environmental conditions) and *attributes* (properties distinct from environmental conditions). The values of the environmental 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 model a system that allows sequences of simultaneous state transitions. It is for this reason that **tcart** replaces sequences of simultaneous transitions in the SCR specification with representative compound transitions. Furthermore, 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 events, 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 an additional state attribute Exit, to indicate that the CTL state is an exit state. The transition leaving the CTL exit state (and entering the CTL mode state of the destination global mode) is anno-

tated with the values of event conditions (*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 event occurrence (WHEN condition *In(BC,299)* is true, but the triggering condition *TrainXing* is false). The two CTL transitions together represent the event's triggering conditions changing value (*TrainXing* becomes true) while its WHEN conditions are satisfied (*In(BC,299)* remains true). Multiple CTL exit states are needed to represent the events of multiple transitions from the same global mode.

Assertion language. A CTL machine can serve as a temporal logic model of a system, and a model checker can be used to test whether declarative specifications (phrased as temporal formulas) hold in the model. The declarative specifications are expressed as formulas in a propositional branching time logic called computational tree logic (CTL). CTL is defined in [4]; the syntax and semantics of the operations used in this paper are summarized below:

1. Every output proposition is an atomic CTL formula.
2. Every input condition is an atomic CTL formula.
3. If f and g are CTL formulas, then so are: $\sim f$, $f \& g$, $f \,|\, g$, AXf, EXf, EFf, AGf.

The symbols $\sim$ (*not*), & (*and*), and | (*or*) are logical connectives and have their usual meanings. Formula AXf (EXf) means that f holds in every (in some) immediate successor of the current state. F is the eventuality operator, and EFf means that along some path, there exists a future state in which f holds. G is the invariance operator, and AGf means that along every path, f holds in every state. Declarative specifications are invariant, so the formulas we want to check are of the form AGf; the other temporal operators are used to describe the properties that should be invariant.

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

$$AG(Crossing \rightarrow Down)$$
$$AG((MoveDown \ \& \sim InMoveDown19) \rightarrow \sim EX(Down))$$
$$EF(MoveDown \ \& \sim InMoveDown19)$$

The first formula is a safety property. It states that if the train is in the railroad crossing, then the gate must be down. The latter two formulas express a delay constraint. The second formula states that the transition from MOVEDOWN to DOWN cannot be activated (next) if the transition's delay constraint has not been satisfied. The paired formula precludes the possibility that the second formula is only vacuously true.

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 subformulas of length one, then F's subformulas of length two, etc., until finally processing the entire formula F. During phase i, all subformulas of length i are evaluated at each state with respect to that state's annotated propositions, its transition conditions, and its evaluations of subformulas 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 [9, 12].

System specification. Modechart is a hierarchical, graphic, requirements language. The root mode contains a set of machine modes, which represent sequential machines that execute in parallel. Each machine mode is described by a set of lower-level modes and transitions among those modes. Each lower-level mode is either a primitive mode or a set of primitive modes running in parallel, where set of primitive modes represents the set of actions that the lower-level mode must perform. The system is always in exactly one lower-level mode in each of the machine modes. Whenever a non-primitive lower-level mode is entered, all of its children begin executing. Whenever such a lower-level mode is exited, all of its children terminate[2]. Transitions between modes may be conditioned on the occurrence of events, the values of predicates, or delay and/or deadline constraints. For example, a mode transition from A to B might be activated by the system's entry into mode C (i.e., $\rightarrow$ C), or by the delay-deadline pair (20,50). 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.

Figure 1 contains a Modechart specification for the railroad crossing problem. Machine mode MONITOR monitors the location of the train, and machine mode GATE-CONTROLLER monitors and 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 between modes in the MONITOR mode are enabled by timing conditions, while those in GATE-CONTROLLER are ei-

[2] The Modechart specification language is actually less structured than the description presented here, but the verifier will only accept specifications in the above format.

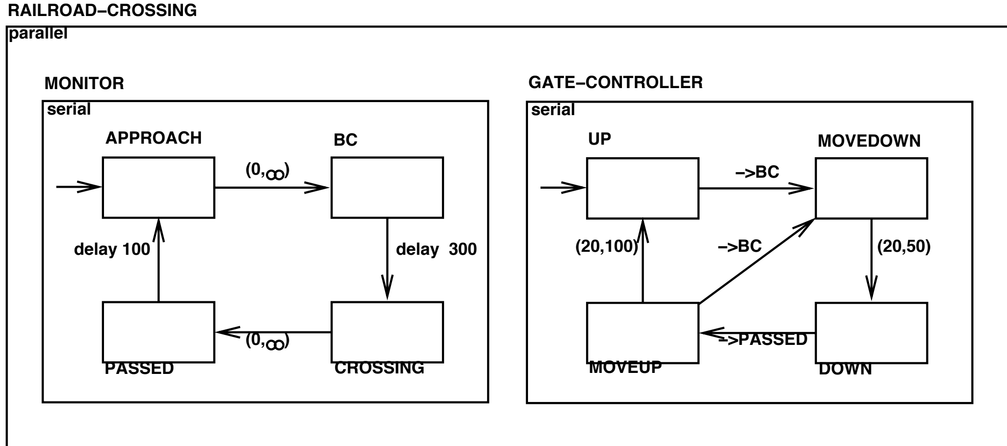


Figure 1: Modechart requirements specification of the railroad crossing system.

ther activated by timing conditions or by mode transitions in the MONITOR. In Modechart, 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 Modechart transition from MoveUp to MoveDown is annotated with →BC, indicating it is activated when the MONITOR enters BC. In the SCR specification, timing conditions *In(MoveUp,19)* and ∼*In(MoveUp100)* correspond to the delay-deadline pair (20,100)[3] and state condition *In(BC)* corresponds to the mode entry event →BC.

Modechart's verifier builds a finite computation graph representing a system's state space. The computation graph is the input to decision procedures that evaluate formulas written in its assertion language. Each node is labeled with the set of current system modes, the current values of state variables, the set of actions being performed, and a list of simultaneous events. Each edge represents one difference between its source node and destination node (e.g., a new system mode or new variable value).

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 and 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 the transitions from another potential successor P_j must activate (due to timing constraints) before any of P_i's transitions are enabled. If two nodes in the computation graph have the same label and the distances[4] between the nodes and their potential successors (and vice versa) are identical, then the two nodes are equivalent; 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 has two sets of operators which permit assertions to be written about modes and mode entry events, respectively. We define the meanings of only the formulas used in our case study.

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

cm M1 M2 evaluates to true if each M1-interval contains a subset which represents an M2-interval, and evaluates to false otherwise.

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

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

For example, 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) would be expressed as a *cm* formula.

cm(Crossing,Down)

M-interval formulas are only verifiable if the modes being compared cannot starve (i.e., there is no (infinite)

[3] We subtract one time unit from each delay constraint so that a transition annotated with a *WHEN* delay constraint is enabled at the same time the Modechart delay would be enabled.

[4] The distance between two points is the maximum sum of the weights along any path from the first point to the second.

execution trace in which either mode does 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. An event occurrence is denoted as an instantiation of the *occurrence function* '@', where $@(E,i)$ is the time of the ith occurrence of event E. For example, $@(\rightarrow M, 10)$ denotes the point in time in which the system enters mode M for the tenth time. The second set of formulas state relationships between MN-intervals. One such formula is **iu**.

iu $\rightarrow$**M1** $\rightarrow$**M2** $\rightarrow$**M3** $\rightarrow$**M4** evaluates to true if the xth entry into M1 is followed (eventually) by the xth entry into M2, the yth entry into M3 is followed by the yth entry into M4, and every (x^{th}) M1M2interval is contained within some (y^{th}) M3M4-interval.

$$\begin{aligned} \forall x \exists y [\quad & (@(\rightarrow M3, y) + c_1\ R_1\ @(\rightarrow M1, x)) \wedge \\ & (@(\rightarrow M1, x) + c_2\ <\ @(\rightarrow M2, x)) \wedge \\ & (@(\rightarrow M2, x) + c_3\ R_3\ @(\rightarrow M4, y))] \end{aligned}$$

where each c_i must be a non-negative integer and each R_i must be either $<$ or $\leq$.

The safety assertion for the railroad system can also be stated as an *iu* formula.

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

This formula states that every CROSSING-PASSED interval is contained within a DOWN-MOVEUP interval. MN-interval formulas 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).

Model checker. The decision procedures for formulas 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.

4 Case Studies

We used our timed SCR/CTL analysis tools and the Modechart verifier to analyze two existing requirements, a railroad crossing gate [11] and a nuclear rods control system [8]. The railroad crossing requirements specification used in this study was adapted to Modechart in [12]. The specification for the nuclear rods control system was originally specified as a set of RTL formulas; from these formulas we produced SCR and Modechart requirements.

Railroad crossing. The SCR and Modechart specifications of the railroad crossing systems were displayed previously in sections 2 and 3, respectively. The SCR requirements includes a list of the relationships that hold between environmental conditions:

Approaching | *Train* | *TrainXing*
GateDown $-\!\!\gg$ ~*GateUp*

The first relationship states that the train is either far away, near the crossing, or in the crossing. The second relationship states that whenever the gate is down, the gate is not up. Based on this list, **tcart** adds additional triggering and WHEN conditions to the table that explicate these relationships. For example, ~*GateUp* is added as a WHEN condition to the second row in GATE-CONTROLLER. In addition, **tcart** deduces all of the timing relationships between the state and timing conditions (e.g., In(MoveUp,100) implies In(MoveUp,19)) and adds triggering and WHEN conditions to explicit these timing relationships.

The two modeclasses are so tightly coupled that our CTL machine and the corresponding Modeclass computation graph are almost simple cycles (e.g., the computation graph contains 13 points, only one of which has more than a single successor). Thus verifying this specification was quite straightforward.

The most important formula to verify was the safety property: if the train is in the railroad crossing, then the gate must be down. This property is expressed by the following CTL formula:

$$AG(Crossing \rightarrow Down)$$

In addition, the specification had several delay and deadline constraints, among which were:

$$\begin{aligned} & AG((MoveDown\ \&\ \sim InMoveDown19) \\ & \qquad \rightarrow \sim EX(Down)) \\ & EF(MoveDown\ \&\ \sim InMoveDown19) \\ & AG(InMoveDown50 \rightarrow AX(Down)) \\ & EF(InMoveDown50) \end{aligned}$$

The first pair of CTL formulas express a delay constraint: the first formula states that the transition from MOVEDOWN to mode DOWN cannot be the next activated transition if its delay constraint has not been satisfied; the paired formula precludes the possibility that the first formula is only vacuously true. The second pair of CTL formulas represent a hard deadline constraint: if the system has been in MOVEDOWN for 50 time units, then the next transition must enter mode DOWN; the paired formula guarantees that the deadline can be reached. All five formulas were successfully verified.

The Modechart verifier provides several operators that can be used to express the safety property.

- $cm(Crossing, Down)$ states that every instance of mode *Crossing* is also an instance of mode *Down*.
- $iu(\rightarrow Crossing, \rightarrow Passed, \rightarrow Down, \rightarrow MoveUp)$ states that every interval between entering *Crossing* and entering *Passed* is contained within some interval that starts when *Down* is entered and ends when *MoveUp* is entered.

When an *iu* command evaluates to true, Modechart's verifier calculates the minimum separation between the interior and exterior intervals. In this example, the verifier finds that the interior interval (between →Crossing and →Passed) is entered at least 250 time units after the exterior interval has been entered:

$$\begin{aligned} \forall x \exists y \ & (@(\rightarrow Down, y) + 250 <= @(\rightarrow Crossing, x)) \wedge \\ & (@(\rightarrow Crossing, x) + 0 < @(\rightarrow Passed, x)) \wedge \\ & (@(\rightarrow Passed, x) + 0 <= @(\rightarrow MoveUp, y)) \end{aligned}$$

Modechart's *et* command verifies delays and deadlines by determining the minimum and maximum times the system spends in any mode. For example,

$$et(\rightarrow MoveUp)$$

calculates that whenever the system enters mode MoveUp, it remains in the mode at least 20 time units and at most 100 time units before exiting.

Nuclear control rods. The nuclear control rod example monitors the movement of two rods in a nuclear reactor. A human operator requests that a rod be moved by pushing a button. The nuclear control rods system then runs tests to ensure that it is safe to move the rod, issues a request to move the rod to the system's Manager process, waits for permission from the Manager to move the rod, and then moves the rod. Timing constraints govern the movement of rods and the frequency with which the Manager can grant requests.

Each specification contains three modeclasses running in parallel: modeclasses SUBSYS1 and SUBSYS2 describe the sub-systems that operate the two control rods and modeclass MANAGER determines which of the sub-systems (if any) can move its rod. The MANAGER modeclass consists of four modes: MStart, Stable (No rods moving.), Grant1, and Grant2. The sub-system modeclasses each have seven modes, describing the various stages of a control rod's movement. The modes for modeclass SUBSYS1 are: S1Start, S1None (waiting for a button press), S1Check, S1Req (ask for 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. None of the environmental conditions are related.

The three modeclasses are not nearly as tightly coupled as the modeclasses in the railroad crossing example were. The CTL machine representation of the nuclear rods control system consists of 393 states and 744 transitions. The computation tree of our initial Modechart specification consisted of 4003 nodes and 37406 edges, and contained a zero-time transition cycle (which went undetected). We had assumed that actions R1 and R2 (which assign request variables req1 and req2 to 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), the system immediately transitions out of S1Req and the action is not restarted. Adding a one unit time delay to the transition from MoveRod to None removed the zero-time cycle and reduced the number of nodes to 1157 and the number of edges to 10218.[5]

The RTL specification included a safety property that stated the two control rods could not move at the same time. The CTL and Modechart verifier formulas that correspond to this safety assertion are

CTL:	$AG(\sim(S1MoveRod \,\&\, S2MoveRod))$
Modechart:	$xm(S1MoveRod, S2MoveRod)$

We verified the CTL formula with the CTL model checker within two seconds wall time. We tried to verify the *xm* formula, but the Modechart verifier did not complete its evaluation given more than 2-3 hours wall time We do not know whether it would have terminated if we had waited longer.

We were able to verify several other interesting invariants with the CTL model checker, but not with the Modechart verifier. Among these assertions were that if a sub-system is moving its rod, then the Manager must be in a mode indicating that it has granted that sub-system permission to move its rod; and the two sub-systems cannot both be in their respective RecGrant modes simultaneously.

Using either the CTL model checker or Modechart, we were able to verify all the delay and deadline constraints. The delay constraints insisted if the Manager was in a Grant mode that it remain there for at least 30 time units. The deadline constraints were that the Manager exit a Grant mode after 30 time units, that each sub-system must start moving its rod within 5 time units of receiving permission to do so, and that

[5] Not all of the edges belong to the pruned computation graph, but the verification algorithms look at them all.

SCR

Manager:

Current Mode	*In*(*S1Wait*)	*In*(S2*Wait*)	*In*(*Grant1*,30)	*In*(*Grant*2,30)	New Mode
Stable	@T	–	–	–	Grant1
	–	@T	–	–	Grant2
Grant1	f	–	@T	–	Stable
	t	–	@T	–	Grant2
Grant2	–	f	–	@T	Stable
	–	t	–	@T	Grant1

Initial Mode: Stable

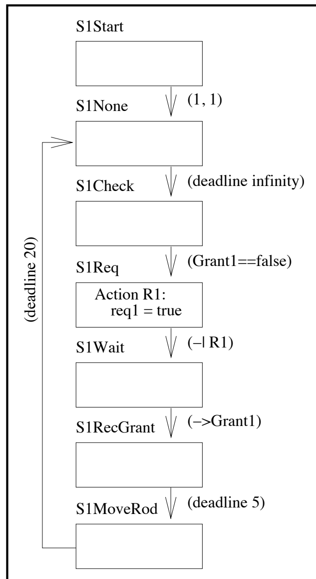
SubSys1:

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	–	–	

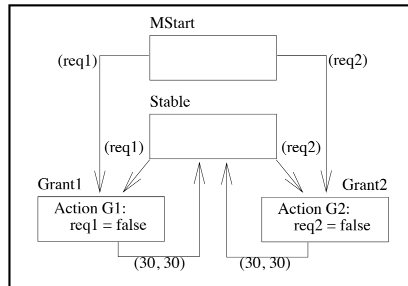
Initial Mode: S1None

MODECHART

SUBSYSTEM 1



MANAGER



SUBSYSTEM 2

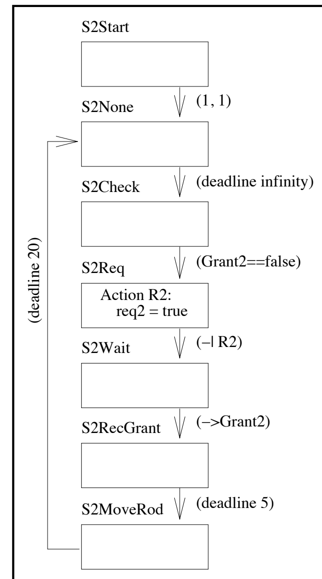


Figure 2: SCR and Modechart specifications of the nuclear rods control system.

each sub-system could only move its rod for 20 time units.

5 Conclusion

The greatest difference between SCR/CTL and Modechart analysis is the construction of the reachability graphs. In a CTL machine, a global SCR mode is represented by one CTL mode state plus a CTL exit state for each satisfiable global transition leaving the global mode. Imprecise (but accurate) timing information about how long a global mode's component modes have been active is used to construct the reachability graph, but this information is absent from the final representative CTL machine. As a result, arbitrary formulas about time cannot be verified using a CTL machine representation.

In a Modechart computation graph, the precise length of time a global mode's component modes have been active is always known because a different graph node is used to represent each equivalence class of possibilities. If one component mode can be active for up to 100 time units and another component mode can also be active for up to 100 time units, the global mode could be represented by up to 10,000 computation graph nodes. As a result, the size of the computation graph explodes whenever the system consists of loosely coupled machines.

Also, each edge in a Modechart computation graph represents a single event. Sequences of simultaneous events are represented by paths that are traversed in zero time units. In a CTL machine, such zero-time paths are collapsed into a single compound transition which represents many state changes. **tcart** detects the existence of zero-time cycles in the specification and issues appropriate error messages. The Modechart verifier does not yet support this capability.

Since information about how long each component mode has been active is retained in the Modechart computation graph, time bounds can be computed. For example, formula **et(M)** gives the minimum and maximum times the system spends in mode **M**. 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 a mode's delays and deadlines. In our experiments, this capability was sufficient.

The greatest difficulty with the SCR/CTL analysis technique involves the construction of the CTL formulas to be checked. One of the strengths of the Modechart verifier is the compact representation of its assertion language. The main purpose of this language is to restrict the type of RTL formulas one can input to the verifier. A similar front-end could be constructed for the CTL model checker that would ease the phrasing of CTL formulas to be verified.

Another strength of the Modechart specification language is that one can specify persistent state variables and actions to be performed upon mode entry. However, since the verification formulas can only reference modes and mode entry events, no analysis can be performed on the values of variables or the status of actions. Furthermore, each computation graph node is annotated with current values of the state variables and the currently activated actions; this further increases the size of the computation graph without increasing the power of the verifier. Most importantly, 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).

The greatest difficulty with using the Modechart verifier is that it is too easy to write specifications that cannot be verified. **cm**, **um**, and **et** formulas can only be verified if modes referenced by these formulas cannot starve (which is possible in specifications having infinite deadlines on transitions). Interval formulas 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. 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. The use of unique initial modes (that are never entered again) helps avoid unpreserved formulas, but this is not a theorem.

The other major problem we had using the Modechart verifier was the response time. On small examples like the railroad crossing system, the response time was negligible. Also, the verifier responded quickly when calculating timing properties of the nuclear rods control system. However, we were not able to evaluate M-interval and MN-interval formulas. It is not known whether the verification routines contain an infinite loop or are simply computationally expensive. We should also note that all of our experiments were run using versions of the Modechart verifier that were available in November 1992 and February 1993. A new version of the verifier was released in March 1993, but there was not time to evaluate it for this publication.

The most important lesson learned from these experiments is that a real-time analysis tool is not always needed to analyze real-time systems. All safety and timing properties one wanted to verify in the railroad crossing and nuclear rods examples could be verified using the CTL model checker. More work needs to be done to determine additional real-time properties that require more powerful verifiers.

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. (to appear).

[4] 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.

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

[6] 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.

[7] F. Jahanian and A. Mok. "Safety Analysis of Timing Properties in Real-Time Systems". *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.